To Ravit, Alma and Goren.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>19</td>
</tr>
<tr>
<td>Preliminaries</td>
<td>27</td>
</tr>
<tr>
<td>0 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>1 Mathematical Background</td>
<td>47</td>
</tr>
<tr>
<td>2 Computation and Representation</td>
<td>85</td>
</tr>
<tr>
<td>I Finite computation</td>
<td>117</td>
</tr>
<tr>
<td>3 Defining computation</td>
<td>119</td>
</tr>
<tr>
<td>4 Syntactic sugar, and computing every function</td>
<td>151</td>
</tr>
<tr>
<td>5 Code as data, data as code</td>
<td>171</td>
</tr>
<tr>
<td>II Uniform computation</td>
<td>195</td>
</tr>
<tr>
<td>6 Loops and infinity</td>
<td>197</td>
</tr>
<tr>
<td>7 Equivalent models of computation</td>
<td>231</td>
</tr>
<tr>
<td>8 Universality and uncomputability</td>
<td>269</td>
</tr>
<tr>
<td>9 Restricted computational models</td>
<td>295</td>
</tr>
<tr>
<td>10 Is every theorem provable?</td>
<td>325</td>
</tr>
<tr>
<td>III Efficient algorithms</td>
<td>341</td>
</tr>
<tr>
<td>11 Efficient computation</td>
<td>343</td>
</tr>
</tbody>
</table>
12 Modeling running time
13 Polynomial-time reductions
14 NP, NP completeness, and the Cook-Levin Theorem
15 What if P equals NP?
16 Space bounded computation

IV Randomized computation
17 Probability Theory 101
18 Probabilistic computation
19 Modeling randomized computation

V Advanced topics
20 Cryptography
21 Proofs and algorithms
22 Quantum computing

VI Appendices
A The NAND Programming Language
B The NAND++ Programming Language
C The Lambda Calculus
# Contents (detailed)

**Preface**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 To the student</td>
<td>21</td>
</tr>
<tr>
<td>0.1.1 Is the effort worth it?</td>
<td>21</td>
</tr>
<tr>
<td>0.2 To potential instructors</td>
<td>22</td>
</tr>
<tr>
<td>0.3 Acknowledgements</td>
<td>24</td>
</tr>
</tbody>
</table>

**Preliminaries**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Introduction</td>
<td>29</td>
</tr>
<tr>
<td>0.1 Extended Example: A faster way to multiply</td>
<td>32</td>
</tr>
<tr>
<td>0.1.1 Beyond Karatsuba’s algorithm</td>
<td>36</td>
</tr>
<tr>
<td>0.2 Algorithms beyond arithmetic</td>
<td>38</td>
</tr>
<tr>
<td>0.3 On the importance of negative results</td>
<td>39</td>
</tr>
<tr>
<td>0.4 Roadmap to the rest of this course</td>
<td>40</td>
</tr>
<tr>
<td>0.4.1 Dependencies between chapters</td>
<td>41</td>
</tr>
<tr>
<td>0.5 Exercises</td>
<td>42</td>
</tr>
<tr>
<td>0.6 Bibliographical notes</td>
<td>44</td>
</tr>
<tr>
<td>0.7 Further explorations</td>
<td>44</td>
</tr>
</tbody>
</table>

**1 Mathematical Background**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 A mathematician’s apology</td>
<td>47</td>
</tr>
<tr>
<td>1.2 A quick overview of mathematical prerequisites</td>
<td>48</td>
</tr>
<tr>
<td>1.3 Reading mathematical texts</td>
<td>50</td>
</tr>
<tr>
<td>1.3.1 Example: Defining a one to one function</td>
<td>52</td>
</tr>
<tr>
<td>1.4 Basic discrete math objects</td>
<td>54</td>
</tr>
<tr>
<td>1.4.1 Sets</td>
<td>54</td>
</tr>
<tr>
<td>1.4.2 Sets in Python (optional)</td>
<td>55</td>
</tr>
<tr>
<td>1.4.3 Special sets</td>
<td>56</td>
</tr>
<tr>
<td>1.4.4 Functions</td>
<td>57</td>
</tr>
<tr>
<td>1.4.5 Graphs</td>
<td>60</td>
</tr>
<tr>
<td>1.4.6 Logic operators and quantifiers</td>
<td>62</td>
</tr>
<tr>
<td>1.4.7 Quantifiers for summations and products</td>
<td>63</td>
</tr>
<tr>
<td>1.4.8 Parsing formulas: bound and free variables</td>
<td>63</td>
</tr>
<tr>
<td>1.4.9 Asymptotics and Big-Ø notation</td>
<td>65</td>
</tr>
</tbody>
</table>
## Index

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.4.10</td>
<td>Some “rules of thumb” for Big-$O$ notation</td>
<td>67</td>
</tr>
<tr>
<td>1.5</td>
<td>Proofs</td>
<td>67</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Proofs and programs</td>
<td>68</td>
</tr>
<tr>
<td>1.6</td>
<td>Extended example: graph connectivity</td>
<td>68</td>
</tr>
<tr>
<td>1.6.1</td>
<td>Mathematical induction</td>
<td>70</td>
</tr>
<tr>
<td>1.6.2</td>
<td>Proving the theorem by induction</td>
<td>71</td>
</tr>
<tr>
<td>1.6.3</td>
<td>Writing down the proof</td>
<td>73</td>
</tr>
<tr>
<td>1.7</td>
<td>Proof writing style</td>
<td>76</td>
</tr>
<tr>
<td>1.7.1</td>
<td>Patterns in proofs</td>
<td>76</td>
</tr>
<tr>
<td>1.8</td>
<td>Non-standard notation</td>
<td>79</td>
</tr>
<tr>
<td>1.9</td>
<td>Exercises</td>
<td>81</td>
</tr>
<tr>
<td>1.10</td>
<td>Bibliographical notes</td>
<td>83</td>
</tr>
<tr>
<td>2</td>
<td>Computation and Representation</td>
<td>85</td>
</tr>
<tr>
<td>2.1</td>
<td>Examples of binary representations</td>
<td>87</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Representing natural numbers</td>
<td>87</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Representing (potentially negative) integers</td>
<td>89</td>
</tr>
<tr>
<td>2.1.3</td>
<td>Representing rational numbers</td>
<td>90</td>
</tr>
<tr>
<td>2.2</td>
<td>Representing real numbers</td>
<td>92</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Can we represent reals exactly?</td>
<td>92</td>
</tr>
<tr>
<td>2.3</td>
<td>Beyond numbers</td>
<td>97</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Finite representations</td>
<td>97</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Prefix-free encoding</td>
<td>98</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Making representations prefix-free</td>
<td>99</td>
</tr>
<tr>
<td>2.3.4</td>
<td>“Proof by Python” (optional)</td>
<td>100</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Representing letters and text</td>
<td>103</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Representing vectors, matrices, images</td>
<td>106</td>
</tr>
<tr>
<td>2.3.7</td>
<td>Representing graphs</td>
<td>106</td>
</tr>
<tr>
<td>2.3.8</td>
<td>Representing lists</td>
<td>107</td>
</tr>
<tr>
<td>2.3.9</td>
<td>Notation</td>
<td>107</td>
</tr>
<tr>
<td>2.4</td>
<td>Defining computational tasks</td>
<td>108</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Distinguish functions from programs</td>
<td>110</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Advanced note: beyond computing functions</td>
<td>111</td>
</tr>
<tr>
<td>2.5</td>
<td>Exercises</td>
<td>113</td>
</tr>
<tr>
<td>2.6</td>
<td>Bibliographical notes</td>
<td>115</td>
</tr>
<tr>
<td>2.7</td>
<td>Further explorations</td>
<td>116</td>
</tr>
</tbody>
</table>

### I Finite computation

#### 3 Defining computation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Defining computation</td>
<td>121</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Boolean formulas with AND, OR, and NOT</td>
<td>123</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The NAND function</td>
<td>127</td>
</tr>
<tr>
<td>3.2</td>
<td>Informally defining “basic operations” and “algorithms”</td>
<td>129</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Efficient matching of regular expressions (advanced, optional)</td>
<td>301</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Equivalence of DFA's and regular expressions (optional)</td>
<td>305</td>
</tr>
<tr>
<td>9.3</td>
<td>Limitations of regular expressions</td>
<td>306</td>
</tr>
<tr>
<td>9.4</td>
<td>Other semantic properties of regular expressions</td>
<td>311</td>
</tr>
<tr>
<td>9.5</td>
<td>Context free grammars</td>
<td>312</td>
</tr>
<tr>
<td>9.5.1</td>
<td>Context-free grammars as a computational model</td>
<td>315</td>
</tr>
<tr>
<td>9.5.2</td>
<td>The power of context free grammars</td>
<td>316</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Limitations of context-free grammars (optional)</td>
<td>318</td>
</tr>
<tr>
<td>9.6</td>
<td>Semantic properties of context free languages</td>
<td>319</td>
</tr>
<tr>
<td>9.6.1</td>
<td>Uncomputability of context-free grammar equivalence (optional)</td>
<td>320</td>
</tr>
<tr>
<td>9.7</td>
<td>Summary of semantic properties for regular expressions and context-free grammars</td>
<td>323</td>
</tr>
<tr>
<td>9.8</td>
<td>Exercises</td>
<td>324</td>
</tr>
<tr>
<td>9.9</td>
<td>Bibliographical notes</td>
<td>324</td>
</tr>
<tr>
<td>9.10</td>
<td>Further explorations</td>
<td>324</td>
</tr>
<tr>
<td>9.11</td>
<td>Acknowledgements</td>
<td>324</td>
</tr>
<tr>
<td>10</td>
<td>Is every theorem provable?</td>
<td>325</td>
</tr>
<tr>
<td>10.1</td>
<td>Hilbert’s Program and Gödel’s Incompleteness Theorem</td>
<td>326</td>
</tr>
<tr>
<td>10.2</td>
<td>Quantified integer statements</td>
<td>330</td>
</tr>
<tr>
<td>10.3</td>
<td>Diophantine equations and the MRDP Theorem</td>
<td>332</td>
</tr>
<tr>
<td>10.4</td>
<td>Hardness of quantified integer statements</td>
<td>333</td>
</tr>
<tr>
<td>10.4.1</td>
<td>Step 1: Quantified mixed statements and computation histories</td>
<td>334</td>
</tr>
<tr>
<td>10.4.2</td>
<td>Step 2: Reducing mixed statements to integer statements</td>
<td>337</td>
</tr>
<tr>
<td>10.5</td>
<td>Exercises</td>
<td>339</td>
</tr>
<tr>
<td>10.6</td>
<td>Bibliographical notes</td>
<td>340</td>
</tr>
<tr>
<td>10.7</td>
<td>Further explorations</td>
<td>340</td>
</tr>
<tr>
<td>10.8</td>
<td>Acknowledgements</td>
<td>340</td>
</tr>
<tr>
<td>III</td>
<td>Efficient algorithms</td>
<td>341</td>
</tr>
<tr>
<td>11</td>
<td>Efficient computation</td>
<td>343</td>
</tr>
<tr>
<td>11.1</td>
<td>Problems on graphs</td>
<td>344</td>
</tr>
<tr>
<td>11.1.1</td>
<td>Finding the shortest path in a graph</td>
<td>345</td>
</tr>
<tr>
<td>11.1.2</td>
<td>Finding the longest path in a graph</td>
<td>347</td>
</tr>
<tr>
<td>11.1.3</td>
<td>Finding the minimum cut in a graph</td>
<td>348</td>
</tr>
<tr>
<td>11.1.4</td>
<td>Finding the maximum cut in a graph</td>
<td>352</td>
</tr>
<tr>
<td>11.1.5</td>
<td>A note on convexity</td>
<td>352</td>
</tr>
<tr>
<td>11.2</td>
<td>Beyond graphs</td>
<td>354</td>
</tr>
</tbody>
</table>
Preface

“We make ourselves no promises, but we cherish the hope that the unobstructed pursuit of useless knowledge will prove to have consequences in the future as in the past”… “An institution which sets free successive generations of human souls is amply justified whether or not this graduate or that makes a so-called useful contribution to human knowledge. A poem, a symphony, a painting, a mathematical truth, a new scientific fact, all bear in themselves all the justification that universities, colleges, and institutes of research need or require”, Abraham Flexner, The Usefulness of Useless Knowledge, 1939.

“I suggest that you take the hardest courses that you can, because you learn the most when you challenge yourself… CS 121 I found pretty hard.”, Mark Zuckerberg, 2005.

This is a textbook for an undergraduate introductory course on Theoretical Computer Science. The educational goals of this course are to convey the following:

- That computation but arises in a variety of natural and manmade systems, and not only in modern silicon-based computers.

- Similarly, beyond being an extremely important tool, computation also serves as a useful lens to describe natural, physical, mathematical and even social concepts.

- The notion of universality of many different computational models, and the related notion of the duality between code and data.

- The idea that one can precisely define a mathematical model of computation, and then use that to prove (or sometimes only conjecture) lower bounds and impossibility results.
• Some of the surprising results and discoveries in modern theoretical computer science, including the prevalence of NP completeness, the power of interaction, the power of randomness on one hand and the possibility of derandomization on the other, the ability to use hardness “for good” in cryptography, and the fascinating possibility of quantum computing.

I hope that following this course, students would be able to recognize computation, with both its power and pitfalls, as it arises in various settings, including seemingly “static” content or “restricted” formalisms such as macros and scripts. They should be able to follow through the logic of proofs about computation, including the pervasive notion of a reduction and understanding the subtle but crucial “self referential” proofs (such as proofs involving programs that use their own code as input). Students should understand the concept that some problems are intractable, and have the ability to recognize the potential for intractability when they are faced with a new problem. While this course only touches on cryptography, students should understand the basic idea of how computational hardness can be utilized for cryptographic purposes. But more than any specific skill, this course aims to introduce students to a new way of thinking of computation as an object in its own right, and illustrate how this new way of thinking leads to far reaching insights and applications.

My aim in writing this text is to try to convey these concepts in the simplest possible way and try to make sure that the formal notation and model help elucidate, rather than obscure, the main ideas. I also tried to take advantage of modern students’ familiarity (or at least interest!) in programming, and hence use (highly simplified) programming languages as the main model of computation, as opposed to automata or Turing machines. That said, this course does not really assume fluency with any particular programming language, but more a familiarity with the general notion of programming. We will use programming metaphors and idioms, occasionally mentioning concrete languages such as Python, C, or Lisp, but students should be able to follow these descriptions even if they are not familiar with these languages.

Proofs in this course, including the existence of a universal Turing Machine, the fact that every finite function can be computed by some circuit, the Cook-Levin theorem, and many others, are often constructive and algorithmic, in the sense that they ultimately involve transforming one program to another. While the code of these transformations (like any code) is not always easy to read, and the ideas behind the proofs can be grasped without seeing it, I do think that having access to the code, and the ability to play around with it
and see how it acts on various programs, can make these theorems more concrete for the students. To that end, an accompanying website (which is still work in progress) allows executing programs in the various computational models we define, as well as see constructive proofs of some of the theorems.

0.1 TO THE STUDENT

This course can be fairly challenging, mainly because it brings together a variety of ideas and techniques in the study of computation. There are quite a few technical hurdles to master, whether it is following the diagonalization argument in proving the Halting Problem is undecidable, combinatorial gadgets in NP-completeness reductions, analyzing probabilistic algorithms, or arguing about the adversary to prove security of cryptographic primitives.

The best way to engage with the material is to read these notes actively. While reading, I encourage you to stop and think about the following:

• When I state a theorem, stop and try to think of how you would prove it yourself before reading the proof in the notes. You will be amazed by how much you can understand a proof better even after only 5 minutes of attempting it yourself.

• When reading a definition, make sure that you understand what the definition means, and what are natural examples of objects that satisfy it and objects that don’t. Try to think of the motivation behind the definition, and whether there are other natural ways to formalize the same concept.

• Actively notice which questions arise in your mind as you read the text, and whether or not they are answered in the text.

This book contains some code snippets, but this is by no means a programming course. You don’t need to know how to program to follow this material. The reason we use code is that it is a precise way to describe computation. Particular implementation details are not as important to us, and so we will emphasize code readability at the expense of considerations such as error handling, encapsulation, etc. that can be extremely important for real-world programming.

0.1.1 Is the effort worth it?

This is not an easy course, so why should you spend the effort taking it? A traditional justification is that you might encounter these concepts in your career. Perhaps you will come across a hard problem and realize it is NP complete, or find a need to use what you learned about regular expressions. This might very well be true, but the main
benefit of this course is not in teaching you any practical tool or technique, but rather in giving you a different way of thinking: an ability to recognize computation even when it might not be obvious that it occurs, a way to model computational tasks and questions, and to reason about them.

But, regardless of any use you will derive from it, I believe this course is important because it teaches concepts that are both beautiful and fundamental. The role that energy and matter played in the 20th century is played in the 21st by computation and information, not just as tools for our technology and economy, but also as the basic building blocks we use to understand the world. This course will give you a taste of some of the theory behind those, and hopefully spark your curiosity to study more.

0.2 TO POTENTIAL INSTRUCTORS

This book was initially written for my course at Harvard, but I hope that other lecturers will find it useful as well. To some extent, it is similar in content to “Theory of Computation” or “Great Ideas” courses such as those taught at CMU or MIT. There are however some differences, with the most significant being that I do not start with finite automata as the basic computational model, but rather with Boolean circuits, or equivalently straight-line programs. In fact, after briefly discussing general Boolean circuits and the AND, OR and NOT gates, our concrete model for non uniform computation is an extremely simple programming language whose only operation is assigning to one variable the NAND of two others.

Automata are discussed later in the course, after we see Turing machines and undecidability, as an example for a restricted computational model where problems such as halting are effectively solvable. This actually corresponds to the historical ordering: Boolean algebra goes back to Boole’s work in the 1850’s, Turing machines and undecidability were of course discovered in the 1930’s, while finite automata were introduced in the 1943 work of McCulloch and Pitts but only really understood in the seminal 1959 work of Rabin and Scott.

More importantly, the main practical application for restricted models such as regular and context free languages (whether it is for parsing, for analyzing liveness and safety, or even for software defined routing tables) are precisely due to the fact that these are tractable models in which semantic questions can be effectively answered. This practical motivation can be better appreciated after students see the undecidability of semantic properties of general computing models.

Moreover, the Boolean circuit / straightline programs model is extremely simple to both describe and analyze, and some of the main lessons of the theory of computation, including the notions of the du-
ality between code and data, and the idea of universality, can already be seen in this context.

The fact that we started with circuits makes proving the Cook Levin Theorem much easier. In fact, transforming a NAND++ program to an instance of CIRCUIT SAT can be (and is) done in a handful of lines of Python, and combining this with the standard reductions (which are also implemented in Python) allows students to appreciate visually how a question about computation can be mapped into a question about (for example) the existence of an independent set in a graph.

Some more minor differences are the following:

• I introduce uniform computation by extending the above straight-line programming language to include loops and arrays. (I call the resulting programming language “NAND++.”) However, in the same chapter we also define Turing machines and show that these two models are equivalent. In fact, we spend some time showing equivalence between different models (including the λ calculus and RAM machines) to drive home the point that the particular model does not matter.

• For measuring time complexity, we use the standard RAM machine model used (implicitly) in algorithms courses, rather than Turing machines. While these are of course polynomially equivalent, this choice makes the distinction between notions such as \( O(n) \) or \( O(n^2) \) time more meaningful, and ensures the time complexity classes correspond to the informal definitions of linear and quadratic time that students encounter in their algorithms lectures (or their whiteboard coding interviews..).

• A more minor notational difference is that rather than talking about languages (i.e., subsets \( L \subseteq \{0, 1\}^* \)), we talk about Boolean functions (i.e., functions \( f : \{0, 1\}^* \rightarrow \{0, 1\} \)). These are of course equivalent, but the function notation extends more naturally to more general computational tasks. Using functions means we have to be extra vigilant about students distinguishing between the specification of a computational task (e.g., the function) and its implementation (e.g., the program). On the other hand, this point is so important that it is worth repeatedly emphasizing and drilling into the students, regardless of the notation used.

Reducing the time dedicated to automata and context free languages allows instructors to spend more time on topics that I believe that a modern course in the theory of computing needs to touch upon, including randomness and computation, the interactions between proofs and programs (including Gödel’s incompleteness theorem, inter-
active proof systems, and even a bit on the $\lambda$-calculus and the Curry-Howard correspondence), cryptography, and quantum computing.

My intention was to write this text in a level of detail that will enable its use for self-study, and in particular for students to be able to read the text before the corresponding lecture. Toward that end, every chapter starts with a list of learning objectives, ends with a recap, and is peppered with “pause boxes” which encourage students to stop and work out an argument or make sure they understand a definition before continuing further.

Section 0.4 contains a “roadmap” for this book, with descriptions of the different chapters, as well as the dependency structure between them. This can help in planning a course based on this book.

0.3 ACKNOWLEDGEMENTS

This text is constantly evolving, and I am getting input from many people, for which I am deeply grateful. Thanks to Scott Aaronson, Michele Amoretti, Marguerite Basta, Sam Benkelman, Jaroslaw Blass, Emily Chan, Christy Cheng, Michelle Chiang, Daniel Chiu, Chi-Ning Chou, Michael Colavita, Robert Darley Waddilove, Juan Esteller, Leor Fishman, William Fu, Piotr Galuszka, Mark Goldstein, Chan Kang, Nina Katz-Christy, Estefania Lahera, Allison Lee, Ondřej Lengál, Raymond Lin, Emma Ling, Alex Lombardi, Lisa Lu, Aditya Mahadevan, Jacob Meyerson, George Moe, Hamish Nicholson, Sandip Nirmel, Sebastian Oberhoff, Thomas Orton, Pablo Parrilo, Juan Perdomo, Aaron Sachs, Abdelrhman Saleh, Brian Sapozhnikov, Peter Schäfer, Josh Seides, Alaisha Sharma, Noah Singer, Matthew Smedberg, Hikari Sorensen, Alec Sun, Everett Sussman, Marika Swanberg, Garrett Tanzer, Sarah Turnill, Salil Vadhan, Patrick Watts, Ryan Williams, Licheng Xu, Wanqian Yang, Elizabeth Yeoh-Wang, Josh Zelinsky, and Jessica Zhu for helpful feedback.

I will keep adding names here as I get more comments. If you have any comments or suggestions, please do post them on the GitHub repository https://github.com/boazbk/tcs.

Salil Vadhan co-taught with me the first iteration of this course, and gave me a tremendous amount of useful feedback and insights during this process. Michele Amoretti and Marika Swanberg read carefully several chapters of this text and gave extremely helpful detailed comments.

Thanks to Anil Ada, Venkat Guruswami, and Ryan O’Donnell for helpful tips from their experience in teaching CMU 15-251. Juan Esteller and Gabe Montague originally implemented the NAND* languages and the nandpl.org website in OCaml and Javascript.

Thanks to David Steurer for writing the scripts (originally written for our joint notes on the sum of squares algorithm) that I am using.
to produce these notes. David’s scripts are themselves based on several other packages, including pandoc, LaTeX, and the Tufte LaTeX package. I used the [Atom editor] to write these notes, and used the hydorgen package, which relies on the Jupyter project, to write code snippets.

Finally, I’d like to thank my family: my wife Ravit, and my children Alma and Goren. Working on this book (and the corresponding course) took much of my time, to the point that Alma wrote in an essay in her fifth grade class that “universities should not pressure professors to work too much”, and all I have to show for it is about 500 pages of ultra boring mathematical text.
PRELIMINARIES
0

Introduction

“Computer Science is no more about computers than astronomy is about telescopes”, attributed to Edsger Dijkstra. ¹

“Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry.”, Paul Graham 2003. ²

“The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? ... I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block.”, Alan Cobham, 1964

The origin of much of science and medicine can be traced back to the ancient Babylonians. But perhaps their greatest contribution to humanity was the invention of the place-value number system. This is the idea that we can represent any number using a fixed number of digits, whereby the position of the digit is used to determine the corresponding value, as opposed to system such as Roman numerals, where every symbol has a fixed numerical value regardless of position. For example, the distance to the moon is 238,900 of our miles or 259,956 Roman miles. The latter quantity, expressed in standard Roman numerals is

¹ This quote is typically read as disparaging the importance of actual physical computers in Computer Science, but note that telescopes are absolutely essential to astronomy as they provide us with the means to connect theoretical predictions with actual experimental observations.

² To be fair, in the following sentence Graham says “you need to know how to calculate time and space complexity and about Turing completeness”. Apparently, NP-hardness, randomization, cryptography, and quantum computing are not essential to a hacker’s education.
Writing the distance to the sun in Roman numerals would require about 100,000 symbols: a 50 page book just containing this single number!

This means that for someone who thinks of numbers in an additive system like Roman numerals, quantities like the distance to the moon or sun are not merely large- they are unspeakable: cannot be expressed or even grasped. It’s no wonder that Eratosthenes, who was the first person to calculate the earth’s diameter (up to about ten percent error) and Hipparchus who was the first to calculate the distance to the moon, did not use a Roman-numeral type system but rather the Babylonian sexadecimal (i.e., base 60) place-value system.

The Babylonians also invented the precursors of the “standard algorithms” that we were all taught in elementary school for adding and multiplying numbers. These algorithms and their variants have been of course essential to people throughout history working with abaci, papyrus, or pencil and paper, but in our computer age, do they really serve any purpose beyond torturing third graders?

To answer this question, let us try to see in what sense is the standard digit by digit multiplication algorithm “better” than the straightforward implementation of multiplication as iterated addition. Let’s start by more formally describing both algorithms:

**Naive multiplication algorithm:**

**Input:** Non-negative integers $x, y$

**Operation:**
1. Let $result \leftarrow 0$.
2. For $i = 1, \ldots, y$: set $result \leftarrow result + x$
3. Output $result$

**Standard grade-school multiplication algorithm:**

**Input:** Non-negative integers $x, y$

**Operation:**
1. Let $n$ be number of digits of $y$, and set $result \leftarrow 0$.
2. For $i = 0, \ldots, n-1$: set $result \leftarrow result + 10^i \times y_i \times x$,
   where $y_i$ is the $i$-th digit of $y$ (i.e. $y = 10^0 y_0 + 10^1 y_1 + \cdots + y_{n-1} 10^{n-1}$)
3. Output $result$

Both algorithms assume that we already know how to add numbers, and the second one also assumes that we can multiply a number

---

3 For more on the actual algorithms the Babylonians used, see Knuth’s paper and Neugebauer’s classic book.
This is a common size in several programming languages; for example the `long` data type in the Java programming language, and (depending on architecture) the `long` or `long long` types in C.

by a power of 10 (which is after all a simple shift) as well as multiply by a single-digit (which like addition, is done by multiplying each digit and propagating carries). Now suppose that \( x \) and \( y \) are two numbers of \( n \) decimal digits each. Adding two such numbers takes at least \( n \) single-digit additions (depending on how many times we need to use a “carry”), and so adding \( x \) to itself \( y \) times will take at least \( n \cdot y \) single-digit additions. In contrast, the standard grade-school algorithm reduces this problem to taking \( n \) products of \( x \) with a single-digit (which require up to \( 2n \) single-digit operations each, depending on carries) and then adding all of those together (total of \( n \) additions, which, again depending on carries, would cost at most \( 2n^2 \) single-digit operations) for a total of at most \( 4n^2 \) single-digit operations. How much faster would \( 4n^2 \) operations be than \( n \cdot y \)? And would this make any difference in a modern computer?

Let us consider the case of multiplying 64-bit or 20-digit numbers.\(^4\) That is, the task of multiplying two numbers \( x, y \) that are between \( 10^{19} \) and \( 10^{20} \). Since in this case \( n = 20 \), the standard algorithm would use at most \( 4n^2 = 1600 \) single-digit operations, while repeated addition would require at least \( n \cdot y \geq 20 \cdot 10^{19} \) single-digit operations. To understand the difference, consider that a human being might do a single-digit operation in about 2 seconds, requiring just under an hour to complete the calculation of \( x \times y \) using the grade-school algorithm. In contrast, even though it is more than a billion times faster, a modern PC that computes \( x \times y \) using naïve iterated addition would require about \( 10^{20}/10^9 = 10^{11} \) seconds (which is more than three millennia!) to compute the same result.

It is important to distinguish between the value of a number, and the length of its representation (i.e., the number of digits it has). There is a big difference between the two: having 1,000,000,000 dollars is not the same as having 10 dollars! When talking about running time of algorithms, “less is more”, and so an algorithm that runs in time proportional to the number of digits of an input number (or even the number of digit squared) is much preferred to an algorithm that runs in time proportional to the value of the input number.

We see that computers have not made algorithms obsolete. On the contrary, the vast increase in our ability to measure, store, and communicate data has led to a much higher demand for developing better and more sophisticated algorithms that can allow us to make better decisions based on these data. We also see that to a large extent the notion of algorithm is independent of the actual computing device that will execute it. The digit-by-digit multiplication algorithm is
vastly better than iterated addition, regardless whether the technology we use to implement it is a silicon based chip, or a third grader with pen and paper.

Theoretical computer science is concerned with the inherent properties of algorithms and computation; namely, those properties that are independent of current technology. We ask some questions that were already pondered by the Babylonians, such as “what is the best way to multiply two numbers?”, but also questions that rely on cutting-edge science such as “could we use the effects of quantum entanglement to factor numbers faster?”.

In Computer Science parlance, a scheme such as the decimal (or sexadecimal) positional representation for numbers is known as a data structure, while the operations on this representations are known as algorithms. Data structures and algorithms have enabled amazing applications, but their importance goes beyond their practical utility. Structures from computer science, such as bits, strings, graphs, and even the notion of a program itself, as well as concepts such as universality and replication, have not just found (many) practical uses but contributed a new language and a new way to view the world.

0.1 EXTENDED EXAMPLE: A FASTER WAY TO MULTIPLY

Once you think of the standard digit-by-digit multiplication algorithm, it seems like obviously the “right” way to multiply numbers. Indeed, in 1960, the famous mathematician Andrey Kolmogorov organized a seminar at Moscow State University in which he conjectured that every algorithm for multiplying two \( \omega \) digit numbers would require a number of basic operations that is proportional to \( \omega^2 \).

An other way to say it, is that he conjectured that in any multiplication algorithm, doubling the number of digits would quadruple the number of basic operations required.

A young student named Anatoly Karatsuba was in the audience, and within a week he found an algorithm that requires only about \( Cn^1 \) operations for some constant \( C \). Such a number becomes much smaller than \( n^2 \) as \( n \) grows. Amazingly, Karatsuba’s algorithm is based on a faster way to multiply two-digit numbers.

Suppose that \( x, y \in [100] = \{0, \ldots, 99\} \) are a pair of two-digit numbers. Let’s write \( x = 10\pi + \pi \) for the “tens” digit of \( x \), and \( \pi \) for the “ones” digit, so that \( x = 10\pi + \pi \) and write similarly \( y = 10\eta + y \) for \( \eta, y \in [10] \).

The grade-school algorithm for multiplying \( x \) and \( y \) is illustrated in Fig. 1.

The grade-school algorithm works by transforming the task of multiplying a pair of two-digit number into four single-digit multipl-
Figure 1: The grade-school multiplication algorithm illustrated for multiplying \(x = 10\bar{x} + \bar{z}\) and \(y = 10\bar{y} + \bar{u}\). It uses the formula \((10\bar{x} + \bar{z}) \times (10\bar{y} + \bar{u}) = 100\bar{x}\bar{y} + 10(\bar{x}\bar{y} + \bar{z}\bar{u}) + \bar{z}\bar{u}\).

cations via the formula

\[
(10\bar{x} + \bar{z}) \times (10\bar{y} + \bar{u}) = 100\bar{x}\bar{y} + 10(\bar{x}\bar{y} + \bar{z}\bar{u}) + \bar{z}\bar{u} \tag{1}
\]

Karatsuba’s algorithm is based on the observation that we can express this also as

\[
(10\bar{x} + \bar{z}) \times (10\bar{y} + \bar{u}) = (100 - 10)\bar{x}\bar{y} + 10[(\bar{x} + \bar{z})(\bar{y} + \bar{u})] - (10 - 1)\bar{x}\bar{u} \tag{2}
\]

which reduces multiplying the two-digit number \(x\) and \(y\) to computing the following three “simple” products: \(\bar{x}\bar{y}\), \(\bar{x}\bar{u}\) and \((\bar{x} + \bar{z})(\bar{y} + \bar{u})\).

Of course if all we wanted to was to multiply two digit numbers, we wouldn’t really need any clever algorithms. It turns out that we can repeatedly apply the same idea, and use them to multiply 4-digit numbers, 8-digit numbers, 16-digit numbers, and so on and so forth. If we used the grade-school approach then our cost for doubling the number of digits would be to quadruple the number of multiplications, which for \(n = 2^\ell\) digits would result in about \(4^\ell = n^2\) operations. In contrast, in Karatsuba’s approach doubling the number of digits only triples the number of operations, which means that for \(n = 2^\ell\) digits we require about \(3^\ell = n\log_2 3 \sim n^{1.58}\) operations.

Specifically, we use a recursive strategy as follows:

\(\text{The term } (\bar{x} + \bar{z})(\bar{y} + \bar{u}) \text{ is not exactly a single-digit multiplication as } \bar{x} + \bar{z} \text{ and } \bar{y} + \bar{u} \text{ are numbers between } 0 \text{ and } 18 \text{ and not between } 0 \text{ and } 9 \text{. As we'll see, it turns out that this does not make much of a difference, since when we use this algorithm recursively on } n \text{-digit numbers, this term will have at most } \lceil n/2 \rceil + 1 \text{ digits, which is essentially half the number of digits as the original input.}\)
Figure 2: Karatsuba’s multiplication algorithm illustrated for multiplying $x = 10^m \overline{x}$ and $y = 10^n \overline{y}$. We compute the three orange, green and purple products $\overline{x} \overline{y}$, $\overline{x} \overline{y}$ and $(\overline{x} + \overline{x})(\overline{y} + \overline{y})$ and then add and subtract them to obtain the result.

**Karatsuba Multiplication:**

**Input:** nonnegative integers $x, y$ each of at most $n$ digits

**Operation:**

1. If $n \leq 2$ then return $x \cdot y$ (using a constant number of single-digit multiplications)
2. Otherwise, let $m = \lfloor n/2 \rfloor$, and write $x = 10^m \overline{x}$ and $y = 10^n \overline{y}$. \(^8\)
   
   2. Use recursion to compute $A = \overline{xy}$, $B = \overline{x} \overline{y}$ and $C = (\overline{x} + \overline{x})(\overline{y} + \overline{y})$. Note that all the numbers will have at most $m + 1$ digits.
3. Return $(10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C$

To understand why the output will be correct, first note that for $n > 2$, it will always hold that $m < n - 1$, and hence the recursive calls will always be for multiplying numbers with a smaller number of digits, and (since eventually we will get to single or double digit numbers) the algorithm will indeed terminate. Now, since $x = 10^m \overline{x}$ and $y = 10^n \overline{y}$

\[
x \times y = 10^n \overline{xy} + 10^m (\overline{xy} + \overline{x} \overline{y}) + x \overline{y}.
\]  

\(^8\)Recall that for a number $x$, $\lfloor x \rfloor$ is obtained by “rounding down” $x$ to the largest integer smaller or equal to $x$.  

Rearranging the terms we see that

\[ x \times y = 10^n \overline{x} \cdot \overline{y} + 10^m \left[ (\overline{x} + \overline{y})(\overline{y} + y) - \overline{x}y - \overline{y}x \right] + xy, \quad (4) \]

which equals \((10^n - 10^m) \cdot A + 10^m \cdot B + (1 - 10^m) \cdot C\), the value returned by the algorithm.

The key observation is that Eq. (4) reduces the task of computing the product of two \(n\)-digit numbers to computing three products of \([n/2]\)-digit numbers. Specifically, we can compute \(x \times y\) from the three products \(xy\), \(xy\) and \((\overline{x} + \overline{y})(\overline{y} + y)\), using a constant number (in fact eight) of additions, subtractions, and multiplications by \(10^n\) or \(10^{[n/2]}\).

(Multiplication by a power of ten can be done very efficiently as it corresponds to simply shifting the digits.) Intuitively this means that as the number of digits doubles, the cost of performing a multiplication via Karatsuba’s algorithm triples instead of quadrupling, as happens in the naive algorithm. This implies that multiplying numbers of \(n = 2^\ell\) digits costs about \(3^\ell = n^{\log_2 3} \sim n^{1.585}\) operations. In an Exercise 0.3, you will formally show that the number of single-digit operations that Karatsuba’s algorithm uses for multiplying \(n\) digit integers is at most \(O(n^{\log_2 3})\) (see also Fig. 2).

**Figure 3:** Running time of Karatsuba’s algorithm vs. the grade-school algorithm. (Python implementation available online.) Note the existence of a “cutoff” length, where for sufficiently large inputs Karatsuba becomes more efficient than the grade-school algorithm. The precise cutoff location varies by implementation and platform details, but will always occur eventually.

**Ceilings, floors, and rounding** One of the benefits of using Big-\(O\) notation is that we can allow ourselves to be a little looser with issues such as rounding numbers etc.. For example, the natural way to describe Karatsuba’s algorithm’s running time is...
Karatsuba’s algorithm reduces an \( n \)-bit multiplication to three \( n/2 \)-bit multiplications, which in turn are reduced to nine \( n/4 \)-bit multiplications and so on. We can represent the computational cost of all these multiplications in a 3-ary tree of depth \( \log_2 n \), where at the root the extra cost is \( cn \) operations, at the first level the extra cost is \( c(n/2) \) operations, and at each of the \( 3^i \) nodes of level \( i \), the extra cost is \( c(n/2^i) \). The total cost is

\[
T(n) = cn \sum_{i=0}^{\log_2 n} (3/2)^i \leq 10cn \log_2 3
\]

by the formula for summing a geometric series.

via the following recursive equation

\[
T(n) = 3T(n/2) + O(n)
\]

but of course if \( n \) is not even then we cannot recursively invoke the algorithm on \( n/2 \)-digit integers. Rather, the true recursion is

\[
T(n) = 3T([n/2] + 1) + O(n).
\]

However, this will not make much difference when we don’t worry about constant factors, since it’s not hard to show that \( T(n + O(1)) \leq T(n) + o(T(n)) \) for the functions we care about. Another way to show that this doesn’t hurt us is to note that for every number \( n \), we can find a number \( n' \leq 2n \), such that \( n' \) is a power of two. Thus we can always “pad” the input by adding some input bits to make sure the number of digits is a power of two, in which case we will never run into these rounding issues. These kind of tricks work not just in the context of multiplication algorithms but in many other cases as well. Thus most of the time we can safely ignore these kinds of “rounding issues”.

0.1.1 Beyond Karatsuba’s algorithm

It turns out that the ideas of Karatsuba can be further extended to yield asymptotically faster multiplication algorithms, as was shown by Toom and Cook in the 1960s. But this was not the end of the line. In 1971, Schönhage and Strassen gave an even faster algorithm using the Fast Fourier Transform; their idea was to somehow treat integers as “signals” and do the multiplication more efficiently by moving to the
The Fourier transform is a central tool in mathematics and engineering, used in a great number of applications. If you have not seen it yet, you will hopefully encounter it at some point in your studies.

The latest asymptotic improvement was given by Fürer in 2007 (though it only starts beating the Schönhage-Strassen algorithm for truly astronomical numbers). And yet, despite all this progress, we still don’t know whether or not there is an $O(\omega)$ time algorithm for multiplying two $n$ digit numbers!

Matrix Multiplication (advanced note) (We will have several such “advanced” or “optional” notes and sections throughout this book. These may assume background that not every student has, and can be safely skipped over as none of the future parts will depend on them.)

It turns out that a similar idea as Karatsuba’s can be used to speed up matrix multiplications as well. Matrices are a powerful way to represent linear equations and operations, widely used in a great many applications of scientific computing, graphics, machine learning, and many many more.

One of the basic operations one can do with two matrices is to multiply them. For example, if $\mathbf{x} = \begin{pmatrix} x_{0,0} & x_{0,1} \\ x_{1,0} & x_{1,1} \end{pmatrix}$ and $\mathbf{y} = \begin{pmatrix} y_{0,0} & y_{0,1} \\ y_{1,0} & y_{1,1} \end{pmatrix}$ then the product of $\mathbf{x}$ and $\mathbf{y}$ is the matrix

$\mathbf{x}\mathbf{y} = \begin{pmatrix} x_{0,0}y_{0,0} + x_{0,1}y_{1,0} & x_{0,0}y_{0,1} + x_{0,1}y_{1,1} \\ x_{1,0}y_{0,0} + x_{1,1}y_{1,0} & x_{1,0}y_{0,1} + x_{1,1}y_{1,1} \end{pmatrix}$. You can see that we can compute this matrix by eight products of numbers.

Now suppose that $n$ is even and $\mathbf{x}$ and $\mathbf{y}$ are a pair of $n \times n$ matrices which we can think of as each composed of four $(n/2) \times (n/2)$ blocks $x_{0,0}, x_{0,1}, x_{1,0}, x_{1,1}$ and $y_{0,0}, y_{0,1}, y_{1,0}, y_{1,1}$. Then the formula for the matrix product of $\mathbf{x}$ and $\mathbf{y}$ can be expressed in the same way as above, just replacing products $x_{a,b}y_{c,d}$ with matrix products, and addition with matrix addition. This means that we can use the formula above to give an algorithm that doubles the dimension of the matrices at the expense of increasing the number of operation by a factor of 8, which for $n = 2^\ell$ will result in $8^\ell = n^3$ operations.

In 1969 Volker Strassen noted that we can compute the product of a pair of two-by-two matrices using only seven products of numbers by observing that each entry of the matrix $\mathbf{xy}$ can be computed by adding and subtracting the following seven terms:

$t_1 = (x_{0,0} + x_{1,1})(y_{0,0} + y_{1,1})$,  
$t_2 = (x_{0,0} + x_{1,1})y_{0,0}$,  
$t_3 = x_{0,0}y_{0,1} - y_{1,1}$,  
$t_4 = x_{1,1}(y_{0,1} - y_{0,0})$,  
$t_5 = (x_{0,0} + x_{0,1})y_{1,0}$,  
$t_6 = (x_{1,0} - x_{0,0})(y_{0,0} + y_{0,1})$,  
$t_7 = (x_{0,1} - x_{1,1})(y_{1,0} + y_{1,1})$. Indeed, one can verify that $\mathbf{xy} = \begin{pmatrix} t_1 + t_2 - t_5 + t_7 & t_3 + t_5 \\ t_2 + t_4 & t_1 + t_3 - t_2 + t_6 \end{pmatrix}$.

Using this observation, we can obtain an algorithm...
such that doubling the dimension of the matrices results in increasing the number of operations by a factor of $7$, which means that for $n = 2^\ell$ the cost is $7^\ell = n^{\log_2 7} \sim n^{2.807}$. A long sequence of work has since improved this algorithm, and the current record has running time about $O(n^{2.373})$. However, unlike the case of integer multiplication, at the moment we don’t know of any algorithm for matrix multiplication that runs in time linear or even close to linear in the size of the input matrices (e.g., an $O(n^2 \text{polylog}(n))$ time algorithm). People have tried to use group representations, which can be thought of as generalizations of the Fourier transform, to obtain faster algorithms, but this effort has not yet succeeded.

0.2 ALGORITHMS BEYOND ARITHMETIC

The quest for better algorithms is by no means restricted to arithmetical tasks such as adding, multiplying or solving equations. Many graph algorithms, including algorithms for finding paths, matchings, spanning trees, cuts, and flows, have been discovered in the last several decades, and this is still an intensive area of research. (For example, the last few years saw many advances in algorithms for the maximum flow problem, borne out of surprising connections with electrical circuits and linear equation solvers.) These algorithms are being used not just for the “natural” applications of routing network traffic or GPS-based navigation, but also for applications as varied as drug discovery through searching for structures in gene-interaction graphs to computing risks from correlations in financial investments.

Google was founded based on the PageRank algorithm, which is an efficient algorithm to approximate the “principal eigenvector” of (a dampened version of) the adjacency matrix of web graph. The Akamai company was founded based on a new data structure, known as consistent hashing, for a hash table where buckets are stored at different servers. The backpropagation algorithm, which computes partial derivatives of a neural network in $O(n)$ instead of $O(n^2)$ time, underlies many of the recent phenomenal successes of learning deep neural networks. Algorithms for solving linear equations under sparsity constraints, a concept known as compressed sensing, have been used to drastically reduce the amount and quality of data needed to analyze MRI images. This is absolutely crucial for MRI imaging of cancer tumors in children, where previously doctors needed to use anesthesia to suspend breath during the MRI exam, sometimes with dire consequences.

Even for classical questions, studied through the ages, new dis-
coveries are still being made. For example, for the question of determining whether a given integer is prime or composite, which has been studied since the days of Pythagoras, efficient probabilistic algorithms were only discovered in the 1970s, while the first deterministic polynomial-time algorithm was only found in 2002. For the related problem of actually finding the factors of a composite number, new algorithms were found in the 1980s, and (as we’ll see later in this course) discoveries in the 1990s raised the tantalizing prospect of obtaining faster algorithms through the use of quantum mechanical effects.

Despite all this progress, there are still many more questions than answers in the world of algorithms. For almost all natural problems, we do not know whether the current algorithm is the “best”, or whether a significantly better one is still waiting to be discovered. As we already saw, even for the classical problem of multiplying numbers we have not yet answered the age-old question of “is multiplication harder than addition?”.

But at least we now know the right way to ask it.

0.3 ON THE IMPORTANCE OF NEGATIVE RESULTS.

Finding better multiplication algorithms is undoubtedly a worthwhile endeavor. But why is it important to prove that such algorithms don’t exist? What useful applications could possibly arise from an impossibility result?

One motivation is pure intellectual curiosity. After all, this is a question even Archimedes could have been excited about. Another reason to study impossibility results is that they correspond to the fundamental limits of our world. In other words, they are laws of nature. In physics, the impossibility of building a perpetual motion machine corresponds to the law of conservation of energy. The impossibility of building a heat engine beating Carnot’s bound corresponds to the second law of thermodynamics, while the impossibility of faster-than-light information transmission is a cornerstone of special relativity.

In mathematics, while we all learned the solution for quadratic equations in high school, the impossibility of generalizing this to equations of degree five or more gave birth to group theory. Another example of an impossibility result comes from geometry. For two millennia, mathematicians tried to show that Euclid’s fifth axiom or “postulate” could be derived from the first four. (This fifth postulate was known as the “parallel postulate”, and roughly speaking it states that every line has a unique parallel line of each distance.) It was shown to be impossible using constructions of so called “non-Euclidean geometries”, which turn out to be crucial for the theory of general relativity.
Indeed, some exciting recent research is focused on trying to use computational complexity to shed light on fundamental questions in physics such as understanding black holes and reconciling general relativity with quantum mechanics.

Lecture Recap

• The history of algorithms goes back thousands of years; they have been essential much of human progress and these days form the basis of multi-billion dollar industries, as well as life-saving technologies.

• There is often more than one algorithm to achieve the same computational task. Finding a faster algorithm can often make a much bigger difference than improving computing hardware.

• Better algorithms and data structures don’t just speed up calculations, but can yield new qualitative insights.

• One question we will study is to find out what is the most efficient algorithm for a given problem.

• To show that an algorithm is the most efficient one for a given problem, we need to be able to prove that it is impossible to solve the problem using a smaller amount of computational resources.

0.4 ROADMAP TO THE REST OF THIS COURSE

Often, when we try to solve a computational problem, whether it is solving a system of linear equations, finding the top eigenvector of a matrix, or trying to rank Internet search results, it is enough to use the “I know it when I see it” standard for describing algorithms. As long as we find some way to solve the problem, we are happy and don’t care so much about formal descriptions of the algorithm. But when
we want to answer a question such as “does there exist an algorithm to solve the problem $P$?” we need to be much more precise.

In particular, we will need to (1) define exactly what it means to solve $P$, and (2) define exactly what an algorithm is. Even (1) can sometimes be non-trivial but (2) is particularly challenging; it is not at all clear how (and even whether) we can encompass all potential ways to design algorithms. We will consider several simple models of computation, and argue that, despite their simplicity, they do capture all “reasonable” approaches to achieve computing, including all those that are currently used in modern computing devices.

Once we have these formal models of computation, we can try to obtain impossibility results for computational tasks, showing that some problems can not be solved (or perhaps can not be solved within the resources of our universe). Archimedes once said that given a fulcrum and a long enough lever, he could move the world. We will see how reductions allow us to leverage one hardness result into a slew of a great many others, illuminating the boundaries between the computable and uncomputable (or tractable and intractable) problems.

Later in this course we will go back to examining our models of computation, and see how resources such as randomness or quantum entanglement could potentially change the power of our model. In the context of probabilistic algorithms, we will see a glimpse of how randomness has become an indispensable tool for understanding computation, information, and communication. We will also see how computational difficulty can be an asset rather than a hindrance, and be used for the “derandomization” of probabilistic algorithms. The same ideas also show up in cryptography, which has undergone not just a technological but also an intellectual revolution in the last few decades, much of it building on the foundations that we explore in this course.

Theoretical Computer Science is a vast topic, branching out and touching upon many scientific and engineering disciplines. This course only provides a very partial (and biased) sample of this area. More than anything, I hope I will manage to “infect” you with at least some of my love for this field, which is inspired and enriched by the connection to practice, but which I find to be deep and beautiful regardless of applications.

0.4.1 Dependencies between chapters

This book is divided into the following parts:

- **Preliminaries**: Introduction, mathematical background, and representing objects as strings.

- **Part I: Finite computation**: Boolean circuits / straightline pro-
grams. Universal gatesets, counting lower bound, representing programs as string and universality.

- **Part II: Uniform computation:** Turing machines / programs with loops. Equivalence of models (including RAM machines and $\lambda$ calculus), universality, uncomputability, Gödel’s incompleteness theorem, restricted models (regular and context free languages).

- **Part III: Efficient computation:** Definition of running time, time hierarchy theorem, $P$ and $NP$, $NP$ completeness, space bounded computation.

- **Part IV: Randomized computation:** Probability, randomized algorithms, $BPP$, amplification, $BPP \subseteq P^{poly}$, pseudorandom generators and derandomization.

- **Part V: Advanced topics:** Cryptography, proofs and algorithms (interactive and zero knowledge proofs, Curry-Howard correspondence), quantum computing.

The book proceeds in linear order, with each chapter building on the previous one, with the following exceptions:

- All chapters in **Part V** (Advanced topics) are independent of one another, and you can choose which one of them to read.

- **Chapter 10** (Gödel’s incompleteness theorem), **Chapter 9** (Restricted computational models), and **Chapter 16** (Space bounded computation), are not used in following chapters. Hence you can choose to skip them.

A course based on this book can use all of Parts I, II, and III (possibly skipping over some or all of Chapter 10, Chapter 9 or Chapter 16), and then either cover all or some of Part IV, and add a “sprinkling” of advanced topics from Part V based on student or instructor interest.

### 0.5 EXERCISES

**Disclaimer** Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.
Exercise 0.1 Rank the significance of the following inventions in speeding up multiplication of large (that is 100-digit or more) numbers. That is, use “back of the envelope” estimates to order them in terms of the speedup factor they offered over the previous state of affairs.

1. Discovery of the grade-school digit by digit algorithm (improving upon repeated addition)
2. Discovery of Karatsuba’s algorithm (improving upon the digit by digit algorithm)
3. Invention of modern electronic computers (improving upon calculations with pen and paper).

Exercise 0.2 The 1977 Apple II personal computer had a processor speed of 1.023 Mhz or about $10^6$ operations per seconds. At the time of this writing the world’s fastest supercomputer performs 93 “petaflops” ($10^{15}$ floating point operations per second) or about $10^{18}$ basic steps per second. For each one of the following running times (as a function of the input length $n$), compute for both computers how large an input they could handle in a week of computation, if they run an algorithm that has this running time:

1. $n$ operations.
2. $n^2$ operations.
3. $n \log n$ operations.
4. $2^n$ operations.
5. $n!$ operations.

Exercise 0.3 — Analysis of Karatsuba’s Algorithm. 1. Suppose that $T_1, T_2, T_3, \ldots$ is a sequence of numbers such that $T_2 \leq 10$ and for every $n, T_n \leq 3T_{\lfloor n/2 \rfloor} + Cn$ for some $C \geq 1$. Prove that $T_n \leq 10C n \log_2 3$ for every $n > 2.11$

2. Prove that the number of single-digit operations that Karatsuba’s algorithm takes to multiply two $n$ digit numbers is at most $1000n \log_2 3$.

Exercise 0.4 Implement in the programming language of your choice functions Gradeschool_multiply(x, y) and Karatsuba_multiply(x, y) that take two arrays of digits x and y and
Hint: Start by showing this for the case that $A = f$ for some natural number $f$, in which case you can do so recursively by breaking the matrices into $f \times f$ blocks.

Exercise 0.5 — Matrix Multiplication (optional, advanced). In this exercise, we show that if for some $\omega > 2$, we can write the product of two $k \times k$ real-valued matrices $A, B$ using at most $k^\omega$ multiplications, then we can multiply two $n \times n$ matrices in roughly $n^\omega$ time for every large enough $n$.

To make this precise, we need to make some notation that is unfortunately somewhat cumbersome. Assume that there is some $k \in \mathbb{N}$ and $m \leq k^\omega$ such that for every $k \times k$ matrices $A, B, C$ such that $C = AB$, we can write for every $i, j \in [k]$:

$$C_{i,j} = \sum_{\ell=0}^{m} \alpha_{i,j}^\ell f_\ell(A)g_\ell(B)$$

for some linear functions $f_0, \ldots, f_{m-1}, g_0, \ldots, g_{m-1} : \mathbb{R}^n \to \mathbb{R}$ and coefficients $\{\alpha_{i,j}^\ell\}_{i, j \in [k], \ell \in [m]}$. Prove that under this assumption for every $\epsilon > 0$, if $n$ is sufficiently large, then there is an algorithm that computes the product of two $n \times n$ matrices using at most $O(n^{\omega+\epsilon})$ arithmetic operations.12

0.6 BIBLIOGRAPHICAL NOTES

For an overview of what we’ll see in this course, you could do far worse than read Bernard Chazelle’s wonderful essay on the Algorithm as an Idiom of modern science.

0.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

- The Fourier transform, the Fast Fourier transform algorithm and how to use it multiply polynomials and integers. This lecture of Jeff Erickson (taken from his collection of notes) is a very good starting point. See also this MIT lecture and this popular article.

- Fast matrix multiplication algorithms, and the approach of obtaining exponent two via group representations.

- The proofs of some of the classical impossibility results in mathematics we mentioned, including the impossibility of proving Euclid’s fifth postulate from the other four, impossibility of trisecting an angle with a straightedge and compass and the impossibility
of solving a quintic equation via radicals. A geometric proof of
the impossibility of angle trisection (one of the three geometric
problems of antiquity, going back to the ancient greeks) is given in
this blog post of Tao. This book of Mario Livio covers some of the
background and ideas behind these impossibility results.
In this chapter, we review some of the mathematical concepts that we will use in this course. Most of these are not very complicated, but do require some practice and exercise to get comfortable with. If you have not previously encountered some of these concepts, there are several excellent freely-available resources online that cover them. In particular, the CS 121 webpage contains a program for self study of all the needed notions using the lecture notes, videos, and assignments of MIT course 6.042j Mathematics for Computer science. (The MIT lecture notes were also used in the past in Harvard CS 20.)

1.1 A MATHEMATICIAN’S APOLOGY

Before explaining the math background, perhaps I should explain why is this course so “mathematically heavy”. After all, this is supposed to be a course about computation; one might think we should be talking mostly about programs, rather than more “mathematical” objects such as sets, functions, and graphs, and doing more coding on an actual computer than writing mathematical proofs with pen and
Actually, scientists currently do not know the answer to this question, but we will see that settling it in either direction has very interesting applications touching on areas as far apart as Internet security and quantum mechanics.

At the end of the day, mathematics is simply a language for modeling concepts in a precise and unambiguous way. In this course, we will be mostly interested in the concept of computation. For example, we will look at questions such as “is there an efficient algorithm to find the prime factors of a given integer?”.\(^1\) To even phrase such a question, we need to give a precise definition of the notion of an algorithm, and of what it means for an algorithm to be efficient. Also, if the answer to this or similar questions turns out to be negative, then this cannot be shown by simply writing and executing some code. After all, there is no empirical experiment that will prove the nonexistence of an algorithm. Thus, our only way to show this type of negative result is to use mathematical proofs. So you can see why our main tools in this course will be mathematical proofs and definitions.

---

\(^1\) Actually, scientists currently do not know the answer to this question, but we will see that settling it in either direction has very interesting applications touching on areas as far apart as Internet security and quantum mechanics.

---

**1.2 A QUICK OVERVIEW OF MATHEMATICAL PREREQUISITES**

The main notions we will use in this course are the following:

- **Proofs**: First and foremost, this course will involve a heavy dose...
of formal mathematical reasoning, which includes mathematical
\textit{definitions, statements,} and \textit{proofs.}

- **Sets:** The basic set \textit{relationships} of membership ($\in$), containment ($\subseteq$), and set \textit{operations}, principally union, intersection, set difference and Cartesian product ($\cup, \cap, \setminus$ and $\times$).

- **Tuples and strings:** The set $\Sigma^k$ of length-$k$ strings/lists over elements in $\Sigma$, where $\Sigma$ is some finite set which is called the \textit{alphabet} (quite often $\Sigma = \{0,1\}$). We use $\Sigma^*$ for the set of all strings of finite length.

- **Some special sets:** The set $\mathbb{N}$ of natural numbers. We will index from zero in this course and so write $\mathbb{N} = \{0,1,2,\ldots\}$. We will use $[n]$ for the set $\{0,1,2,\ldots,n-1\}$. We use $\{0,1\}^*$ for the set of all binary strings and $\{0,1\}^n$ for the set of strings of length $n$. If $x$ is a string of length $n$, then we refer to its coordinate by $x_0, x_1, \ldots, x_{n-1}$.

- **Functions:** The \textit{domain} and \textit{codomain} of a function, properties such as being \textit{one-to-one} (also known as \textit{injective}) or \textit{onto} (also known as \textit{surjective}) functions, as well as \textit{partial functions} (that, unlike standard or "total" functions, are not necessarily defined on all elements of their domain).

- **Logical operations:** The operations AND, OR, and NOT ($\land, \lor, \neg$) and the quantifiers “there exists” and “for all” ($\exists, \forall$).

- **Basic combinatorics:** Notions such as $\binom{n}{k}$ (the number of $k$-sized subsets of a set of size $n$).

- **Graphs:** Undirected and directed graphs, connectivity, paths, and cycles.

- **Big-O notation:** $O, o, \Omega, \omega, \Theta$ notation for analyzing asymptotic growth of functions.

- **Discrete probability:** Later on in we will use \textit{probability theory}, and specifically probability over \textit{finite} samples spaces such as tossing $n$ coins, including notions such as \textit{random variables, expectation,} and \textit{concentration.} We will only use probability theory in the second half of this text, and will review it beforehand. However, probabilistic reasoning is a subtle (and extremely useful!) skill, and it’s always good to start early in acquiring it.

In the rest of this chapter we briefly review the above notions. This is partially to remind the reader and reinforce material that might not be fresh in your mind, and partially to introduce our notation and conventions which might occasionally differ from those you’ve encountered before.
1.3 READING MATHEMATICAL TEXTS

In this course, we will eventually tackle some fairly complex definitions. For example, let us consider one of the definitions that we will encounter towards the very end of this text:

**Definition 1.1 — The complexity class BQP.** If $G : \{0,1\}^n \rightarrow \{0,1\}$ is a finite function and $Q$ is a Quantum circuit then we say that $Q$ computes $G$ if for every $x \in \{0,1\}^n$, $\Pr[Q(x) = G(x)] \geq 2/3$.

The class BQP (which stands for "bounded-error quantum polynomial time") is the set of all functions $F : \{0,1\}^* \rightarrow \{0,1\}$ such that there exists a polynomial-time Turing Machine $M$ that satisfies the following: for every $n \in \mathbb{N}$, $M(1^n)$ is a Quantum circuit $Q_n$ that computes $F_n$, where $F_n : \{0,1\}^n \rightarrow \{0,1\}$ is the restriction of $F$ to inputs of length $n$. That is, $\Pr[M(1^n)(x) = F(x)] \geq 2/3$ for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$.

We will also see the following theorem:

**Theorem 1.2 — Shor’s Algorithm.** Let $F : \{0,1\}^* \rightarrow \{0,1\}$ be the function that on input a string representation of a pair $(m,i)$ of natural numbers, outputs 1 if and only if the $i$-th bit of the smallest prime factor of $m$ is equal to 1. Then $F \in \text{BQP}$. While it should make sense to you by the end of the term, at the current point in time it is perfectly fine if Definition 1.1 and Theorem 1.2 seem to you as a meaningless combination of inscrutable terms. Indeed, to a large extent they are such a combination, as they contain many terms that we have not defined (and that we would need to build on a semester’s worth of material to be able to define). Yet, even when faced with what seems like completely incomprehensible gibberish, it is still possible for us to try to make some sense of it, and try to at least to be able to "know what we don’t know". Let’s use Definition 1.1 and Theorem 1.2 as examples. For starters, let me tell you what this definition and this theorem are about. Quantum computing is an approach to use the peculiarities of quantum mechanics to build computing devices that can solve certain problems exponentially faster than current computers. Many large companies and governments are extremely excited about this possibility, and are investing hundreds of millions of dollars in trying to make this happen. To a first order of approximation, the reason they are so excited is Shor’s Algorithm (i.e., Theorem 1.2), which says that the problem of integer factoring, with history going back thousands of years, and whose difficulty is (as we’ll see) closely tied to the security of many
current encryption schemes, can be solved efficiently using quantum computers.

Theorem 1.2 was proven by Peter Shor in 1994. However, Shor could not even have stated this theorem, let alone prove it, without having the proper definition (i.e., Definition 1.1) in place. Definition 1.1 defines the class BQP of functions that can be computed in polynomial time by quantum computers. Like any mathematical definition, it defines a new concept (in this case the class BQP) in terms of other concepts. In this case the concepts that are needed are

- The notion of a function, which is a mapping of one set to another. In this particular case we use functions whose output is a single number that is either zero or one (i.e., a bit) and the input is a list of bits (i.e., a string) which can either have a fixed length $n$ (this is denoted as the set $\{0, 1\}^n$) or have length that is not a priori bounded (this is denoted by $\{0, 1\}^\ast$).

- Restrictions of functions. If $F$ is a function that takes strings of arbitrary length as input (i.e., members of the set $\{0, 1\}^\ast$) then $F_n$ is the restriction of $F$ to inputs of length $n$ (i.e., members of $\{0, 1\}^n$).

- We use the notion of a Quantum circuit which will be our computational model for quantum computers, and which we will encounter later on in the course. Quantum circuits can compute functions with a fixed input length $n$, and we define the notion of computing a function $G$ as outputting on input $x$ the value $G(x)$ with probability at least $2/3$.

- We will also use the notion of Turing machines which will be our computational model for “classical” computers.\footnote{As we’ll see, there is a great variety of ways to model “classical computers”, including RAM machines, $\lambda$-calculus, and NAND++ programs.}

- We require that for every $n \in \mathbb{N}$, the quantum circuit $Q_n$ for $F_n$ can be generated efficiently, in the sense that there is a polynomial-time classical program $P$ that on input a string of $n$ ones (which we shorthand as $1^n$) outputs $Q_n$.

The point of this example is not for you to understand Definition 1.1 and Theorem 1.2. Fully understanding them will require background that will take us weeks to develop. The point is to show that you should not be afraid of even the most complicated looking definitions and mathematical terminology. No matter how convoluted the notation, and how many layers of indirection, you can always look at mathematical definitions and try to at least attempt at answering the following questions:

1. What is the intuitive notion that this definition aims at modeling?
Dealing with mathematical text is in many ways not so different from dealing with any other complex text, whether it’s a legal argument, a philosophical treatise, an English Renaissance play, or even the source code of an operating system. You should not expect it to be clear in a first reading, but you need not despair. Rather you should engage with the text, trying to figure out both the high level intentions as well as the underlying details. Luckily, compared to philosophers or even programmers, mathematicians have a greater discipline of introducing definitions in linear order, making sure that every new concept is defined only in terms of previously defined notions. As you read through the rest of this chapter and this text, try to ask yourself questions 1-3 above every time that you encounter a new definition.

### 1.3.1 Example: Defining a one to one function

Here is a simpler mathematical definition, which you may have encountered in the past (and will encounter again shortly):

#### Definition 1.3 — One to one function. A function \( f : S \to T \) is one to one if for every two elements \( x, x' \in S \), if \( x \neq x' \) then \( f(x) \neq f(x') \).

This definition captures a simple concept, but even so it uses quite a bit of notation. When reading this definition, or any other piece of mathematical text, it is often useful to annotate it with a pen as you’re going through it, as in Fig. 1.1. For every identifier you encounter (for example \( f, S, T, x, x' \) in this case), make sure that you realize what sort of object is it: is it a set, a function, an element, a number, a gremlin? Make sure you understand how the identifiers are quantified. For example, in Definition 1.3 there is a universal or “for all” (sometimes denotes by \( \forall \)) quantifier over pairs \((x, x')\) of distinct elements in \( S \). Finally, and most importantly, make sure that aside from being able to parse the definition formally, you also have an intuitive understanding of what is it that the text is actually saying. For example, Definition 1.3 says that a one to one function is a function where every output is obtained by a unique input.

Reading mathematical texts in this way takes time, but it gets easier with practice. Moreover, this is one of the most transferable skills you could take from this course. Our world is changing rapidly, not just in the realm of technology, but also in many other human endeavors,
whether it is medicine, economics, law or even culture. Whatever your future aspirations, it is likely that you will encounter texts that use new concepts that you have not seen before (for semi-random recent examples from current “hot areas”, see Fig. 1.2 and Fig. 1.3). Being able to internalize and then apply new definitions can be hugely important. It is a skill that’s much easier to acquire in the relatively safe and stable context of a mathematical course, where one at least has the guarantee that the concepts are fully specified, and you have access to your teaching staff for questions.

Figure 1.2: A snippet from the “methods” section of the “AlphaGo Zero” paper by Silver et al, Nature, 2017.
1.4 BASIC DISCRETE MATH OBJECTS

We now quickly review some of the mathematical objects (the “basic data structures” of mathematics, if you will) we use in this course.

1.4.1 Sets

A set is an unordered collection of objects. For example, when we write \( S = \{2, 4, 7\} \), we mean that \( S \) denotes the set that contains the numbers 2, 4, and 7. (We use the notation “\( 2 \in S \)” to denote that 2 is an element of \( S \).) Note that the set \( \{2, 4, 7\} \) and \( \{7, 4, 2\} \) are identical, since they contain the same elements. Also, a set either contains an element or does not contain it – there is no notion of containing it “twice” – and so we could even write the same set \( S \) as \( \{2, 2, 4, 7\} \) (though that would be a little weird). The cardinality of a finite set \( S \), denoted by \( |S| \), is the number of elements it contains.\(^3\) So, in the example above, \( |S| = 3 \). A set \( S \) is a subset of a set \( T \), denoted by \( S \subseteq T \), if every element of \( S \) is also an element of \( T \). (We can also describe this by saying that \( T \) is a superset of \( S \).) For example, \( \{2, 7\} \subseteq \{2, 4, 7\} \).

The set that contains no elements is known as the empty set and it is denoted by \( \emptyset \).

We can define sets by either listing all their elements or by writing down a rule that they satisfy such as

\[
\text{EVEN} = \{x : x = 2y \text{ for some non-negative integer } y\} .
\] (1.1)

Of course there is more than one way to write the same set, and often we will use intuitive notation listing a few examples that illustrate the rule. For example, we can also define EVEN as

\[
\text{EVEN} = \{0, 2, 4, \ldots\} .
\] (1.2)

\(^3\) Later in this course we will discuss how to extend the notion of cardinality to infinite sets.
Note that a set can be either finite (such as the set \{2, 4, 7\}) or infinite (such as the set EVEN). Also, the elements of a set don’t have to be numbers. We can talk about the sets such as the set \{a, e, i, o, u\} of all the vowels in the English language, or the set \{New York, Los Angeles, Chicago, Houston, Philadelphia, Phoenix, San Antonio, San Diego, Dallas\} of all cities in the U.S. with population more than one million per the 2010 census. A set can even have other sets as elements, such as the set \{\emptyset, \{1, 2\}, \{2, 3\}, \{1, 3\}\} of all even-sized subsets of \{1, 2, 3\}.

**Operations on sets:** The union of two sets \(S, T\), denoted by \(S \cup T\), is the set that contains all elements that are either in \(S\) or in \(T\). The intersection of \(S\) and \(T\), denoted by \(S \cap T\), is the set of elements that are both in \(S\) and in \(T\). The set difference of \(S\) and \(T\), denoted by \(S \setminus T\) (and in some texts also by \(S - T\)), is the set of elements that are in \(S\) but not in \(T\).

**Tuples, lists, strings, sequences:** A tuple is an ordered collection of items. For example \((1, 5, 2, 1)\) is a tuple with four elements (also known as a 4-tuple or quadruple). Since order matters, this is not the same tuple as the 4-tuple \((1, 1, 5, 2)\) or the 3-tuple \((1, 5, 2)\). A 2-tuple is also known as a pair. We use the terms tuples and lists interchangeably. A tuple where every element comes from some finite set \(\Sigma\) (such as \{0, 1\}) is also known as a string. Analogously to sets, we denote the length of a tuple \(T\) by \(|T|\). Just like sets, we can also think of infinite analogues of tuples, such as the ordered collection \((1, 2, 4, 9, \ldots)\) of all perfect squares. Infinite ordered collections are known as sequences; we might sometimes use the term “infinite sequence” to emphasize this, and use “finite sequence” as a synonym for a tuple.\(^4\)

**Cartesian product:** If \(S\) and \(T\) are sets, then their Cartesian product, denoted by \(S \times T\), is the set of all ordered pairs \((s, t)\) where \(s\) is in \(S\) and \(t\) is in \(T\). For example, if \(S = \{1, 2, 3\}\) and \(T = \{10, 12\}\), then \(S \times T\) contains the 6 elements \((1, 10), (2, 10), (3, 10), (1, 12), (2, 12), (3, 12)\). Similarly if \(S, T, U\) are sets then \(S \times T \times U\) is the set of all ordered triples \((s, t, u)\) where \(s\) is in \(S\), \(t\) is in \(T\), and \(u\) is in \(U\). More generally, for every positive integer \(n\) and sets \(S_0, \ldots, S_{n-1}\), we denote by \(S_0 \times S_1 \times \cdots \times S_{n-1}\) the set of ordered \(n\)-tuples \((s_0, \ldots, s_{n-1})\) where \(s_i\) is in \(S_i\) for every \(i \in \{0, \ldots, n-1\}\). For every set \(S\), we denote the set \(S \times S\) by \(S^2\), \(S \times S \times S\) by \(S^3\), \(S \times S \times S \times S\) by \(S^4\), and so on and so forth.

### 1.4.2 Sets in Python (optional)

To get more comfortable with sets, one can also play with the set data structure in Python:\(^5\)

```python
A = {7, 10, 12}
B = {12, 8, 5}
print(A==B)
```

\(^4\) We can identify a sequence \((a_0, a_1, a_2, \ldots)\) of elements in some set \(S\) with a function \(A : \mathbb{N} \to S\) (where \(a_n = A(n)\) for every \(n \in \mathbb{N}\)). Similarly, we can identify a \(k\)-tuple \((a_0, \ldots, a_{k-1})\) of elements in \(S\) with a function \(A : \{0, \ldots, k\} \to S\).

\(^5\) The set data structure only corresponds to finite sets; infinite sets are much more cumbersome to handle in programming languages, though mechanisms such as Python generators and lazy evaluation in general can be helpful.
We start our indexing of both $\mathbb{N}$ and $\{0\}$ from 0, while many other texts index those sets from 1. Starting from zero or one is simply a convention that doesn’t make much difference, as long as one is consistent about it.

The letter $\mathbb{Z}$ stands for the German word “Zahlen”, which means numbers.

### 1.4.3 Special sets

There are several sets that we will use in this course time and again, and so find it useful to introduce explicit notation for them. For starters we define

$$\mathbb{N} = \{0, 1, 2, \ldots\}$$ \hspace{1cm} (1.3)

to be the set of all natural numbers, i.e., non-negative integers. For any natural number $n$, we define the set $[n]$ as $\{0, \ldots, n - 1\} = \{k \in \mathbb{N} : k < n\}$.

We will also occasionally use the set $\mathbb{Z} = \{\ldots, -2, -1, 0, +1, +2, \ldots\}$ of (negative and non-negative) integers, as well as the set $\mathbb{R}$ of real numbers. (This is the set that includes not just the integers, but also fractional and even irrational numbers; e.g., $\mathbb{R}$ contains numbers such as $+0.5$, $-\pi$, etc.) We denote by $\mathbb{R}_+$ the set $\{x \in \mathbb{R} : x > 0\}$ of positive real numbers. This set is sometimes also denoted as $(0, \infty)$.

**Strings:** Another set we will use time and again is

$$\{0, 1\}^n = \{(x_0, \ldots, x_{n-1}) : x_0, \ldots, x_{n-1} \in \{0, 1\}\}$$ \hspace{1cm} (1.4)
which is the set of all $n$-length binary strings for some natural number $n$. That is, $\{0, 1\}^n$ is the set of all $n$-tuples of zeroes and ones. This is consistent with our notation above: $\{0, 1\}^2$ is the Cartesian product $\{0, 1\} \times \{0, 1\}$, $\{0, 1\}^3$ is the product $\{0, 1\} \times \{0, 1\} \times \{0, 1\}$ and so on.

We will write the string $(x_0, x_1, \ldots, x_{n-1})$ as simply $x_0 x_1 \cdots x_{n-1}$ and so for example

$$\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\} . \quad (1.5)$$

For every string $x \in \{0, 1\}^n$ and $i \in [n]$, we write $x_i$ for the $i^{th}$ coordinate of $x$. If $x$ and $y$ are strings, then $xy$ denotes their concatenation.

That is, if $x \in \{0, 1\}^n$ and $y \in \{0, 1\}^m$, then $xy$ is equal to the string $z \in \{0, 1\}^{n+m}$ such that for $i \in [n]$, $z_i = x_i$ and for $i \in [n, \ldots, n+m-1]$, $z_i = y_{i-n}$.

We will also often talk about the set of binary strings of all lengths, which is

$$\{0, 1\}^* = \{(x_0, \ldots, x_{n-1}) : n \in \mathbb{N}, x_0, \ldots, x_{n-1} \in \{0, 1\}\} . \quad (1.6)$$

Another way to write this set is as

$$\{0, 1\}^* = \{0, 1\}^0 \cup \{0, 1\}^1 \cup \{0, 1\}^2 \cup \cdots \quad (1.7)$$

or more concisely as

$$\{0, 1\}^* = \bigcup_{n \in \mathbb{N}} \{0, 1\}^n . \quad (1.8)$$

The set $\{0, 1\}^*$ contains also the “string of length 0” or “the empty string”, which we will denote by "".\footnote{We follow programming languages in this notation; other texts sometimes use $\epsilon$ or $\lambda$ to denote the empty string. However, this doesn’t matter much since we will rarely encounter this “edge case”.}

**Generalizing the star operation:** For every set $\Sigma$, we define

$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n . \quad (1.9)$$

For example, if $\Sigma = \{a, b, c, d, \ldots, z\}$ then $\Sigma^*$ denotes the set of all finite length strings over the alphabet a-z.

**Concatenation:** As mentioned in Section 1.4.3, the concatenation of two strings $x \in \Sigma^n$ and $y \in \Sigma^m$ is the $(n+m)$-length string $xy$ obtained by writing $y$ after $x$.

1.4.4 Functions

If $S$ and $T$ are nonempty sets, a function $F$ mapping $S$ to $T$, denoted by $F : S \rightarrow T$, associates with every element $x \in S$ an element $F(x) \in T$. The set $S$ is known as the domain of $F$ and the set $T$ is known as the codomain of $F$. The image of a function $F$ is the set $\{F(x) \mid x \in S\}$ which is the subset of $F$’s codomain consisting of all output elements that are mapped from some input.\footnote{Some texts use range to denote the image of a function, while other texts use codomain to denote the codomain of a function. Hence we will avoid using the term “range” altogether.} Just as with sets, we can write a
function either by listing the table of all the values it gives for elements in $S$ or using a rule. For example if $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $T = \{0, 1\}$, then the table below defines a function $F : S \to T$. Note that this function is the same as the function defined by the rule $F(x) = (x \mod 2)$.\(^{10}\)

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>

\(^{10}\) For two natural numbers $x$ and $a$, $x \mod a$ (where \(\mod\) is shorthand for “modulo”) denotes the remainder of $x$ when it is divided by $a$. That is, it is the number $r$ in $\{0, \ldots, a - 1\}$ such that $x = ak + r$ for some integer $k$. We sometimes also use the notation $x = y(\mod a)$ to denote the assertion that $x \mod a$ is the same as $y \mod a$.

If $F : S \to T$ satisfies that $F(x) \neq F(y)$ for all $x \neq y$ then we say that $F$ is one-to-one (also known as an injective function or simply an injection). If $F$ satisfies that for every $y \in T$ there is some $x \in S$ such that $F(x) = y$ then we say that $F$ is onto (also known as a surjective function or simply a surjection). A function that is both one-to-one and onto is known as a bijective function or simply a bijection. A bijection from a set $S$ to itself is also known as a permutation of $S$. If $F : S \to T$ is a bijection then for every $y \in T$ there is a unique $x \in S$ s.t. $F(x) = y$. We denote this value $x$ by $F^{-1}(y)$. Note that $F^{-1}$ is itself a bijection from $T$ to $S$ (can you see why?).

Giving a bijection between two sets is often a good way to show they have the same size. In fact, the standard mathematical definition of the notion that “$S$ and $T$ have the same cardinality” is that there exists a bijection $f : S \to T$. In particular, the cardinality of a set $S$ is defined to be $n$ if there is a bijection from $S$ to the set $\{0, \ldots, n - 1\}$. As we will see later in this course, this is a definition that can generalizes to defining the cardinality of infinite sets.

**Partial functions:** We will sometimes be interested in partial functions from $S$ to $T$. A partial function is allowed to be undefined on some subset of $S$. That is, if $F$ is a partial function from $S$ to $T$, then for every $s \in S$, either there is (as in the case of standard functions) an element $F(s)$ in $T$, or $F(s)$ is undefined. For example, the partial function $F(x) = \sqrt{x}$ is only defined on non-negative real numbers. When we want to distinguish between partial functions and standard (i.e., non-partial) functions, we will call the latter total functions. When we say “function” without any qualifier then we mean a total function.
The notion of partial functions is a strict generalization of functions, and so every function is a partial function, but not every partial function is a function. (That is, for every nonempty $S$ and $T$, the set of partial functions from $S$ to $T$ is a proper superset of the set of total functions from $S$ to $T$.) When we want to emphasize that a function $f$ from $A$ to $B$ might not be total, we will write $f : A \to_p B$. We can think of a partial function $F$ from $S$ to $T$ also as a total function from $S$ to $T \cup \{ \perp \}$ where $\perp$ is some special “failure symbol”, and so instead of saying that $F$ is undefined at $x$, we can say that $F(x) = \perp$.

**Basic facts about functions:** Verifying that you can prove the following results is an excellent way to brush up on functions:

- If $F : S \to T$ and $G : T \to U$ are one-to-one functions, then their composition $H : S \to U$ defined as $H(s) = G(F(s))$ is also one to one.

- If $F : S \to T$ is one to one, then there exists an onto function $G : T \to S$ such that $G(F(s)) = s$ for every $s \in S$.

- If $G : T \to S$ is onto then there exists a one-to-one function $F : S \to T$ such that $G(F(s)) = s$ for every $s \in S$.

- If $S$ and $T$ are finite sets then the following conditions are equivalent to one another: (a) $|S| \leq |T|$, (b) there is a one-to-one function $F : S \to T$, and (c) there is an onto function $G : T \to S$.\(^{11}\)

\(^{11}\)This is actually true even for infinite $S$ and $T$: in that case (b) is the commonly accepted definition for $|S| \leq |T|$. 

---

**Figure 1.4:** We can represent finite functions as a directed graph where we put an edge from $x$ to $f(x)$. The *onto* condition corresponds to requiring that every vertex in the codomain of the function has in-degree at least one. The *one-to-one* condition corresponds to requiring that every vertex in the codomain of the function has in-degree at most one. In the examples above $F$ is an onto function, $G$ is one to one, and $H$ is neither onto nor one to one.
strongly suggest you try to prove them on your own, or at least convince yourself that they are true by proving special cases of those for small sizes (e.g., $|S| = 3, |T| = 4, |U| = 5$).

Let us prove one of these facts as an example:

**Lemma 1.4** If $S, T$ are non-empty sets and $F : S \rightarrow T$ is one to one, then there exists an onto function $G : T \rightarrow S$ such that $G(F(s)) = s$ for every $s \in S$.

**Proof.** Let $S, T$ and $F : S \rightarrow T$ be as in the Lemma’s statement, and choose some $s_0 \in S$. We will define the function $G : T \rightarrow S$ as follows: for every $t \in T$, if there is some $s \in S$ such that $F(s) = t$ then set $G(t) = s$ (the choice of $s$ is well defined since by the one-to-one property of $F$, there cannot be two distinct $s, s'$ that both map to $t$). Otherwise, set $G(t) = s_0$. Now for every $s \in S$, by the definition of $G$, if $t = F(s)$ then $G(t) = G(F(s)) = s$. Moreover, this also shows that $G$ is onto, since it means that for every $s \in S$ there is some $t$ (namely $t = F(s)$) such that $G(t) = s$. ■

1.4.5 Graphs

**Graphs** are ubiquitous in Computer Science, and many other fields as well. They are used to model a variety of data types including social networks, road networks, deep neural nets, gene interactions, correlations between observations, and a great many more. The formal definitions of graphs are below, but if you have not encountered them before then I urge you to read up on them in one of the sources linked above. Graphs come in two basic flavors: **undirected** and **directed**.\[12\]

**Figure 1.5**: An example of an undirected and a directed graph. The undirected graph has vertex set $\{1, 2, 3, 4\}$ and edge set $\{\{1, 2\}, \{2, 3\}, \{2, 4\}\}$. The directed graph has vertex set $\{a, b, c\}$ and the edge set $\{(a, b), (b, c), (c, a), (a, c)\}$.

**Definition 1.5 — Undirected graphs.** An undirected graph $G = (V, E)$ consists of a set $V$ of **vertices** and a set $E$ of edges. Every edge is a \[12\] It is possible, and sometimes useful, to think of an undirected graph as simply a directed graph with the special property that for every pair $u, v$ either both the edges $uv$ and $vu$ are present or neither of them is. However, in many settings there is a significant difference between undirected and directed graphs, and so it’s typically best to think of them as separate categories.
size two subset of \( V \). We say that two vertices \( u, v \in V \) are neighbors, denoted by \( u \sim v \), if the edge \( \{u, v\} \) is in \( E \).

Given this definition, we can define several other properties of graphs and their vertices. We define the degree of \( u \) to be the number of neighbors \( u \) has. A path in the graph is a tuple \( (u_0, \ldots, u_k) \in V^k \), for some \( k > 0 \) such that \( u_{i+1} \) is a neighbor of \( u_i \) for every \( i \in [k] \). A simple path is a path \( (u_0, \ldots, u_k) \) where all the \( u_i \)'s are distinct. A cycle is a path \( (u_0, \ldots, u_k) \) where \( u_0 = u_k \). We say that two vertices \( u, v \in V \) are connected if either \( u = v \) or there is a path from \( (u_0, \ldots, u_k) \) where \( u_0 = u \) and \( u_k = v \). We say that the graph \( G \) is connected if every pair of vertices in it is connected.

Here are some basic facts about undirected graphs. We give some informal arguments below, but leave the full proofs as exercises. (The proofs can also be found in most basic texts on graph theory.)

Lemma 1.6 In any undirected graph \( G = (V, E) \), the sum of the degrees of all vertices is equal to twice the number of edges.

Lemma 1.6 can be shown by seeing that every edge \( \{u, v\} \) contributes twice to the sum of the degrees (once for \( u \) and the second time for \( v \)).

Lemma 1.7 The connectivity relation is transitive, in the sense that if \( u \) is connected to \( v \), and \( v \) is connected to \( w \), then \( u \) is connected to \( w \).

Lemma 1.7 can be shown by simply attaching a path of the form \( (u, u_1, u_2, \ldots, u_{k-1}, v) \) to a path of the form \( (v, u'_1, \ldots, u'_{k'-1}, w) \) to obtain the path \( (u, u_1, \ldots, u_{k-1}, v, u'_1, \ldots, u'_{k'-1}, w) \) that connects \( u \) to \( w \).

Lemma 1.8 For every undirected graph \( G = (V, E) \) and connected pair \( u, v \), the shortest path from \( u \) to \( v \) is simple. In particular, for every connected pair there exists a simple path that connects them.

Lemma 1.8 can be shown by “shortcutting” any non simple path of the form \( (u, u_1, \ldots, u_{i-1}, w, u_{i+1}, \ldots, u_{j-1}, w, u_{j+1}, \ldots, u_{k-1}, v) \) where the same vertex \( w \) appears in both the \( i \)-th and \( j \)-position, to obtain the shorter path \( (u, u_1, \ldots, u_{i-1}, w, u_{j+1}, \ldots, u_{k-1}, v) \).

If you haven’t seen these proofs before, it is indeed a great exercise to transform the above informal exercises into fully rigorous proofs.

**Definition 1.9 — Directed graphs.** A directed graph \( G = (V, E) \) consists of a set \( V \) and a set \( E \subseteq V \times V \) of ordered pairs of \( V \). We denote the edge \( (u, v) \) also as \( u \rightarrow v \). If the edge \( u \rightarrow v \) is present in the graph then we say that \( v \) is an out-neighbor of \( u \) and \( u \) is an in-neighbor of \( v \).
A directed graph might contain both $\overrightarrow{uw}$ and $\overrightarrow{vu}$ in which case $u$ will be both an in-neighbor and an out-neighbor of $v$ and vice versa. The in-degree of $u$ is the number of in-neighbors it has, and the out-degree of $v$ is the number of out-neighbors it has. A path in the graph is a tuple $(u_0, \ldots, u_k) \in V^k$, for some $k > 0$ such that $u_{i+1}$ is an out-neighbor of $u_i$ for every $i \in [k]$. As in the undirected case, a simple path is a path $(u_0, \ldots, u_{k-1})$ where all the $u_i$'s are distinct and a cycle is a path $(u_0, \ldots, u_k)$ where $u_0 = u_k$. One type of directed graphs we often care about is directed acyclic graphs or DAGs, which, as their name implies, are directed graphs without any cycles.

The lemmas we mentioned above have analogs for directed graphs. We again leave the proofs (which are essentially identical to their undirected analogs) as exercises for the reader:

**Lemma 1.10** In any directed graph $G = (V, E)$, the sum of the in-degrees is equal to the sum of the out-degrees, which is equal to the number of edges.

**Lemma 1.11** In any directed graph $G$, if there is a path from $u$ to $v$ and a path from $v$ to $w$, then there is a path from $u$ to $w$.

**Lemma 1.12** For every directed graph $G = (V, E)$ and a pair $u, v$ such that there is a path from $u$ to $v$, the shortest path from $u$ to $v$ is simple.

**Graph terminology** The word graph in the sense above was coined by the mathematician Sylvester in 1878 in analogy with the chemical graphs used to visualize molecules. There is an unfortunate confusion with the more common usage of the term as a way to plot data, and in particular a plot of some function $f(x)$ as a function of $x$. We can merge these two meanings by thinking of a function $f : A \rightarrow B$ as a special case of a directed graph over the vertex set $V = A \cup B$ where we put the edge $xf(x)$ for every $x \in A$. In a graph constructed in this way every vertex in $A$ has out-degree one.

The following lecture of Berkeley CS70 provides an excellent overview of graph theory.

### 1.4.6 Logic operators and quantifiers.

If $P$ and $Q$ are some statements that can be true or false, then $P$ AND $Q$ (denoted as $P \land Q$) is the statement that is true if and only if both $P$ and $Q$ are true, and $P$ OR $Q$ (denoted as $P \lor Q$) is the statement that is true if and only if either $P$ or $Q$ is true. The negation of $P$, denoted as $\neg P$ or $\overline{P}$, is the statement that is true if and only if $P$ is false.

Suppose that $P(x)$ is a statement that depends on some parameter $x$ (also sometimes known as an unbound variable) in the sense that for
every instantiation of $x$ with a value from some set $S$, $P(x)$ is either true or false. For example, $x > 7$ is a statement that is not a priori true or false, but does become true or false whenever we instantiate $x$ with some real number. In such case we denote by $\forall x \in S P(x)$ the statement that is true if and only if $P(x)$ is true for every $x \in S$.\footnote{In these notes we will place the variable that is bound by a quantifier in a subscript and so write $\forall x \in S P(x)$ whereas other texts might use $\forall x \in S, P(x)$.} We denote by $\exists x \in S P(x)$ the statement that is true if and only if there exists some $x \in S$ such that $P(x)$ is true.

For example, the following is a formalization of the true statement that there exists a natural number $n$ larger than 100 that is not divisible by 3:

$$\exists n \in \mathbb{N} (n > 100) \land (\forall k \in \mathbb{N} k + k \neq n) . \quad (1.10)$$

“For sufficiently large $n” One expression which comes up time and again is the claim that some statement $P(n)$ is true “for sufficiently large $n”$. What this means is that there exists an integer $N_0$ such that $P(n)$ is true for every $n > N_0$. We can formalize this as $\exists N_0 \in \mathbb{N} \forall n > N_0 P(n)$.

### 1.4.7 Quantifiers for summations and products

The following shorthands for summing up or taking products of several numbers are often convenient. If $S = \{s_0, \ldots, s_{n-1}\}$ is a finite set and $f : S \to \mathbb{R}$ is a function, then we write $\sum_{x \in S} f(x)$ as shorthand for

$$f(s_0) + f(s_1) + f(s_2) + \ldots + f(s_{n-1}) , \quad (1.11)$$

and $\prod_{x \in S} f(x)$ as shorthand for

$$f(s_0) \cdot f(s_1) \cdot f(s_2) \cdot \ldots \cdot f(s_{n-1}) . \quad (1.12)$$

For example, the sum of the squares of all numbers from 1 to 100 can be written as

$$\sum_{i \in \{1, \ldots, 100\}} i^2 . \quad (1.13)$$

Since summing up over intervals of integers is so common, there is a special notation for it, and for every two integers $a \leq b$, $\sum_{i=a}^{b} f(i)$ denotes $\sum_{i \in S} f(i)$ where $S = \{x \in \mathbb{Z} : a \leq x \leq b\}$. Hence we can write the sum Eq. (1.13) as

$$\sum_{i=1}^{100} i^2 . \quad (1.14)$$

### 1.4.8 Parsing formulas: bound and free variables

In mathematics, as in coding, we often have symbolic “variables” or “parameters”. It is important to be able to understand, given some
formula, whether a given variable is bound or free in this formula. For example, in the following statement $n$ is free but $a$ and $b$ are bound by the $\exists$ quantifier:

$$\exists_{a,b \in \mathbb{N}} (a \neq 1) \land (a \neq n) \land (n = a \times b)$$  \hspace{1cm} (1.15)$$

Since $n$ is free, it can be set to any value, and the truth of the statement Eq. (1.15) depends on the value of $n$. For example, if $n = 8$ then Eq. (1.15) is true, but for $n = 11$ it is false. (Can you see why?)

The same issue appears when parsing code. For example, in the following snippet from the C++ programming language

```cpp
for (int i=0 ; i<n ; i=i+1) {
    printf(" * ");
}
```

the variable $i$ is bound to the `for` operator but the variable $n$ is free.

The main property of bound variables is that we can change them to a different name (as long as it doesn’t conflict with another used variable) without changing the meaning of the statement. Thus for example the statement

$$\exists_{x,y \in \mathbb{N}} (x \neq 1) \land (x \neq n) \land (n = x \times y)$$  \hspace{1cm} (1.16)$$

is equivalent to Eq. (1.15) in the sense that it is true for exactly the same set of $n$’s. Similarly, the code

```cpp
for (int j=0 ; j<n ; j=j+1) {
    printf(" * ");
}
```

produces the same result as the code above that used $i$ instead of $j$.

---

**Aside: mathematical vs programming notation**

Mathematical notation has a lot of similarities with programming language, and for the same reasons. Both are formalisms meant to convey complex concepts in a precise way. However, there are some cultural differences. In programming languages, we often try to use meaningful variable names such as `NumberOfVertices` while in math we often use short identifiers such as $n$. (Part of it might have to do with the tradition of mathematical proofs as being handwritten and verbally presented, as opposed to typed up and compiled.)

One consequence of that is that in mathematics we often end up reusing identifiers, and also “run out” of letters and hence use Greek letters too, as well as
distinguish between small and capital letters. Similarly, mathematical notation tends to use quite a lot of “overloading”, using operators such as + for a great variety of objects (e.g., real numbers, matrices, finite field elements, etc.), and assuming that the meaning can be inferred from the context.

Both fields have a notion of “types”, and in math we often try to reserve certain letters for variables of a particular type. For example, variables such as $i, j, k, \ell, m, n$ will often denote integers, and $\epsilon$ will often denote a small positive real number. When reading or writing mathematical texts, we usually don’t have the advantage of a “compiler” that will check type safety for us. Hence it is important to keep track of the type of each variable, and see that the operations that are performed on it “make sense”.

1.4.9 Asymptotics and Big-$O$ notation

“log log log $n$ has been proved to go to infinity, but has never been observed to do so.”, Anonymous, quoted by Carl Pomerance (2000)

It is often very cumbersome to describe precisely quantities such as running time and is also not needed, since we are typically mostly interested in the “higher order terms”. That is, we want to understand the scaling behavior of the quantity as the input variable grows. For example, as far as running time goes, the difference between an $n^5$-time algorithm and an $n^2$-time one is much more significant than the difference between an $100n^2 + 10n$ time algorithm and an $10n^2$ time algorithm. For this purpose, $O$-notation is extremely useful as a way to “declutter” our text and focus our attention on what really matters. For example, using $O$-notation, we can say that both $100n^2 + 10n$ and $10n^2$ are simply $\Theta(n^2)$ (which informally means “the same up to constant factors”), while $n^2 = o(n^3)$ (which informally means that $n^2$ is “much smaller than” $n^3$).\(^{14}\)

Generally (though still informally), if $F, G$ are two functions mapping natural numbers to non-negative reals, then “$F = O(G)$” means that $F(n) \leq G(n)$ if we don’t care about constant factors, while “$F = o(G)$” means that $F$ is much smaller than $G$, in the sense that no matter by what constant factor we multiply $F$, if we take $n$ to be large enough then $G$ will be bigger (for this reason, sometimes $F = o(G)$ is written as $F \ll G$). We will write $F = \Theta(G)$ if $F = O(G)$ and $G = O(F)$, which one can think of as saying that $F$ is the same as $G$ if we don’t care about constant factors. More formally, we define Big-$O$ notation as follows:

\(^{14}\) While Big-$O$ notation is often used to analyze running time of algorithms, this is by no means the only application. At the end of the day, Big-$O$ notation is just a way to express asymptotic inequalities between functions on integers. It can be used regardless of whether these functions are a measure of running time, memory usage, or any other quantity that may have nothing to do with computation.
Recall that \( \mathbb{R}_+ \), which is also sometimes denoted as \((0, \infty)\), is the set of positive real numbers, so the above is just a way of saying that \( \epsilon \) and \( \delta \)'s outputs are always positive numbers.

**Definition 1.13 — Big-O notation.** For \( F, G : \mathbb{N} \to \mathbb{R}_+ \), we define \( F = O(G) \) if there exist numbers \( a, N_0 \in \mathbb{N} \) such that \( F(n) \leq a \cdot G(n) \) for every \( n > N_0 \). \(^{15}\) We define \( F = \Omega(G) \) if \( G = O(F) \).

We write \( F = o(G) \) if for every \( \epsilon > 0 \) there is some \( N_0 \) such that \( F(n) < \epsilon G(n) \) for every \( n > N_0 \). We write \( F = \omega(G) \) if \( G = o(F) \).

We write \( F = \Theta(G) \) if \( F = O(G) \) and \( G = O(F) \).

We can also use the notion of limits to define Big- and Little-O notation. You can verify that \( F = O(G) \) (or, equivalently, \( G = \omega(F) \)) if and only if \( \lim_{n \to \infty} \frac{F(n)}{G(n)} = 0 \). Similarly, if the limit \( \lim_{n \to \infty} \frac{F(n)}{G(n)} \) exists and is a finite number then \( F = O(G) \). If you are familiar with the notion of supremum, then you can verify that \( F = O(G) \) if and only if \( \limsup_{n \to \infty} \frac{F(n)}{G(n)} < \infty \).

**Figure 1.6.** If \( F(n) = o(G(n)) \) then for sufficiently large \( n \), \( F(n) \) will be smaller than \( G(n) \). For example, if Algorithm \( A \) runs in time \( 1000 \cdot n + 10^6 \) and Algorithm \( B \) runs in time \( 0.01 \cdot n^2 \) then even though \( B \) might be more efficient for smaller inputs, when the inputs get sufficiently large, \( A \) will run much faster than \( B \).

\(^{15}\) Recall that \( \mathbb{R}_+ \), which is also sometimes denoted as \((0, \infty)\), is the set of positive real numbers, so the above is just a way of saying that \( F \) and \( G \)'s outputs are always positive numbers.
It’s often convenient to use “anonymous functions” in the context of $O$-notation, and also to emphasize the input parameter to the function. For example, when we write a statement such as $F(n) = O(n^3)$, we mean that $F = O(G)$ where $G$ is the function defined by $G(n) = n^3$. Chapter 7 in Jim Apsnes’ notes on discrete math provides a good summary of $O$ notation; see also this tutorial for a gentler and more programmer-oriented introduction.

1.4.10 Some “rules of thumb” for Big-$O$ notation
There are some simple heuristics that can help when trying to compare two functions $F$ and $G$:

• Multiplicative constants don’t matter in $O$-notation, and so if $F(n) = O(G(n))$ then $100F(n) = O(G(n))$.

• When adding two functions, we only care about the larger one. For example, for the purpose of $O$-notation, $n^3 + 100n^2$ is the same as $n^3$, and in general in any polynomial, we only care about the larger exponent.

• For every two constants $a, b > 0$, $n^a = O(n^b)$ if and only if $a \leq b$, and $n^a = o(n^b)$ if and only if $a < b$. For example, combining the two observations above, $100n^2 + 10n + 100 = o(n^3)$.

• Polynomial is always smaller than exponential: $n^a = o(2^n)$ for every two constants $a > 0$ and $\epsilon > 0$ even if $\epsilon$ is much smaller than $a$. For example, $100n^{100} = o(2^{\sqrt{n}})$.

• Similarly, logarithmic is always smaller than polynomial: $(\log n)^a$ (which we write as $\log^a n$) is $o(n^\epsilon)$ for every two constants $a, \epsilon > 0$. For example, combining the observations above, $100n^2 \log^{100} n = o(n^3)$.

In most (though not all!) cases we use $O$-notation, the constants hidden by it are not too huge and so on an intuitive level, you can think of $F = O(G)$ as saying something like $F(n) \leq 1000G(n)$ and $F = \Omega(G)$ as saying something $F(n) \geq 0.001G(n)$.

1.5 PROOFS
Many people think of mathematical proofs as a sequence of logical deductions that starts from some axioms and ultimately arrives at a conclusion. In fact, some dictionaries define proofs that way. This
is not entirely wrong, but in reality a mathematical proof of a statement \( X \) is simply an argument that convinces the reader that \( X \) is true beyond a shadow of a doubt. To produce such a proof you need to:

1. Understand precisely what \( X \) means.
2. Convince yourself that \( X \) is true.
3. Write your reasoning down in plain, precise and concise English (using formulas or notation only when they help clarity).

In many cases, Step 1 is the most important one. Understanding what a statement means is often more than halfway towards understanding why it is true. In Step 3, to convince the reader beyond a shadow of a doubt, we will often want to break down the reasoning to “basic steps”, where each basic step is simple enough to be “self evident”. The combination of all steps yields the desired statement.

### 1.5.1 Proofs and programs

There is a great deal of similarity between the process of writing proofs and that of writing programs, and both require a similar set of skills. Writing a program involves:

1. Understanding what is the task we want the program to achieve.
2. Convincing yourself that the task can be achieved by a computer, perhaps by planning on a whiteboard or notepad how you will break it up to simpler tasks.
3. Converting this plan into code that a compiler or interpreter can understand, by breaking up each task into a sequence of the basic operations of some programming language.

In programs as in proofs, step 1 is often the most important one. A key difference is that the reader for proofs is a human being and for programs is a compiler.\(^{16}\) Thus our emphasis is on readability and having a clear logical flow for the proof (which is not a bad idea for programs as well…). When writing a proof, you should think of your audience as an intelligent but highly skeptical and somewhat petty reader, that will “call foul” at every step that is not well justified.

### 1.6 EXTENDED EXAMPLE: GRAPH CONNECTIVITY

To illustrate these ideas, let us consider the following example of a true theorem:

**Theorem 1.14 — Minimum edges for connected graphs.** Every con-
connected undirected graph of \( n \) vertices has at least \( n - 1 \) edges.

We are going to take our time to understand how one would come up with a proof for Theorem 1.14, and how to write such a proof down. This will not be the shortest way to prove this theorem, but hopefully following this process will give you some general insights on reading, writing, and discovering mathematical proofs.

Before trying to prove Theorem 1.14, we need to understand what it means. Let’s start with the terms in the theorems. We defined undirected graphs and the notion of connectivity in Section 1.4.5 above. In particular, an undirected graph \( G = (V, E) \) is connected if for every pair \( u, v \in V \), there is a path \( (u_0, u_1, \ldots, u_k) \) such that \( u_0 = u, u_k = v \), and \( \{u_i, u_{i+1}\} \in E \) for every \( i \in [k] \).

It is crucial that at this point you pause and verify that you completely understand the definition of connectivity. Indeed, you should make a habit of pausing after any statement of a theorem, even before looking at the proof, and verifying that you understand all the terms that the theorem refers to.

To prove Theorem 1.14 we need to show that there is no 2-vertex connected graph with fewer than 1 edges, 3-vertex connected graph with fewer than 2 edges, and so on and so forth. One of the best ways to prove a theorem is to first try to disprove it. By trying and failing to come up with a counterexample, we often understand why the theorem can not be false. For example, if you try to draw a 4-vertex graph with only two edges, you can see that there are basically only two choices for such a graph as depicted in Fig. 1.7, and in both there will remain some vertices that cannot be connected.

![Figure 1.7](image)

**Figure 1.7.** In a four vertex graph with two edges, either both edges have a shared vertex or they don’t. In both cases the graph will not be connected.

In fact, we can see that if we have a budget of 2 edges and we choose some vertex \( u \), we will not be able to connect to \( u \) more than two other vertices, and similarly with a budget of 3 edges we will not
be able to connect to \( u \) more than three other vertices. We can keep trying to draw such examples until we convince ourselves that the theorem is probably true, at which point we want to see how we can prove it.

If you have not seen the proof of this theorem before (or don’t remember it), this would be an excellent point to pause and try to prove it yourself. One way to do it would be to describe an algorithm that on input a graph \( G \) on \( n \) vertices and \( n - 2 \) or fewer edges, finds a pair \( u, v \) of vertices such that \( u \) is disconnected from \( v \).

1.6.1 Mathematical induction
There are several ways to prove Theorem 1.14. One approach to do is to start by proving it for small graphs, such as graphs with 2, 3 or 4 edges, for which we can check all the cases, and then try to extend the proof for larger graphs. The technical term for this proof approach is proof by induction.

Induction is simply an application of the self-evident Modus Ponens rule that says that if (a) \( P \) is true and (b) \( P \) implies \( Q \) then \( Q \) is true. In the setting of proofs by induction we typically have a statement \( Q(k) \) that is parameterized by some integer \( k \), and we prove that (a) \( Q(0) \) is true and (b) For every \( k > 0 \), if \( Q(0), \ldots, Q(k - 1) \) are all true then \( Q(k) \) is true. By repeatedly applying Modus Ponens, we can deduce from (a) and (b) that \( Q(1) \) is true, and then from (a), (b) and \( Q(1) \) that \( Q(2) \) is true, and so on and so forth to obtain that \( Q(k) \) is true for every \( k \). The statement (a) is called the “base case”, while (b) is called the “inductive step”. The assumption in (b) that \( Q(i) \) holds for \( i < k \) is called the “inductive hypothesis”.

17 Usually proving (b) is the hard part, though there are examples where the “base case” (a) is quite subtle.

Induction and recursion Proofs by inductions are closely related to algorithms by recursion. In both cases we reduce solving a larger problem to solving a smaller instance of itself. In a recursive algorithm to solve some problem \( P \) on an input of length \( k \) we ask ourselves “what if someone handed me a way to solve \( P \) on instances smaller than \( k \)”. In an inductive proof to prove a statement \( Q \) parameterized by a number \( k \), we ask ourselves “what if I already knew that \( Q(k') \) is true for \( k' < k \)”. Both induction and recursion are crucial concepts for this course and Computer Science at large (and even other areas of inquiry, including not just mathematics but other sciences as well). Both can be initially (and even post-initially) confusing, but with time and practice they become clearer. For more on proofs by induc-
1.6.2 Proving the theorem by induction

There are several ways to use induction to prove Theorem 1.14. We will do so by following our intuition above that with a budget of $k$ edges, we cannot connect to a vertex more than $k$ other vertices. That is, we will define the statement $Q(k)$ as follows:

\[ Q(k) \text{ is } \ \text{"For every graph } G = (V, E) \text{ with at most } k \text{ edges and every } u \in V, \text{ the number of vertices that are connected to } u (\text{including } u \text{ itself}) \text{ is at most } k + 1" \]

Note that $Q(n - 2)$ implies our theorem, since it means that in an $n$ vertex graph of $n - 2$ edges, there would be at most $n - 1$ vertices that are connected to $u$, and hence in particular there would be some vertex that is not connected to $u$. More formally, if we define, given any undirected graph $G$ and vertex $u$ of $G$, the set $C_G(u)$ to contain all vertices connected to $u$, then the statement $Q(k)$ is that for every undirected graph $G = (V, E)$ with $|E| = k$ and $u \in V$, $|C_G(u)| \leq k + 1$.

To prove that $Q(k)$ is true for every $k$ by induction, we will first prove that (a) $Q(0)$ is true, and then prove (b) if $Q(0), \ldots, Q(k - 1)$ are true then $Q(k)$ is true as well. In fact, we will prove the stronger statement (b') that if $Q(k - 1)$ is true then $Q(k)$ is true as well. (b') is a stronger statement than (b) because it has same conclusion with a weaker assumption.) Thus, if we show both (a) and (b') then we complete the proof of Theorem 1.14.

Proving (a) (i.e., the “base case”) is actually quite easy. The statement $Q(0)$ says that if $G$ has zero edges, then $|C_G(u)| = 1$, but this is clear because in a graph with zero edges, $u$ is only connected to itself. The heart of the proof is, as typical with induction proofs, is in proving a statement such as (b') (or even the weaker statement (b)). Since we are trying to prove an implication, we can assume the so-called “inductive hypothesis” that $Q(k - 1)$ is true and need to prove from this assumption that $Q(k)$ is true. So, suppose that $G = (V, E)$ is a graph of $k$ edges, and $u \in V$. Since we can use induction, a natural approach would be to remove an edge $e \in E$ from the graph to create a new graph $G'$ of $k - 1$ edges. We can use the induction hypothesis to argue that $|C_{G'}(u)| \leq k$. Now if we could only argue that removing the edge $e$ reduced the connected component of $u$ by at most a single vertex, then we would be done, as we could argue that $|C_G(u)| \leq |C_{G'}(u)| + 1 \leq k + 1$. 


Please ensure that you understand why showing that $|C_G(u)| \leq |C_{G'}(u)| + 1$ completes the inductive proof.

Figure 1.8: Removing a single edge $e$ can greatly decrease the number of vertices that are connected to a vertex $u$.

Alas, this might not be the case. It could be that removing a single edge $e$ will greatly reduce the size of $C_G(u)$. For example that edge might be a “bridge” between two large connected components; such a situation is illustrated in Fig. 1.8. This might seem as a real stumbling block, and at this point we might go back to the drawing board to see if perhaps the theorem is false after all. However, if we look at various concrete examples, we see that in any concrete example, there is always a “good” choice of an edge, adding which will increase the component connect to $u$ by at most one vertex.

Figure 1.9: Removing an edge $e = \{s, w\}$ where $w \in C_G(u)$ has degree one removes only $w$ from $C_G(u)$.

The crucial observation is that this always holds if we choose an edge $e = \{s, w\}$ where $w \in C_G(u)$ has degree one in the graph $G$, see Fig. 1.9. The reason is simple. Since every path from $u$ to $w$ must
pass through $s$ (which is $w$’s only neighbor), removing the edge $\{s, w\}$ merely has the effect of disconnecting $w$ from $u$, and hence $C_{G'}(u) = C_G(u) \setminus \{w\}$ and in particular $|C_{G'}(u)| = |C_G(u)| - 1$, which is exactly the condition we needed.

Now the question is whether there will always be a degree one vertex in $C_G(u) \setminus \{u\}$. Of course generally we are not guaranteed that a graph would have a degree one vertex, but we are not dealing with a general graph here but rather a graph with a small number of edges. We can assume that $|C_G(u)| > k + 1$ (otherwise we’re done) and each vertex in $C_G(u)$ must have degree at least one (as otherwise it would not be connected to $u$). Thus, the only case where there is no vertex $w \in C_G(u) \setminus \{u\}$ of degree one, is when the degrees of all vertices in $C_G(u)$ are at least 2. But then by Lemma 1.6 the number of edges in the graph is at least $\frac{1}{2} \cdot 2 \cdot (k + 1) > k$, which contradicts our assumption that the graph $G$ has at most $k$ edges. Thus we can conclude that either $|C_G(u)| \leq k + 1$ (in which case we’re done) or there is a degree one vertex $w \neq u$ that is connected to $u$. By removing the single edge $e$ that touches $w$, we obtain a $k - 1$ edge graph $G'$ which (by the inductive hypothesis) satisfies $|C_{G'}(u)| \leq k$, and hence $|C_G(u)| = |C_{G'}(u) \cup \{w\}| \leq k + 1$. This suffices to complete an inductive proof of statement $Q(k)$.

### 1.6.3 Writing down the proof

All of the above was a discussion of how we discover the proof, and convince ourselves that the statement is true. However, once we do that, we still need to write it down. When writing the proof, we use the benefit of hindsight, and try to streamline what was a messy journey into a linear and easy-to-follow flow of logic that starts with the word “Proof:” and ends with “QED” or the symbol ■. All our discussions, examples and digressions can be very insightful, but we keep them outside the space delimited between these two words, where (as described by this excellent handout) “every sentence must be load bearing”. Just like we do in programming, we can break the proof into little “subroutines” or “functions” (known as lemmas or claims in math language), which will be smaller statements that help us prove the main result. However, it should always be crystal-clear to the reader in what stage we are of the proof. Just like it should always be clear to which function a line of code belongs to, it should always be clear whether an individual sentence is part of a proof of some intermediate result, or is part of the argument showing that this intermediate result implies the theorem. Sometimes we highlight this partition by noting after each occurrence of “QED” to which lemma or claim it belongs.

Let us see how the proof of Theorem 1.14 looks in this streamlined
fashion. We start by repeating the theorem statement

**Theorem 1.15 — Minimum edges for connected graphs (restated).** Every connected undirected graph of \( n \) vertices has at least \( n - 1 \) edges.

**Proof of Theorem 1.15.** The proof will follow from the following lemma:

**Lemma 1.16** For every \( k \in \mathbb{N} \), undirected graph \( G = (V, E) \) of at most \( k \) edges, and \( u \in V \), the number of vertices connected to \( u \) in \( G \) is at most \( k + 1 \).

We start by showing that **Lemma 1.16** implies the theorem:

**Proof of Theorem 1.15 from Lemma 1.16:** We will show that for undirected graph \( G = (V, E) \) of \( n \) vertices and at most \( n - 2 \) edges, there is a pair \( u, v \) of vertices that are disconnected in \( G \). Let \( G \) be such a graph and \( u \) be some vertex of \( G \). By **Lemma 1.16**, the number of vertices connected to \( u \) is at most \( n - 1 \), and hence (since \( |V| = n \)) there is a vertex \( v \in V \) that is not connected to \( u \), thus completing the proof. **QED (Proof of Theorem 1.15 from Lemma 1.16)**

We now turn to proving **Lemma 1.16**. Let \( G = (V, E) \) be an undirected graph of \( k \) edges and \( u \in V \). We define \( C_G(u) \) to be the set of vertices connected to \( u \). To complete the proof of **Lemma 1.16**, we need to prove that \( |C_G(u)| \leq k + 1 \). We will do so by induction on \( k \).

The base case that \( k = 0 \) is true because a graph with zero edges, \( u \) is only connected to itself.

Now suppose that **Lemma 1.16** is true for \( k - 1 \) and we will prove it for \( k \). Let \( G = (V, E) \) and \( u \in V \) be as above, where \( |E| = k \), and suppose (towards a contradiction) that \( |C_G(u)| \geq k + 2 \). Let \( S = C_G(u) \setminus \{u\} \). Denote by \( \deg(v) \) the degree of any vertex \( v \). By **Lemma 1.16**, \( \sum_{v \in S} \deg(v) \leq \sum_{v \in V} \deg(v) = 2|E| = 2k \). Hence in particular, under our assumption that \( |S| + 1 = |C_G(u)| \geq k + 2 \), we get that \( \frac{1}{|S|} \sum_{v \in S} \deg(v) \leq 2k/(k + 1) < 2 \). In other words, the average degree of a vertex in \( S \) is smaller than 2, and hence in particular there is some vertex \( w \in S \) with degree smaller than 2. Since \( w \) is connected to \( u \), it must have degree at least one, and hence (since \( w \)’s degree is smaller than two) degree exactly one. In other words, \( w \) has a single neighbor which we denote by \( s \).

Let \( G' \) be the graph obtained by removing the edge \( \{s, w\} \) from \( G \). Since \( G' \) has at most \( k - 1 \) edges, by the inductive hypothesis we can assume that \( |C_{G'}(u)| \leq k \). The proof of the lemma is concluded by showing the following claim:
Claim: Under the above assumptions, \(|C_G(u)| \leq |C_{G'}(u)| + 1\).

Proof of claim: The claim says that \(C_{G'}(u)\) has at most one fewer element than \(C_G(u)\). Thus it follows from the following statement (*): 
\(C_{G'}(u) \supseteq C_G(u) \setminus \{w\}\). To prove (*) we need to show that for every \(v \neq w\) that is connected to \(u, v \in C_{G'}(u)\). Indeed for every such \(v\), Lemma 1.8 implies that there must be some simple path 
\((t_0, t_1, ..., t_{i-1}, t_i)\) in the graph \(G\) where \(t_0 = u\) and \(t_i = v\). But \(w\) cannot belong to this path, since \(w\) is different from the endpoints \(u\) and \(v\) of the path and can’t equal one of the intermediate points either, since it has degree one and that would make the path not simple. More formally, if \(w = t_j\) for \(0 < j < i\), then since \(w\) has only a single neighbor \(s\), it would have to hold that \(w\)'s neighbor \(s\) satisfies \(s = t_{j-1} = t_{j+1}\), contradicting the simplicity of the path. Hence the path from \(u\) to \(v\) is also a path in the graph \(G'\), which means that \(v \in C_{G'}(u)\), which is what we wanted to prove. QED (claim)

The claim implies Lemma 1.16 since by the inductive assumption, \(|C_{G'}(u)| \leq k\), and hence by the claim \(|C_G(u)| \leq k+1\), which is what we wanted to prove. This concludes the proof of Lemma 1.16 and hence also of Theorem 1.15. QED (Lemma 1.16), QED (Theorem 1.15)

Averaging Principle The proof above used the observation that if the average of some \(n\) numbers \(x_0, ..., x_{n-1}\) is at most \(X\), then there must exists at least a single number \(x_i \leq X\). (In this particular proof, the numbers were the degrees of vertices in \(S\).) This is known as the averaging principle, and despite its simplicity, it is often extremely useful.

Reading a proof is no less of an important skill than producing one. In fact, just like understanding code, it is a highly non-trivial skill in itself. Therefore I strongly suggest that you re-read the above proof, asking yourself at every sentence whether the assumption it makes are justified, and whether this sentence truly demonstrates what it purports to achieve. Another good habit is to ask yourself when reading a proof for every variable you encounter (such as \(u, t_i, G'\), etc. in the above proof) the follow-
1.7 PROOF WRITING STYLE

A mathematical proof is a piece of writing, but it is a specific genre of writing with certain conventions and preferred styles. As in any writing, practice makes perfect, and it is also important to revise your drafts for clarity.

In a proof for the statement $\mathcal{S}$, all the text between the words “Proof:” and “QED” should be focused on establishing that $\mathcal{S}$ is true. Digressions, examples, or ruminations should be kept outside these two words, so they do not confuse the reader. The proof should have a clear logical flow in the sense that every sentence or equation in it should have some purpose and it should be crystal-clear to the reader what this purpose is. When you write a proof, for every equation or sentence you include, ask yourself:

1. Is this sentence or equation stating that some statement is true?

2. If so, does this statement follow from the previous steps, or are we going to establish it in the next step?

3. What is the role of this sentence or equation? Is it one step towards proving the original statement, or is it a step towards proving some intermediate claim that you have stated before?

4. Finally, would the answers to questions 1-3 be clear to the reader? If not, then you should reorder, rephrase or add explanations.

Some helpful resources on mathematical writing include this handout by Lee, this handout by Hutching, as well as several of the excellent handouts in Stanford’s CS 103 class.

1.7.1 Patterns in proofs

“If it was so, it might be; and if it were so, it would be; but as it isn’t, it ain’t. That’s logic.”, Lewis Carroll, Through the looking-glass.

Just like in programming, there are several common patterns of proofs that occur time and again. Here are some examples:

Proofs by contradiction: One way to prove that $X$ is true is to show that if $X$ was false then we would get a contradiction as a result. Such
proofs often start with a sentence such as “Suppose, towards a contradiction, that \( X \) is false” and end with deriving some contradiction (such as a violation of one of the assumptions in the theorem statement). Here is an example:

**Lemma 1.17** There are no natural numbers \( a, b \) such that \( \sqrt{2} = \frac{a}{b} \).

Proof. Suppose, towards the sake of contradiction that this is false, and so let \( a \in \mathbb{N} \) be the smallest number such that there exists some \( b \in \mathbb{N} \) satisfying \( \sqrt{2} = \frac{a}{b} \). Squaring this equation we get that \( 2 = \frac{a^2}{b^2} \) or \( a^2 = 2b^2 \) (\(*\)). But this means that \( a^2 \) is even, and since the product of two odd numbers is odd, it means that \( a \) is even as well, or in other words, \( a = 2a' \) for some \( a' \in \mathbb{N} \). Yet plugging this into \((*)\) shows that \( 4a'^2 = 2b^2 \) which means \( b^2 = 2a'^2 \) is an even number as well. By the same considerations as above we get that \( b \) is even and hence \( a/2 \) and \( b/2 \) are two natural numbers satisfying \( \sqrt{2} = \frac{a/2}{b/2} \), contradicting the minimality of \( a \). ■

**Proofs of a universal statement:** Often we want to prove a statement \( \forall x \in \mathbb{N} \). Such proofs often start with a sentence such as “Let \( x \in \mathbb{N} \) be some number. If \( x/2 \) is a whole number then we are done, since then \( x = 2(x/2) \) and hence it is even. Otherwise, \( x/2 + 1/2 \) is a whole number, and hence \( 2(x/2 + 1/2) = x + 1 \) is even.

**Lemma 1.18** For every natural number \( n \in \mathbb{N} \), either \( n \) or \( n + 1 \) is even.

Proof. Let \( n \in \mathbb{N} \) be some number. If \( n/2 \) is a whole number then we are done, since then \( n = 2(n/2) \) and hence it is even. Otherwise, \( n/2 + 1/2 \) is a whole number, and hence \( 2(n/2 + 1/2) = n + 1 \) is even. ■

**Proofs of an implication:** Another common case is that the statement \( X \) has the form “\( A \) implies \( B \)”. Such proofs often start with a sentence such as “Assume that \( A \) is true” and end with a derivation of \( B \) from \( A \). Here is a simple example:

**Lemma 1.19** If \( b^2 \geq 4ac \) then there is a solution to the quadratic equation \( ax^2 + bx + c = 0 \).

Proof. Suppose that \( b^2 \geq 4ac \). Then \( d = b^2 - 4ac \) is a non-negative number and hence it has a square root \( s \). Thus \( x = (-b + s)/(2a) \) satisfies

\[
ax^2 + bx + c = a((-b + s)^2)/(4a^2) + b(-b + s)/(2a) + c = (b^2 - 2bs + s^2)/(4a) + (-b^2 + bs)/(2a) + c .
\] (1.17) ■
Rearranging the terms of Eq. (1.17) we get
\[
s^2/(4a) + c - b^2/(4a) = (b^2 - 4ac)/(4a) + c - b^2/(4a) = 0 \quad (1.18)
\]

**Proofs of equivalence:** If a statement has the form “A if and only if B” (often shortened as “A iff B”) then we need to prove both that A implies B and that B implies A. We call the implication that A implies B the “only if” direction, and the implication that B implies A the “if” direction.

**Proofs by combining intermediate claims:** When a proof is more complex, it is often helpful to break it apart into several steps. That is, to prove the statement \(X\), we might first prove statements \(X_1, X_2,\) and \(X_3\) and then prove that \(X_1 \land X_2 \land X_3\) implies \(X\).\(^{19}\) Our proof of Theorem 1.14 had this form.

**Proofs by case distinction:** This is a special case of the above, where to prove a statement \(X\) we split into several cases \(C_1, \ldots, C_k\), and prove that (a) the cases are exhaustive, in the sense that one of the cases \(C_i\) must happen and (b) go one by one and prove that each one of the cases \(C_i\) implies the result \(X\) that we are after.

**“Without loss of generality (w.l.o.g)”**: This term can be initially quite confusing to students. It is essentially a way to shorten case distinctions such as the above. The idea is that if Case 1 is equal to Case 2 up to a change of variables or a similar transformation, then the proof of Case 1 will also imply the proof of case 2. It is always a statement that should be viewed with suspicion. Whenever you see it in a proof, ask yourself if you understand why the assumption made is truly without loss of generality, and when you use it, try to see if the use is indeed justified. Sometimes it might be easier to just repeat the proof of the second case (adding a remark that the proof is very similar to the first one).

**Proofs by induction:** We can think of such proofs as a variant of the above, where we have an unbounded number of intermediate claims \(X_0, X_1, \ldots, X_k\), and we prove that \(X_0\) is true, as well that \(X_0\) implies \(X_1\), and that \(X_0 \land X_1\) implies \(X_2\), and so on and so forth. The website for CMU course 15-251 contains a useful handout on potential pitfalls when making proofs by induction.

---

\(^{19}\) As mentioned below, \(\land\) denotes the logical AND operator.
the assumption that $A$ is true, and ending with
the claim that $B$ is true.

- Every claim is followed by a proof showing how
it is derived from the previous assumptions or
claims.
- The proof for each claim is itself a sequence of
subclaims.

The advantage of Lamport’s format is that it is very
clear for every sentence in the proof what is the
role that it plays. It is also much easier to transform
such proofs into machine-checkable format. The
dis disadvantage is that such proofs can be more tedious
to read and write, with less differentiation on the
important parts of the arguments versus the more
routine ones.

1.8 NON-STANDARD NOTATION

Most of the notation we discussed above is standard and is used in
most mathematical texts. The main points where we diverge are:

- We index the natural numbers $\mathbb{N}$ starting with 0 (though many
  other texts, especially in computer science, do the same).

- We also index the set $[n]$ starting with 0, and hence define it as
  $\{0, \ldots, n - 1\}$. In most texts it is defined as $\{1, \ldots, n\}$. Similarly, we
index coordinates of our strings starting with 0, and hence a string
$x \in \{0,1\}^n$ is written as $x_0x_1\cdots x_{n-1}$.

- We use partial functions which are functions that are not necessarily
defined on all inputs. When we write $f : A \to B$ this will refer to a
total function unless we say otherwise. When we want to emphasize
that $f$ can be a partial function, we will sometimes write $f : A \to \partial B$.

- As we will see later on in the course, we will mostly describe our
computational problems in the terms of computing a Boolean function
$f : \{0,1\}^* \to \{0,1\}$. In contrast, most textbooks will refer
to this as the task of deciding a language $L \subseteq \{0,1\}^*$. These two
viewpoints are equivalent, since for every set $L \subseteq \{0,1\}^*$ there is
a corresponding function $f = 1_L$ such that $f(x) = 1$ if and only if
$x \in L$. Computing partial functions corresponds to the task known
in the literature as a solving a promise problem.\footnote{Because the language notation is
so prevalent in textbooks, we will occasionally remind the reader of this
correspondence.}

- Some other notation we use is $\lceil x \rceil$ and $\lfloor x \rfloor$ for the “ceiling” and
“floor” operators that correspond to “rounding up” or “rounding
down” a number to the nearest integer. We use $(x \mod y)$
to denote the “remainder” of $x$ when divided by $y$. That is, $(x \mod y) = x - y\lfloor x/y \rfloor$. In context when an integer is expected we’ll typically “silently round” the quantities to an integer. For example, if we say that $x$ is a string of length $\sqrt{n}$ then we’ll typically mean that $x$ is of length $\lceil \sqrt{n} \rceil$. (In most such cases, it will not make a difference whether we round up or down.)

• Like most Computer Science texts, we default to the logarithm in base two. Thus, $\log n$ is the same as $\log_2 n$.

• We will also use the notation $f(n) = poly(n)$ as a short hand for $f(n) = n^{O(1)}$ (i.e., as shorthand for saying that there are some constants $a, b$ such that $f(n) \leq a \cdot n^b$ for every sufficiently large $n$). Similarly, we will use $f(n) = polylog(n)$ as shorthand for $f(n) = poly(\log n)$ (i.e., as shorthand for saying that there are some constants $a, b$ such that $f(n) \leq a \cdot (\log n)^b$ for every sufficiently large $n$).

---

**Lecture Recap**

• The basic “mathematical data structures” we’ll need are numbers, sets, tuples, strings, graphs and functions.

• We can use basic objects to define more complex notions. For example, graphs can be defined as a list of pairs.

• Given precise definitions of objects, we can state unambiguous and precise statements. We can then use mathematical proofs to determine whether these statements are true or false.

• A mathematical proof is not a formal ritual but rather a clear, precise and “bulletproof” argument certifying the truth of a certain statement.

• Big-$O$ notation is an extremely useful formalism to suppress less significant details and allow us to focus on the high level behavior of quantities of interest.

• The only way to get comfortable with mathematical notions is to apply them in the contexts of solving problems. You should expect to need to go back time and again to the definitions and notation in this lecture as you work through problems in this course.
1.9 EXERCISES

Disclaimer: Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 1.1 — Logical expressions. 1. Write a logical expression \( \varphi(x) \) involving the variables \( x_0, x_1, x_2 \) and the operators \( \land \) (AND), \( \lor \) (OR), and \( \neg \) (NOT), such that \( \varphi(x) \) is true if the majority of the inputs are \textit{True}.

2. Write a logical expression \( \varphi(x) \) involving the variables \( x_0, x_1, x_2 \) and the operators \( \land \) (AND), \( \lor \) (OR), and \( \neg \) (NOT), such that \( \varphi(x) \) is true if the sum \( \sum_{i=0}^{2} x_i \) (identifying “true” with 1 and “false” with 0) is \textit{odd}.

Exercise 1.2 — Quantifiers. Use the logical quantifiers \( \forall \) (for all), \( \exists \) (there exists), as well as \( \land, \lor, \neg \) and the arithmetic operations \( +, \times, =, >, < \) to write the following:

1. An expression \( \varphi(n, k) \) such that for every natural numbers \( n, k \), \( \varphi(n, k) \) is true if and only if \( k \) divides \( n \).

2. An expression \( \varphi(n) \) such that for every natural number \( n \), \( \varphi(n) \) is true if and only if \( n \) is a power of three.

Exercise 1.3 — Set construction notation. Describe in words the following sets:

1. \( S = \{ x \in \{0,1\}^{100} : \forall_{i \in \{0,\ldots,99\}} x_i = x_{99-i} \} \)

2. \( T = \{ x \in \{0,1\}^* : \forall_{i, j \in \{2,\ldots,|x|-1\}} i \cdot j \neq |x| \} \)

Exercise 1.4 — Existence of one to one mappings. For each one of the following pairs of sets \( (S, T) \), prove or disprove the following statement: there is a one to one function \( f \) mapping \( S \) to \( T \).

1. Let \( n > 10 \). \( S = \{0,1\}^n \) and \( T = [n] \times [n] \times [n] \).

2. Let \( n > 10 \). \( S \) is the set of all functions mapping \( \{0,1\}^n \) to \( \{0,1\} \). \( T = \{0,1\}^{n^3} \).
3. Let $n > 100$. $S = \{ k \in [n] \mid k \text{ is prime} \}$, $T = \{0, 1\}^{\lceil \log n - 1 \rceil}$.

**Exercise 1.5** — Inclusion Exclusion. 1. Let $A, B$ be finite sets. Prove that $|A \cup B| = |A| + |B| - |A \cap B|$.

2. Let $A_0, \ldots, A_{k-1}$ be finite sets. Prove that $|A_0 \cup \cdots \cup A_{k-1}| \geq \sum_{i=0}^{k-1} |A_i| - \sum_{0 \leq i < j < k} |A_i \cap A_j|$.

3. Let $A_0, \ldots, A_{k-1}$ be finite subsets of $\{1, \ldots, n\}$, such that $|A_i| = m$ for every $i \in [k]$. Prove that if $k > 100n$, then there exist two distinct sets $A_i, A_j$ s.t. $|A_i \cap A_j| \geq m^2/(10n)$.

**Exercise 1.6** Prove that if $S, T$ are finite and $F : S \to T$ is one to one then $|S| \leq |T|$.

**Exercise 1.7** Prove that if $S, T$ are finite and $F : S \to T$ is onto then $|S| \geq |T|$.

**Exercise 1.8** Prove that for every finite $S, T$, there are $([T] + 1)^{|S|}$ partial functions from $S$ to $T$.

**Exercise 1.9** Suppose that $\{S_n\}_{n \in \mathbb{N}}$ is a sequence such that $S_0 \leq 10$ and for $n > 1$ $S_n \leq 5S_{\lceil n/5 \rceil} + 2n$. Prove by induction that $S_n \leq 100n \log n$ for every $n$.

**Exercise 1.10** Describe the following statement in English words: $\forall n \in \mathbb{N} \exists p \in \mathbb{P}, \forall a, b \in \mathbb{N} (a \times b \neq p) \lor (a = 1)$.

**Exercise 1.11** Prove that for every undirected graph $G$ of 100 vertices, if every vertex has degree at most 4, then there exists a subset $S$ of at 20 vertices such that no two vertices in $S$ are neighbors of one another.

**Exercise 1.12** — $O$-notation. For every pair of functions $F, G$ below, determine which of the following relations holds: $F = O(G)$, $F = \Omega(G)$, $F = o(G)$ or $F = \omega(G)$.

1. $F(n) = n, G(n) = 100n$.
2. $F(n) = n, G(n) = \sqrt{n}$.
3. $F(n) = n \log n, G(n) = 2^{\log(n)^2}$.
4. $F(n) = \sqrt{n}, G(n) = 2^{\sqrt{\log n}}$.
5. $F(n) = \binom{n}{0.2n}, G(n) = 2^{0.1n}$.

**Exercise 1.13** Give an example of a pair of functions $F, G : \mathbb{N} \to \mathbb{N}$ such that neither $F = O(G)$ nor $G = O(F)$ holds.
Exercise 1.14 — **Topological sort.** Prove that for every directed acyclic graph (DAG) \( G = (V, E) \), there exists a map \( f : V \to \mathbb{N} \) such that \( f(u) < f(v) \) for every edge \( 
abla u\rightarrow v \) in the graph.\(^{21}\)

**Exercise 1.15** Prove that for every undirected graph \( G \) on \( n \) vertices, if \( G \) has at least \( n \) edges then \( G \) contains a cycle.

### 1.10 BIBLIOGRAPHICAL NOTES

The section heading “A Mathematician’s Apology”, refers of course to Hardy’s **classic book**. Even when Hardy is wrong, he is very much worth reading.

\(^{21}\) Hint: Use induction on the number of vertices. You might want to first prove the claim that every DAG contains a *sink*: a vertex without an outgoing edge.
2
Computation and Representation

“The alphabet was a great invention, which enabled men to store and to learn with little effort what others had learned the hard way – that is, to learn from books rather than from direct, possibly painful, contact with the real world.”, B.F. Skinner

“The name of the song is called 'HADDOCK'S EYES.'” [said the Knight]

“Oh, that's the name of the song, is it?” Alice said, trying to feel interested.

“No, you don't understand,” the Knight said, looking a little vexed. “That's what the name is CALLED. The name really is 'THE AGED AGED MAN.'”

“Then I ought to have said ‘That's what the SONG is called’?” Alice corrected herself.

“No, you oughtn't: that's quite another thing! The SONG is called 'WAYS AND MEANS': but that's only what it's CALLED, you know!”

“Well, what IS the song, then?” said Alice, who was by this time completely bewildered.

“I was coming to that,” the Knight said. “The song really IS 'A-SITTING ON A GATE': and the tune's my own invention.”

Lewis Carroll, Through the looking glass

To a first approximation, computation can be thought of as a process that maps an input to an output.

When discussing computation, it is important to separate the question of what is the task we need to perform (i.e., the specification) from the question of how we achieve this task (i.e., the implementation). For example, as we’ve seen, there is more than one way to achieve the

Learning Objectives:
• Representing an object as a string (often of zeroes and ones).
• Examples of representations for common objects such as numbers, vectors, lists, graphs.
• Prefix-free representations.
• Distinguish between specification and implementation, or equivalently between algorithms/programs and mathematical functions.
computational task of computing the product of two integers.

In this chapter we focus on the what part, namely defining computational tasks. For starters, we need to define the inputs and outputs. A priori this seems nontrivial, since computation today is applied to a huge variety of objects. We do not compute merely on numbers, but also on texts, images, videos, connection graphs of social networks, MRI scans, gene data, and even other programs. We will represent all these objects as strings of zeroes and ones, that is objects such as 0011101 or 1011 or any other finite list of 1’s and 0’s.

Figure 2.1: Our basic notion of computation is some process that maps an input to an output.

Figure 2.2: We represent numbers, texts, images, networks and many other objects using strings of zeroes and ones. Writing the zeroes and ones themselves in green font over a black background is optional.

Today, we are so used to the notion of digital representation that we are not surprised by the existence of such an encoding. But it is a deep insight with significant implications. Many animals can convey a particular fear or desire, but what’s unique about humans is language: we use a finite collection of basic symbols to describe a potentially
unlimited range of experiences. Language allows transmission of information over both time and space, and enables societies that span a great many people and accumulate a body of shared knowledge over time.

Over the last several decades, we’ve seen a revolution in what we are able to represent and convey in digital form. We can capture experiences with almost perfect fidelity, and disseminate it essentially instantaneously to an unlimited audience. What’s more, once information is in digital form, we can compute over it, and gain insights from data that were not accessible in prior times. At the heart of this revolution is this simple but profound observation that we can represent an unbounded variety of objects using a finite set of symbols (and in fact using only the two symbols \(0\) and \(1\)).

In later lectures, we will often fall back on taking this representation for granted, and hence write something like “program \(P\) takes \(x\) as input” when \(x\) might be a number, a vector, a graph, or any other objects, when we really mean that \(P\) takes as input the representation of \(x\) as a binary string. However, in this chapter, let us dwell a little bit on how such representations can be devised.

### 2.1 Examples of Binary Representations

In many instances, choosing the “right” string representation for a piece of data is highly nontrivial, and finding the “best” one (e.g., most compact, best fidelity, most efficiently manipulable, robust to errors, most informative features, etc.) is the object of intense research. But for now, let us start by describing some simple representations for various natural objects.

#### 2.1.1 Representing natural numbers

Perhaps the simplest object we want to represent is a natural number. That is, a member \(x\) of the set \(\mathbb{N} = \{0, 1, 2, 3, \ldots\}\). We can represent a number \(x \in \mathbb{N}\) as a string using the binary basis. Specifically, every natural number \(x\) can be written in a unique way as

\[
x = x_0 \cdot 2^0 + x_1 \cdot 2^1 + \cdots + x_n \cdot 2^{n-1}
\]

(or \(\sum_{i=0}^{n-1} x_i \cdot 2^i\) for short) where \(x_0, \ldots, x_{n-1}\) are zero/one and \(n\) is the smallest number such that \(2^n > x\) (and hence \(x_{n-1} = 1\) for every nonzero \(x\)). We can then represent \(x\) as the string \((x_0, x_1, \ldots, x_{n-1})\). For example, the number 35 is represented as the string \((1, 1, 0, 0, 0, 1)\).

We can think of a representation as consisting of encoding and decoding functions. In the case of the binary representation for integers, the encoding function \(E : \mathbb{N} \to \{0, 1\}^*\) maps a natural number to the string representing it, and the decoding function \(D : \{0, 1\}^* \to \mathbb{N}\) maps a string into the number it represents (i.e., \(D(x_0, \ldots, x_{n-1}) = 2^0 x_0 + 2^1 x_1 + \cdots + 2^{n-1} x_{n-1}\) for every \(x_0, \ldots, x_{n-1} \in \{0, 1\}\) in the

\footnote{There is nothing “holy” about using zero and one as the basic symbols, and we can (indeed sometimes people do) use any other finite set of two or more symbols as the fundamental “alphabet”. We use zero and one in this course mainly because it simplifies notation.}

\footnote{We can represent the number zero either as some string that contains only zeroes, or as the empty string. The choice will not make any difference for us.}

\footnote{Typically when people write down the binary representation, they would print the string \(x\) in reverse order, with the least significant digit as the rightmost one. Representing the number \(x\) as \((x_{n-1}, x_{n-2}, \ldots, x_0)\) will of course work just as well. We chose the particular representation above for the sake of simplicity, so the the \(i\)-th bit corresponds to \(2^i\), but such low level choices will not make a difference in this course. A related, but not identical, distinction is the Big Endian vs Little Endian representation for integers in computing architecture.}
Python programming language, we can compute these encoding and decoding functions as follows:

```python
from math import floor, log

def int2bits(n):
    return [floor(n / 2**i) % 2 for i in range(floor(log(n, 2)) + 1)]

print(int2bits(236))  # [0, 0, 1, 1, 0, 1, 1, 1]
print(int2bits(19))   # [1, 1, 0, 0, 1]

def bits2int(L):
    return sum([2**i * L[i] for i in range(len(L))])

print(bits2int([0, 0, 1, 1, 0, 1, 1, 1]))  # 236
```

**Programming examples** In this book, we will often illustrate our points by using programming languages to present certain computations. Our examples will be fairly short, and our point will always be to emphasize that certain computations can be done concretely, rather than focus on a particular language feature. We often use Python, but that choice is rather arbitrary. Indeed, one of the messages of this course is that all programming languages are in some sense equivalent to one another, and hence we could have just as well used JavaScript, C, COBOL, Visual Basic or even BrainF*ck.

This is not a programming course, and it is absolutely fine if you are not familiar with Python and don’t follow the fine points of code examples such as the above. Still you might find it instructive to try to parse them, with the help of websites such as Google or Stackoverflow. In particular, the function `int2bits` above uses the fact that the binary representation of a number $n$ is the list \([\lfloor \frac{n}{2} \rfloor, \ldots, \lfloor \log_2 n \rfloor]\), which in Python-speak is written as \([\text{floor}(n / 2**i) \mod 2)_{i=0}^{\log_2 n}\).

**Well defined representations.** For a representation to be well defined, we need every natural number to be represented by some string, where two distinct numbers must have distinct representations. This corresponds to requiring the encoding function to be one-to-one, and the decoding function to be onto.
While the Babylonians already invented a positional system much earlier, the decimal positional system we use today was invented by Indian mathematicians around the third century. It was taken up by Arab mathematicians in the 8th century. It was mainly introduced to Europe in the 1202 book “Liber Abaci” by Leonardo of Pisa, also known as Fibonacci, but did not displace Roman numerals in common usage until the 15th century.

If you don’t remember the definitions of one-to-one, onto, total and partial functions, now would be an excellent time to review them. Make sure you understand why the function $E$ described above is one-to-one, and the function $D$ is onto.

Meaning of representation It is natural for us to think of 236 as the “actual” number, and of 00110111 as “merely” its representation. However, for most Europeans in the middle ages CCXXXVI would be the “actual” number and 236 (if they have heard about it) would be the weird Hindu-Arabic positional representation. When our AI robot overlords materialize, they will probably think of 00110111 as the “actual” number and of 236 as “merely” a representation that they need to use when they give commands to humans.

So what is the “actual” number? This is a question that philosophers of mathematics have pondered over the generations. Plato argued that mathematical objects exist in some ideal sphere of existence (that to a certain extent is more “real” than the world we perceive via our senses, as this latter world is merely the shadow of this ideal sphere). Thus in Plato’s vision the symbols 236 are merely notation for some ideal object, that, in homage to the late musician, we can refer to as “the number commonly represented by 236”. Wittgenstein argued that mathematical objects don’t exist at all, and the only thing that exists are the actual splotches on paper that make up 236, 00110111 or CCXXXVI, and mathematics are just about formal manipulation of symbols that don’t have any inherent meaning. You can also think of the “actual” number as (somewhat recursively) “that thing which is common to 236, 00110111 and CCXXXVI and all other past and future representations meant to capture the same object”. (Some mathematicians would say that the actual number can be thought of as an equivalence class of these representations.)

In this course you are free to choose your own philosophy of mathematics, as long as you maintain the distinction between the mathematical objects themselves to the particular choice of representing them, whether as splotches of ink, pixels on a screen, zeroes and one, or in another format.

2.1.2 Representing (potentially negative) integers Now that we can represent natural numbers, we can represent the full set of integers (i.e., members of the set $\mathbb{Z}$ =
In programming language, the compiler or interpreter determines the representation of the sequence of bits corresponding to a variable based on the variable’s type. Recall that the concatenation of two strings $x$ and $y$ is the string of length $|x| + |y|$ obtained by writing $y$ after $x$.

The decoding function of a representation should always be onto, since every object must be represented by some string. However, it does not always have to be one to one. For example, in this particular representation the two strings 1 and 0 both represent the number zero (since they can be thought of as representing $-0$ and $+0$ respectively, can you see why?). We can also allow a partial decoding function for representations. For example, in the representation above there is no number that is represented by the empty string. But this is still a fine representation, since the decoding partial function is onto and the encoding function is the one-to-one total function $E : \mathbb{Z} \rightarrow \{0,1\}^*$ which maps an integer of the form $a \times k$, where $a \in \{-1,1\}$ and $k \in \mathbb{N}$, to the bit $(-1)^a$ concatenated with the binary representation of $k$. That is, every integer can be represented as a string, and every two distinct integers have distinct representations.

2.1.3 Representing rational numbers

We can represent a rational number of the form $\frac{a}{b}$ by representing the two numbers $a$ and $b$ (again, this is not a unique representation but this is fine). However, simply concatenating the representations of $a$ and $b$ will not work. For example, recall that we represent $4$ as $(0,0,1)$ and $35$ as $(1,1,0,0,0,1)$, but the concatenation $(0,0,1,1,0,0,0,1)$ of these strings is also the concatenation of the representation $(0,0,1,1)$ of $12$ and the representation $(1,0,0,0,1)$ of $35$, so it cannot represent $\frac{4}{35}$. In programming language, the compiler or interpreter determines the representation of the sequence of bits corresponding to a variable based on the variable’s type.

\[ (-1)^x [x_0 2^0 + \cdots x_{n-1} 2^n] \] (2.1)
17. Hence, if we used such simple concatenation then we would not be able to tell if the string \((0, 0, 1, 1, 1, 0, 0, 1)\) is supposed to represent \(4/35\) or \(12/17\).\(^7\)

The way to tackle this is to find a general representation for pairs of numbers. If we were using a pen and paper, we would simply use a separator such as the symbol \(\parallel\) to represent, for example, the pair consisting of the numbers represented by \((0, 1)\) and \((1, 0, 0, 0, 1)\) as the length-9 string \(s = “01110001”\). This is just like people add spaces and punctuation to separate words in English. By adding a little redundancy, we can do just that in the digital domain. The idea is that we will map the three element set \(\Sigma = \{0, 1, \parallel\}\) to the four element set \(\{0, 1\}^2\) via the one-to-one map that takes \(0\) to \(00\), \(1\) to \(11\), \(\parallel\) to \(01\).

**Example 2.1 — Representing a rational number as a string.** Consider the rational number \(r = 19/236\). In our convention, we represent 19 as the string 11001 and 236 as the string 00110111, and so we could represent \(r\) as the pair of strings \((11001, 00110111)\). We can then represent this pair as the length 14 string 1100100110111 over the alphabet \(\{0, 1, \parallel\}\). Now, applying the map \(0 \mapsto 00, 1 \mapsto 11, \parallel \mapsto 01\), we can represent the latter string as the length 28 string \(s = 11110000110100011111001111\) over the alphabet \(\{0, 1\}\). So we represent the rational number \(r = 19/36\) be the binary string \(s = 11110000110100011111001111\).

More generally, we obtained a representation of the non-negative rational numbers as binary strings by composing the following representations:

1. Representing a non-negative rational number as a pair of natural numbers.
2. Representing a natural number by a string via the binary representation. (We can use the representation of integers to handle rational numbers that can be negative.)
3. Combining 1 and 2 to obtain representation of a rational number as a pair of strings.
4. Representing a pair of strings over \(\{0, 1\}\) as a single string over \(\Sigma = \{0, 1, \parallel\}\).
5. Representing a string over \(\Sigma\) as a longer string over \(\{0, 1\}\).

More generally, the above encoding yields a one-to-one map \(E\) from strings over the alphabet \(\Sigma\) to binary strings, such that for every \(s \in \Sigma^*\), \(|E(s)| = 2|s|\). Using this, we get a one-to-one map \(E' : (\{0, 1\}^*) \times (\{0, 1\}^*) \rightarrow \{0, 1\}^*\) mapping pairs of binary strings into a single binary
string. Given every pair \((a, b)\) of binary strings, we will first map it in a one-to-one way to a string \(s \in \Sigma^*\) using \(\|\) as a separator, and then map \(s\) to a single (longer) binary string using the encoding \(E\). The same idea can be used to represent triples, quadruples, and generally all tuples of strings as a single string (can you see why?)?

### 2.2 Representing Real Numbers

The set of real numbers \(\mathbb{R}\) contains all numbers including positive, negative, and fractional, as well as irrational numbers such as \(\pi\) or \(e\). Every real number can be approximated by a rational number, and so up to a small error we can represent every real number \(x\) by a rational number \(a/b\) that is very close to \(x\). For example, we can represent \(\pi\) by \(22/7\) with an error of about \(10^{-3}\) and if we wanted smaller error (e.g., about \(10^{-4}\)) then we can use \(311/99\) and so on and so forth.

This is a fine representation though a more common choice to represent real numbers is the floating point representation, where we represent \(x\) by the pair \((a, b)\) of (positive or negative) integers of some prescribed sizes (determined by the desired accuracy) such that \(a \times 2^b\) is closest to \(x\). The reader might be (rightly) worried about this issue of approximation. In many (though not all) computational applications, one can make the accuracy tight enough so that this does not affect the final result, though sometimes we do need to be careful. This representation is called “floating point” because we can think of the number \(a\) as specifying a sequence of binary digits, and \(b\) as describing the location of the “binary point” within this sequence. The use of floating representation is the reason why in many programming systems printing the expression \(0.1 + 0.2\) will result in \(0.30000000000000004\) and not \(0.3\), see here, here and here for more. A floating point error has been implicated in the explosion of the Ariane 5 rocket, a bug that cost more than 370 million dollars, and the failure of a U.S. Patriot missile to intercept an Iraqi Scud missile, costing 28 lives. Floating point is often problematic in financial applications as well.

#### 2.2.1 Can we represent reals exactly?

Given the issues with floating point representation, we could ask whether we could represent real numbers exactly as strings. Unfortunately, the following theorem says this cannot be done.

**Theorem 2.2 — Reals are uncountable.** There is no one-to-one function \(RtS : \mathbb{R} \to \{0, 1\}^*\).  

*You can think of this as related to scientific notation.* In scientific notation we represent a number \(x\) as an integer \(r\) \times \(10^s\) for integers \(r, s\). Sometimes we write this as \(x = r \times E^s\). For example, in many programming languages \(1.21E2\) is the same as \(121.0\). In scientific notation, to represent \(\pi\) up to accuracy \(10^{-3}\) we will simply use \(3141 \times 10^{-3}\) and to represent it up to accuracy \(10^{-4}\) we will use \(31415 \times 10^{-4}\).

### Theorem 2.2

This representation is called “floating point” because we can think of the number \(a\) as specifying a sequence of binary digits, and \(b\) as describing the location of the “binary point” within this sequence. The use of floating representation is the reason why in many programming systems printing the expression \(0.1 + 0.2\) will result in \(0.30000000000000004\) and not \(0.3\), see here, here and here for more. A floating point error has been implicated in the explosion of the Ariane 5 rocket, a bug that cost more than 370 million dollars, and the failure of a U.S. Patriot missile to intercept an Iraqi Scud missile, costing 28 lives. Floating point is often problematic in financial applications as well.

Theorem 2.2 was proven by Georg Cantor in 1874. The result (and the theory around it) was quite shocking to mathematicians at the time.
time. By showing that there is no one-to-one map from \( \mathbb{R} \) to \( \{0,1\}^* \) (or \( \mathbb{N} \)), Cantor showed that these two infinite sets have “different forms of infinity” and that the set of real numbers \( \mathbb{R} \) is in some sense “bigger” than the infinite set \( \{0,1\}^* \). The notion that there are “shades of infinity” was deeply disturbing to mathematicians and philosophers at the time. The philosopher Ludwig Wittgenstein called Cantor’s results “utter nonsense” and “laughable”. Others thought they were even worse than that. Leopold Kronecker called Cantor a “corrupter of youth”, while Henri Poincaré said that Cantor’s ideas “should be banished from mathematics once and for all”. The tide eventually turned, and these days Cantor’s work is universally accepted as the cornerstone of set theory and the foundations of mathematics. As we will see later in this course, Cantor’s ideas also play a huge role in the theory of computation.

Now that we discussed the theorem’s importance, let us see the proof. **Theorem 2.2** follows from the following two results:

**Lemma 2.3** Let \( \{0,1\}^\infty \) be the set \( \{f \mid f : \mathbb{N} \to \{0,1\}\} \) of functions from \( \mathbb{N} \) to \( \{0,1\} \).\(^{11}\) Then there is no one-to-one map \( \text{FtS} : \{0,1\}^\infty \to \{0,1\}^* \).\(^{12}\)

**Lemma 2.4** There does exist a one-to-one map \( \text{FtR} : \{0,1\}^\infty \to \mathbb{R} \).\(^{13}\)

**Lemma 2.3** and **Lemma 2.4** together imply **Theorem 2.2**. To see why, suppose, for the sake of contradiction, that there did exist a one-to-one function \( \text{E} : \{0,1\}^* \to \{0,1\}^\infty \) (e.g., see **Lemma 1.4**). We will derive a contradiction by coming up with some function \( \xi : \mathbb{N} \to \{0,1\} \) such that \( \xi \neq \text{E}(\xi) \) for every \( \xi \in \{0,1\}^* \). Since two functions are identical if and only if they agree on every input, to do this we need to show that there is some \( \eta \in \mathbb{N} \) such that \( f(\eta) \neq \xi(\eta) \). (All these quantifiers can be confusing, so let’s again recap where we are and where we want to get to. We assumed by contradiction there is a one-to-one \( \text{FtS} \) and hence

\(^{11}\) We can also think of \( \{0,1\}^\infty \) as the set of all infinite sequences of bits, since a function \( f : \mathbb{N} \to \{0,1\} \) can be identified with the sequence \( (f(0), f(1), f(2), \ldots) \).

\(^{12}\) \( \text{FtS} \) stands for “functions to strings”.

\(^{13}\) \( \text{FtR} \) stands for “functions to reals.”
Figure 2.3. We prove Theorem 2.2 by combining Lemma 2.3 and Lemma 2.4. Lemma 2.4, which uses standard calculus tools, shows the existence of a 1 to 1 map \( \text{FtF} \) from the set \( \{0,1\}^\infty \) to the real numbers. So, if a hypothetical 1 to 1 map \( \text{RsS} : \mathbb{R} \to \{0,1\}^* \) existed, then we could compose them to get a 1 to 1 map \( \text{FtS} : \{0,1\}^\infty \to \{0,1\}^* \). Yet this contradicts Lemma 2.3 - the heart of the proof - which rules out the existence of such a map.

The idea is to construct \( f^* \) iteratively: for every \( x \in \{0,1\}^* \) we will “ruin” \( f^* \) in one input \( n(x) \in \mathbb{N} \) to ensure that \( f^*(n(x)) \neq g(n(x)) \) where \( g = StF(x) \). If we are successful then this would ensure that \( f^\neq StF(x) \) for every \( x \). Specifically, for every \( x \in \{0,1\}^* \), let \( n(x) \in \mathbb{N} \) be the number \( x_0 + 2x_1 + 4x_2 + \cdots + 2k-1x_{k-1} + 2^k \) where \( k = |x| \). That is, \( n(x) = 2^k + \sum_{i=0}^{k-1} 2^i x_i \). If \( x \neq x' \) then \( n(x) \neq n(x') \) (we leave verifying this as an exercise to you, the reader).

Now for every \( x \in \{0,1\}^* \), we define

\[
f^*(n(x)) = 1 - g(n(x)) \tag{2.2}
\]

where \( g = StF(x) \). For every \( n \) that is not of the form \( n = n(x) \) for some \( x \), we set \( f^*(n) = 0 \). Eq. (2.2) is well defined since the map \( x \mapsto n(x) \) is one-to-one and hence we will not try to give \( f^*(n) \) two different values.

Now by Eq. (2.2), for every \( x \in \{0,1\}^* \), if \( g = StF(x) \) and \( n = n(x) \) then \( f^*(n) = 1 - g(n) \neq g(n) \). Hence \( StF(x) \neq f^* \) for every \( x \in \{0,1\}^* \), contradicting the assumption that \( StF \) is onto. This proof is known as the “diagonal” argument, as the construction of \( f^* \) can be thought of as going over the diagonal elements of a table that in the \( n \)-th row and \( m \)-column contains \( StF(x)(m) \) where \( x \) is the string such that \( n(x) = n \), see Fig. 2.4.
We construct a function $f^*$ such that $f^* \neq StF(x)$ for every $x \in \{0,1\}^*$ by ensuring that $f^*(n(x)) \neq StF(x)(n(x))$ for every $x \in \{0,1\}^*$. We can think of this as building a table where the columns correspond to numbers $m \in \mathbb{N}$ and the rows correspond to $x \in \{0,1\}^*$ (sorted according to $n(x)$). If the entry in the $x$-th row and the $m$-th column corresponds to $f^*(n(x))$ where $f^* = StF(x)$ then $f^*$ is obtained by going over the “diagonal” elements in this table (the entries corresponding to the $x$-th row and $n(x)$-th column) and ensuring that $f^*(n(x)) \neq StF(x)(n(x))$.

Generalizing beyond strings and reals. Lemma 2.3 doesn’t really have much to do with the natural numbers or the strings. An examination of the proof shows that it really shows that for every set $S$, there is no one-to-one map $F : \{0,1\}^S \rightarrow S$ where $\{0,1\}^S$ denotes the set $\{ f \mid f : S \rightarrow \{0,1\} \}$ of all Boolean functions with domain $S$. Since we can identify a subset $V \subseteq S$ with its characteristic function $f = 1_V$ (i.e., $1_V(x) = 1$ iff $x \in V$), we can think of $\{0,1\}^S$ also as the set of all subsets of $S$. This subset is sometimes called the power set of $S$. The proof of Lemma 2.3 can be generalized to show that there is no one-to-one map between a set and its power set. In particular, it means that the set $\{0,1\}^S$ is “even bigger” than $\mathbb{R}$. Cantor used these ideas to construct an infinite hierarchy of shades of infinity. The number of such shades turn out to be much larger than $|\mathbb{N}|$ or even $|\mathbb{R}|$. He denoted the cardinality of $\mathbb{N}$ by $\aleph_0$, where $\aleph$ is the first letter in the Hebrew alphabet, and called the the next largest infinite number by $\aleph_1$. Cantor also made the continuum hypothesis that $|\mathbb{R}| = \aleph_1$. We will come back to the very interesting story of this hypothesis later on in this course. This lecture of Aaronson mentions some of these issues (see also this Berkeley CS 70 lecture).

To complete the proof of Theorem 2.2, we need to show
Lemma 2.4. This requires some calculus background, but is otherwise straightforward. The idea is that we can construct a one-to-one map from \( \{0, 1\}^\infty \) to the real numbers by mapping the function \( f : \mathbb{N} \to \{0, 1\} \) to the number that has the infinite decimal expansion \( f(0)f(1)f(2)f(3)f(4)f(5) \ldots \) (i.e., the number between 0 and 2 that is \( \sum_{i=0}^{\infty} f(i)10^{-i} \)). We will now do this more formally. If you have not had much experience with limits of real series before, then the formal proof might be a little hard to follow. This part is not the core of Cantor’s argument, nor are such limits very crucial to this course, so feel free to also just take Lemma 2.4 on faith and skip the formal proof.

Proof of Lemma 2.4. For every \( f \in \{0, 1\}^\infty \) and \( n \in \mathbb{N} \), we define \( S(f)_n = \sum_{i=0}^{n} f(i)10^{-i} \). It is a known result (that we won’t repeat here) that for every \( f : \mathbb{N} \to \{0, 1\} \), the sequence \( (S(f)_n)_{n=0}^{\infty} \) has a limit. That is, for every \( f \) there exists some value \( x(f) \) (often denoted as \( \sum_{i=0}^{\infty} f(i)10^{-i} \)) such that for every \( \varepsilon > 0 \), if \( n \) is sufficiently large then \( |S_f(n) - x(f)| < \varepsilon \). We define \( FtR(f) \) to be this value \( x(f) \). In other words, we define

\[
FtR(f) = \sum_{i=0}^{\infty} f(i)10^{-i} \tag{2.3}
\]

which will be a number between 0 and 2.

To show that \( FtR \) is one to one, we need to show that \( FtR(f) \neq FtR(g) \) for every distinct \( f, g : \mathbb{N} \to \{0, 1\} \). Let \( f \neq g \) be such functions, and let \( k \) be the smallest number for which \( f(k) \neq g(k) \). We will show that \( |FtR(f) - FtR(g)| > 0.5 \cdot 10^{-k} \). This will complete the proof since in particular it implies \( FtR(f) \neq FtR(g) \).

Assume without loss of generality that \( f(k) = 0 \) and \( g(k) = 1 \) (otherwise switch the roles of \( f \) and \( g \)). Define \( S = \sum_{i=0}^{k-1} 10^{-i}f(i) = \sum_{i=0}^{k-1} 10^{-i}f(i) \) (the equality holds since \( f \) and \( g \) agree up to \( k \)). Now, since \( g(k) = 1 \),

\[
FtR(g) = \sum_{i=0}^{\infty} g(i)10^{-i} \geq \sum_{i=0}^{k} g(i)10^{-i} = S + 10^{-k} \, . \tag{2.4}
\]

On the other hand, since \( f(k) = 0 \) and \( f(k+1+j) \leq 1 \) for every \( j \geq 0 \),

\[
FtR(f) = \sum_{i=0}^{\infty} f(i)10^{-i} = S + \sum_{i=k+1}^{\infty} f(i)10^{-i} \leq S + 10^{-k-1} \sum_{j=0}^{\infty} 10^{-j} \, . \tag{2.5}
\]

Now \( \sum_{j=0}^{\infty} 10^{-j} \) is simply the number 1.11111... = 11/9, and hence we get that \( FtR(f) \leq S + 11/9 \cdot 10^{-k-1} \) while \( FtR(g) \geq S + 10^{-k} \) which means the difference between them is larger than \( 0.5 \cdot 10^{-k} \). ■
2.3 BEYOND NUMBERS

We can of course represent objects other than numbers as binary strings. Let us give a general definition for representation:

**Definition 2.5 — String representation.** Let \( \mathcal{O} \) be some set. A representation scheme for \( \mathcal{O} \) consists of a pair \((E, D)\) where \( E : \mathcal{O} \to \{0,1\}^* \) is a total one-to-one function, \( D : \{0,1\}^* \to \mathcal{O} \) is a (possibly partial) function, and such that \( D \) and \( E \) satisfy that \( D(E(o)) = o \) for every \( o \in \mathcal{O} \). \( E \) is known as the encoding function and \( D \) is known as the decoding function.

Note that the condition \( D(E(o)) = o \) for every \( o \in \mathcal{O} \) implies that \( D \) is onto (can you see why?). It turns out that to construct a representation scheme we only need to find an encoding function. That is, every one-to-one encoding function has a corresponding decoding function, as shown in the following lemma:

**Lemma 2.6** Suppose that \( E : \mathcal{O} \to \{0,1\}^* \) is one-to-one. Then there exists a function \( D : \{0,1\}^* \to \mathcal{O} \) such that \( D(E(o)) = o \) for every \( o \in \mathcal{O} \).

*Proof.* Let \( o_0 \) be some arbitrary element of \( O \). For every \( x \in \{0,1\}^* \), there exists either zero or a single \( o \in \mathcal{O} \) such that \( E(o) = x \) (otherwise \( E \) would not be one-to-one). We will define \( D(x) \) to equal \( o_0 \) in the first case and this single object \( o \) in the second case. By definition \( D(E(o)) = o \) for every \( o \in \mathcal{O} \). \( \square \)

Note that, while in general we allowed the decoding function to be partial. This proof shows that we can always obtain a total decoding function if we need to. This observation can sometimes be useful.

### 2.3.1 Finite representations

If \( \mathcal{O} \) is finite, then we can represent every object in \( o \) as a string of length at most some number \( n \). What is the value of \( n \)? Let us denote the set \( \{x \in \{0,1\}^* : |x| \leq n\} \) of strings of length at most \( n \) by \( \{0,1\}^\leq n \).

To obtain a representation of objects in \( \mathcal{O} \) as strings in \( \{0,1\}^\leq n \) we need to come up with a one-to-one function from the former set to the latter. We can do so, if and only if \( |\mathcal{O}| \leq 2^{n+1} - 1 \) as is implied by the following lemma:

**Lemma 2.7** For every two finite sets \( S, T \), there exists a one-to-one \( E : S \to T \) if and only if \( |S| \leq |T| \).

*Proof.* Let \( k = |S| \) and \( m = |T| \) and so write the elements of \( S \) and \( T \) as \( S = \{s_0, s_1, \ldots, s_{k-1}\} \) and \( T = \{t_0, t_1, \ldots, t_{m-1}\} \). We need to show that there is a one-to-one function \( E : S \to T \) iff \( k \leq m \). For
the “if” direction, if \( k \leq m \) we can simply define \( E(s_i) = t_i \) for every \( i \in [k] \). Clearly for \( i \neq j, t_i = E(s_i) \neq E(s_j) = t_j \), and hence this function is one-to-one. In the other direction, suppose that \( k > m \) and \( E : S \rightarrow T \) is some function. Then \( E \) cannot be one-to-one. Indeed, for \( i = 0, 1, \ldots, m - 1 \) let us “mark” the element \( t_j = E(s_i) \) in \( T \). If \( t_j \) was marked before, then we have found two objects in \( S \) mapping to the same element \( t_j \). Otherwise, since \( T \) has \( m \) elements, when we get to \( i = m - 1 \) we mark all the objects in \( T \). Hence, in this case \( E(s_m) \) must map to an element that was already marked before.\(^{14}\)

Now the size of \( \{0, 1\}^n \) is \( 2^n \), and the size of \( \{0, 1\}^{\leq n} \) is only slightly bigger: \( 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \) by the formula for a geometric series.

### 2.3.2 Prefix-free encoding

In our discussion of the representation of rational numbers, we used the “hack” of encoding the alphabet \( \{0, 1, \|\} \) to represent tuples of strings as a single string. This turns out to be a special case of the general paradigm of prefix-free encoding. An encoding function \( E : \mathcal{O} \rightarrow \{0, 1\}^* \) is prefix-free if there are no two objects \( o \neq o' \) such that the representation \( E(o) \) is a prefix of the representation \( E(o') \). The definition of prefix is as you would expect: a length \( n \) string \( x \) is a prefix of a length \( n' \geq n \) string \( x' \) if \( x_i = x'_i \) for every \( 1 \leq i \leq n \). Given a representation scheme for \( \mathcal{O} \) with a prefix-free encoding map, we can use simple concatenation to encode tuples of objects in \( \mathcal{O} \):

**Theorem 2.8 — Prefix-free implies tuple encoding.** Suppose that \((E, D)\) is a representation scheme for \( \mathcal{O} \) and \( E \) is prefix free. Then there exists a representation scheme \((E', D')\) for \( \mathcal{O}' \) such that for every \((o_0, \ldots, o_{k-1}) \in \mathcal{O}'\), \( E'(o_0, \ldots, o_{k-1}) = E(o_0)E(o_1) \ldots E(o_{k-1}) \).

**Theorem 2.8** is one of those statements that are a little hard to parse, but in fact are fairly straightforward to prove once you understand what they mean. Thus I highly recommend that you pause here, make sure you understand statement of the theorem, and try to prove it yourself before proceeding further.

---

\(^{14}\) This direction is sometimes known as the “Pigeon Hole Principle”: the principle that if you have a pigeon coop with \( m \) holes, and \( k > m \) pigeons, then there must be two pigeons in the same hole.
**Proof Idea:** The idea behind the proof is simple. Suppose that for example we want to decode a triple \( (x_0, x_1, x_2) \) from its representation \( x = E'(o_0, o_1, o_2) = E(o_0)E(o_1)E(o_2) \). We will do so by first finding the first prefix \( x_0 \) of \( x \) such is a representation of some object. Then we will decode this object, remove \( x_0 \) from \( x \) to obtain a new string \( x' \), and continue onwards to find the first prefix \( x_1 \) of \( x' \) and so on and so forth (see Exercise 2.5). The prefix-freeness property of \( E \) will ensure that \( x_0 \) will in fact be \( E(o_0) \), \( x_1 \) will be \( E(o_1) \) etc.

\[ E'(o_0, \ldots, o_{k-1}) = E'(o'_0, \ldots, o'_{k'-1}) \] (2.6)

We denote \( x_i = E(o_i) \) and \( x'_i = E(o'_i) \). By our assumption and the definition of \( E' \), \( x_0 x_1 \cdots x_{k-1} = x'_0 x'_1 \cdots x'_{k'-1} \). Without loss of generality we can assume \( k' \leq k \). Let \( i \) be the largest number such that \( x_j = x'_j \) for all \( j < i \). (If \( x'_0 \neq x_0 \) then \( i = 0 \); if \( x_j = x'_j \) for all \( j < k \) then we let \( i = k \); note that the fact that the concatenation of \( x_0, \ldots, x_{k-1} \) is equal to the concatenation of \( x'_0, \ldots, x'_{k'-1} \) does not mean the individual components have to agree.)

Since \( x_j = x'_j \) for all \( j < i \), the strings \( x_i \cdots x_{k-1} \) and \( x'_i \cdots x'_{k'-1} \) are identical, and we denote this string by \( s \). If \( i < k \) then both \( x_i \) and \( x'_i \) are prefixes of \( s \) which means that one of them is a prefix of the other, since by the choice of \( i \), \( x_i \neq x'_i \) we get that both of them are valid representation of distinct objects which contradicts prefix-freeness.

If \( i = k \) then the string \( s \) must be empty, but this would mean that \( i = k' \) as well, which means that \( x_i = x'_i \) for all \( i \), which means that the original tuples of objects must have been the same.

**2.3.3 Making representations prefix-free**

Some natural representations are prefix-free. For example, every fixed output length representation (i.e., one-to-one function \( E : \mathcal{O} \rightarrow \{0,1\}^n \)) is automatically prefix-free, since a string \( x \) can only be a prefix of an equal-length \( x' \) if \( x \) and \( x' \) are identical. Moreover, the approach we used for representing rational numbers can be used to show the following:

**Lemma 2.9** Let \( E : \mathcal{O} \rightarrow \{0,1\}^* \) be a one-to-one function. Then there is a one-to-one prefix-free encoding \( \overline{E} \) such that \( |\overline{E}(o)| \leq 2|E(o)| + 2 \) for every \( o \in \mathcal{O} \).
For the sake of completeness, we will include the proof below, but it is a good idea for you to pause here and try to prove it yourself, using the same technique we used for representing rational numbers.

Proof of Lemma 2.9. Define the function \( PF : \{0, 1\}^* \rightarrow \{0, 1\}^* \) as follows \( PF(x) = x_0x_1x_2x_3...x_{n-1}x_{n-2}101 \) for every \( x \in \{0, 1\}^* \). If \( E : \mathcal{O} \rightarrow \{0, 1\}^* \) is the (potentially not prefix-free) representation for \( \mathcal{O} \), then we transform it into a prefix-free representation \( \overline{E} : \mathcal{O} \rightarrow \{0, 1\}^* \) by defining \( \overline{E}(o) = PF(E(o)) \).

To prove the lemma we need to show that (1) \( \overline{E} \) is one-to-one and (2) \( \overline{E} \) is prefix-free. In fact (2) implies (1), since if \( \overline{E}(o) \) is never a prefix of \( \overline{E}(o') \) for every \( o \neq o' \) then in particular \( \overline{E} \) is one-to-one. Now suppose, toward a contradiction, that there are \( o \neq o' \) in \( \mathcal{O} \) such that \( \overline{E}(o) \) is a prefix of \( \overline{E}(o') \). (That is, if \( y = \overline{E}(o) \) and \( y' = \overline{E}(o') \), then \( y_j = y'_j \) for every \( j < |y| \).)

Define \( x = E(o) \) and \( x' = E(o') \). Note that since \( E \) is one-to-one, \( x \neq x' \). (Recall that two strings \( x, x' \) are distinct if they either differ in length or have at least one distinct coordinate.) Under our assumption, \( |PF(x)| \leq |PF(x')| \), and since by construction \( |PF(x)| = 2|x| + 2 \), it follows that \( |x| \leq |x'| \). If \( |x| = |x'| \) then, since \( x \neq x' \), there must be a coordinate \( i \in \{0, ..., |x|-1\} \) such that \( x_i \neq x'_i \). But since \( PF(x)_{2i} = x_i \), we get that \( PF(x)_{2i} \neq PF(x')_{2i} \) and hence \( \overline{E}(o) = PF(x) \) is not a prefix of \( \overline{E}(o') = PF(x') \). Otherwise (if \( |x| \neq |x'| \)) then it must be that \( |x| < |x'| \), and hence if \( n = |x| \), then \( PF(x)_{2n} = 0 \) and \( PF(x)_{2n+1} = 1 \). But since \( n < |x'| \), \( PF(x')_{2n}, PF(x')_{2n+1} \) is equal to either 00 or 11, and in any case we get that \( \overline{E}(o) = PF(x) \) is not a prefix of \( \overline{E}(o') = PF(x') \). ■

In fact, we can even obtain a more efficient transformation where \( |E'(o)| \leq |o| + O(\log |o|) \). We leave proving this as an exercise (see Exercise 2.6).

2.3.4 “Proof by Python” (optional)
The proofs of Theorem 2.8 and Lemma 2.9 are constructive in the sense that they give us:

- a way to transform the encoding and decoding functions of any representation of an object \( O \) to a encoding and decoding functions that are prefix free;
- a way to extend prefix-free encoding and decoding of single objects to encoding and decoding of lists of objects by concatenation.
Specifically, we could transform any pair of Python functions encode and decode to functions pfencode and pfdecode that correspond to a prefix-free encoding and decoding. Similarly, given pfencode and pfdecode for single objects, we can extend them to encoding of lists. Let us show how this works for the case of the int2bits and bits2int functions we defined above.

```python
# takes functions encode and decode mapping # objects to lists of bits and vice versa, # and returns functions pfencode and pfdecode that # maps objects to lists of bits and vice versa # in a prefix-free way. # Also returns a function pfvalid that says # whether a list is a valid encoding

def prefixfree(encode, decode):
    def pfencode(o):
        L = encode(o)
        return [L[i//2] for i in range(2+2*len(L))]+[0,1]
    def pfdecode(L):
        return decode([L[j] for j in range(0,2*len(L)-2,2)])
    def pfvalid(L):
        return (len(L) % 2 == 0) and L[-2:]==[0,1]

    return pfencode, pfdecode, pfvalid

pfint2bits, pfbits2int, pfvalidint =
    prefixfree(int2bits,bits2int)

print(int2bits(23))
# [1, 1, 1, 0, 1]
print(pfint2bits(23))
# [1, 1, 1, 1, 1, 1, 0, 0, 0, 1]

print(pfbits2int(pfint2bits(23)))
# 23

print(pfvalidint(pfint2bits(23)))
# true

print(pfvalidint([1,1,1,1]))
# false
```
Note that Python function `prefixfree` above takes two Python functions as input and outputs three Python functions as output. You don’t have to know Python in this course, but you do need to get comfortable with the idea of functions as mathematical objects in their own right, that can be used as inputs and outputs of other functions.

```
# Takes functions pfencode, pfdecode and pfvalid,
# and returns functions encodelists, decodelists
# that can encode and decode
# lists of the objects respectively
def represlists(pfencode,pfdecode,pfvalid):
    def encodelist(L):
        """Gets list of objects, encodes it as list of
        bits""
        return [bit for obj in L for bit in pfencode(obj)]
    def decodelist(S):
        """Gets lists of bits, returns lists of
        objects""
        i=0; j=1; res = []
        while j<=len(S):
            if pfvalid(S[i:j]):
                res += [pfdecode(S[i:j])]
                i=j
            j+= 1
        return res
    return encodelist,decodelist

intlist2bits, bits2intlist = represlists(pfint2bits,pfbits2int,pfvalidint)

print(intlist2bits([[12,8]]))
# [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0]
print(bits2intlist(intlist2bits([[12,8]])))
# [12, 8]
```

When it’s not too awkward, I use the term “Python function” or “subroutine” to distinguish between such snippets of python programs and mathematical functions. However, in comments in python source I use “functions” to denote python functions, just as I use “integers” to denote python int objects.
2.3.5 Representing letters and text

We can represent a letter or symbol by a string, and then if this representation is prefix-free, we can represent a sequence of symbols by simply concatenating the representation of each symbol. One such representation is the ASCII that represents 128 letters and symbols as strings of 7 bits. Since it is a fixed-length representation it is automatically prefix-free (can you see why?). Unicode is a representation of (at the time of this writing) about 128,000 symbols into numbers (known as code points) between 0 and 1,114,111. There are several types of prefix-free representations of the code points, a popular one being UTF-8 that encodes every codepoint into a string of length between 8 and 32.

Example 2.10 — Representing objects in C (optional). We can use programming languages to probe how our computing environment represents various values. This is easier to do in “unsafe” programming languages such as C that allow direct access to the memory.

Using a simple C program we have produced the following representations of various values. One can see that for integers, multiplying by 2 corresponds to a “left shift” inside each byte. In contrast, for floating point numbers, multiplying by two corresponds to adding one to the exponent part of the representation. A negative number is represented using the two’s complement approach. Strings are represented in a prefix-free form by ensuring that a zero byte is at their end.

```
int 2 : 00000010 00000000 00000000
\ 00000000
int 4 : 00000100 00000000 00000000
\ 00000000
int 513 : 00000001 00000010 00000000
\ 00000000
long 513 : 00000001 00000010 00000000
\ 00000000 00000000 00000000 00000000
int -1 : 11111111 11111111 11111111
\ 11111111
int -2 : 11111110 11111111 11111111
\ 11111111
string Hello: 01001000 01100101 01101100
\ 01101100 01101111 00000000
string abcd : 01100001 01100010 01100011
\ 01100100 00000000
```
If you are curious, the code for this program (which you can run here) is the following:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *bytes(void *p, int n){
    int i;
    int j;
    char *a = (char *) p;
    char *s = malloc(9*n+2);
    s[9*n] = '\n';
    s[9*n+1] = 0;
    j = 0;
    for(i=0;i< n*8;i++){
        s[j] = a[i/8] & (128 >> (i % 8)) ? '1' : '0';
        if (i % 8 == 7) { s[++j] = ' '; } 
        ++j;
    }
    return s;
}

void printint(int a) {
    printf("%-8s %-5d: %s", "int", a, 
   .bytes(&a,sizeof(int)));
}

void printlong(long a) {
    printf("%-8s %-5d: %s", "long", a, 
   .bytes(&a,sizeof(long)));
}
```
void printstring(char *s) {
    printf("%-8s %-5s: %s", "string", s,
            bytes(s,strlen(s)+1));
}

void printfloat(float f) {
    printf("%-8s %-5.1f: %s", "float", f,
            bytes(&f,sizeof(float)));
}

void printdouble(double f) {
    printf("%-8s %-5.1f: %s", "double", f,
            bytes(&f,sizeof(double)));
}

int main(void) {

    printf(2);
    printf(4);
    printf(513);
    printf(513);

    printf(-1);
    printf(-2);

    printf("Hello");
    printf("abcd");

    printf(33);
    printf(66);
    printf(132);
    printf(132);

    return 0;
}
2.3.6 Representing vectors, matrices, images

Once we can represent numbers, and lists of numbers, then we can obviously represent vectors (which are just lists of numbers). Similarly, we can represent lists of lists, and thus in particular can represent matrices. To represent an image, we can represent the color at each pixel by a list of three numbers corresponding to the intensity of Red, Green and Blue.\textsuperscript{16} Thus an image of $n$ pixels would be represented by a list of $n$ such length-three lists. A video can be represented as a list of images.\textsuperscript{17}

2.3.7 Representing graphs

A graph on $n$ vertices can be represented as an $n \times n$ adjacency matrix whose $(i, j)^{th}$ entry is equal to 1 if the edge $(i, j)$ is present and is equal to 0 otherwise. That is, we can represent an $n$ vertex directed graph $G = (V, E)$ as a string $A \in \{0, 1\}^n$ such that $A_{i,j} = 1$ iff the edge $i \rightarrow j \in E$. We can transform an undirected graph to a directed graph by replacing every edge $\{i, j\}$ with both edges $i \rightarrow j$ and $j \rightarrow i$.

Another representation for graphs is the adjacency list representation. That is, we identify the vertex set $V$ of a graph with the set $[n]$ where $n = |V|$, and represent the graph $G = (V, E)$ a a list of $n$ lists, where the $i$-th list consists of the out-neighbors of vertex $i$. The difference between these representations can be important for some applications, though for us would typically be immaterial.

![Figure 2.6: Representing the graph $G = \{(0, 1, 2, 3, 4), \{(1, 0), (4, 0), (1, 4), (4, 1), (2, 1), (3, 2), (4, 3)\}$ in the adjacency matrix and adjacency list representations.](image)

Once again, we can also define these encoding and decoding functions in python:

```python
from graphviz import Graph

# get n by n matrix (as list of n lists)
# return graph corresponding to it
def matrix2graph(M):
```

\textsuperscript{16} We can restrict to three basic colors since (most) humans only have three types of cones in their retinas. We would have needed 16 basic colors to represent colors visible to the Mantis Shrimp.

\textsuperscript{17} Of course these representations are rather wasteful and much more compact representations are typically used for images and videos, though this will not be our concern in this course.
G = Graph(); n = len(M)
for i in range(n):
    G.node(str(i))  # add vertex i
for j in range(n):
    G.node(str(j))
    if M[i][j]: G.edge(str(i), str(j))
    # if M[i][j] is nonzero then add edge
    # between i and j
return G

text12.png

2.3.8 Representing lists
If we have a way of represent objects from a set \( \mathcal{O} \) as binary strings, then we can represent lists of these objects by applying a prefix-free transformation. Moreover, we can use a trick similar to the above to handle nested lists. The idea is that if we have some representation \( E : \mathcal{O} \rightarrow \{0, 1\}^* \), then we can represent nested lists of items from \( \mathcal{O} \) using strings over the five element alphabet \( \Sigma = \{0, 1, [ , ] , , \} \). For example, if \( o_1 \) is represented by 0011, \( o_2 \) is represented by 10011, and \( o_3 \) is represented by 00111, then we can represent the nested list \( (o_1, (o_2, o_3)) \) as the string "[0011, [1011, 00111]]" over the alphabet \( \Sigma \). By encoding every element of \( \Sigma \) itself as a three-bit string, we can transform any representation for objects \( \mathcal{O} \) into a representation that allows to represent (potentially nested) lists of these objects.

2.3.9 Notation
We will typically identify an object with its representation as a string. For example, if \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \) is some function that maps
strings to strings and $x$ is an integer, we might make statements such as “$F(x) + 1$ is prime” to mean that if we represent $x$ as a string $x$ and let $y = F(x)$, then the integer $y$ represented by the string $y$ satisfies that $y + 1$ is prime. (You can see how this convention of identifying objects with their representation can save us a lot of cumbersome formalism.) Similarly, if $x, y$ are some objects and $F$ is a function that takes strings as inputs, then by $F(x, y)$ we will mean the result of applying $F$ to the representation of the order pair $(x, y)$. We will use the same notation to invoke functions on $k$-tuples of objects for every $k$.

This convention of identifying an object with its representation as a string is one that we humans follow all the time. For example, when people say a statement such as “17 is a prime number”, what they really mean is that the integer whose decimal representation is the string “17”, is prime.

### 2.4 DEFINING COMPUTATIONAL TASKS

Abstractly, a *computational process* is some process that takes an input which is a string of bits, and produces an output which is a string of bits. This transformation of input to output can be done using a modern computer, a person following instructions, the evolution of some natural system, or any other means.

In future chapters, we will turn to mathematically defining computational process, but, as we discussed above for now we want to focus on computational tasks; i.e., focus on the specification and not the implementation. Again, at an abstract level, a computational task can specify any relation that the output needs to have with the input. But for most of this course, we will focus on the simplest and most common task of computing a function. Here are some examples:

- Given (a representation) of two integers $x, y$, compute the product $x \times y$. Using our representation above, this corresponds to computing a function from $\{0, 1\}^* \to \{0, 1\}^*$. We’ve seen that there is more
than one way to solve this computational task, and in fact, we still don’t know the best algorithm for this problem.

• Given (a representation of) an integer $z$, compute its factorization; i.e., the list of primes $p_1 \leq \cdots \leq p_k$ such that $z = p_1 \cdots p_k$. This again corresponds to computing a function from $\{0,1\}^*$ to $\{0,1\}^*$. The gaps in our knowledge of the complexity of this problem are even longer.

• Given (a representation of) a graph $G$ and two vertices $s$ and $t$, compute the length of the shortest path in $G$ between $s$ and $t$, or do the same for the longest path (with no repeated vertices) between $s$ and $t$. Both these tasks correspond to computing a function from $\{0,1\}^*$ to $\{0,1\}^*$, though it turns out that there is a huge difference in their computational difficulty.

• Given the code of a Python program, determine whether there is an input that would force it into an infinite loop. This corresponds to computing a partial function from $\{0,1\}^*$ to $\{0,1\}$; though it is easy to make it into a total function by mapping every string into the trivial Python program that stops without doing anything. We will see that we do understand the computational status of this problem, but the answer is quite surprising.

• Given (a representation of) an image $I$, decide if $I$ is a photo of a cat or a dog. This correspond to computing some (partial) function from $\{0,1\}^*$ to $\{0,1\}$.

For every particular function $F$, there can be several possible algorithms to compute $F$. We will be interested in questions such as:

• For a given function $F$, can it be the case that there is no algorithm to compute $F$?

---

**R** **Boolean functions and languages** An important special case of computational tasks corresponds to computing Boolean functions, whose output is a single bit $\{0,1\}$. Computing such functions corresponds to answering a YES/NO question, and hence this task is also known as a decision problem. Given any function $F : \{0,1\}^* \to \{0,1\}$ and $x \in \{0,1\}^*$, the task of computing $F(x)$ corresponds to the task of deciding whether or not $x \in L$ where $L = \{x : F(x) = 1\}$ is known as the language that corresponds to the function $F$.  

---

18 The language terminology is due to historical connections between the theory of computation and formal linguistics as developed by Noam Chomsky.
Figure 2.8: A subset $L \subseteq \{0, 1\}^*$ can be identified with the function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ such that $F(x) = 1$ if $x \in L$ and $F(x) = 0$ if $x \notin L$. Functions with a single bit of output are called Boolean functions, while subsets of strings are called languages. The above shows that the two are essentially the same object, and we can identify the task of deciding membership in $L$ (known as deciding a language in the literature) with the task of computing the function $F$.

- If there is an algorithm, what is the best one? Could it be that $F$ is “effectively uncomputable” in the sense that every algorithm for computing $F$ requires a prohibitively large amount of resources?

- If we can’t answer this question, can we show equivalence between different functions $F$ and $F'$ in the sense that either they are both easy (i.e., have fast algorithms) or they are both hard?

- Can a function being hard to compute ever be a good thing? Can we use it for applications in areas such as cryptography?

In order to do that, we will need to mathematically define the notion of an algorithm, which is what we’ll do in Chapter 3.

2.4.1 Distinguish functions from programs

You should always watch out for potential confusions between specification and implementation or equivalently between mathematical functions and algorithms/programs. It does not help that programming languages (my favorite Python included) use the term “functions” to denote (parts of) programs. This confusion also stems from thousands of years of mathematical history, where people typically defined functions by means of a way to compute them.

For example, consider the multiplication function on natural numbers. This is the function $MULT : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ that maps a pair $(x, y)$
of natural numbers to the number $x \cdot y$. As we mentioned, it can be implemented in more than one way:

```python
def mult1(x, y):
    res = 0
    while y > 0:
        res += x
        y -= 1
    return res

def mult2(x, y):
    a = int2bits(x)
    b = int2bits(y)
    res = [0] * (len(a) + len(b))
    for i in range(len(a)):
        for j in range(len(b)):
            res[i + j] += a[i] * b[j]
    return bits2int(res)
```

Both `mult1` and `mult2` produce the same output given the same pair of inputs. (Though `mult1` will take far longer to do so when the numbers become large.) Hence, even though these are two different programs, they compute the same mathematical function. This distinction between a program or algorithm $A$, and the function $F$ that $A$ computes will be absolutely crucial for us in this course (see also Fig. 2.9).

### 2.4.2 Advanced note: beyond computing functions

Functions capture quite a lot of computational tasks, but one can consider more general settings as well. For starters, we can and will talk about partial functions, that are not defined on all inputs. When computing a partial function, we only need to worry about the inputs on which the function is defined. Another way to say it is that we can design an algorithm for a partial function $F$ under the assumption that someone “promised” us that all inputs $x$ would be such that $F(x)$ is defined (as otherwise we don’t care about the result). Hence such
A function is a mapping of inputs to outputs. A program is a set of instructions of how to obtain an output given an input. A program computes a function, but it is not the same as a function, popular programming language terminology notwithstanding.

Another generalization is to consider relations that may have more than one possible admissible output. For example, consider the task of finding any solution for a given set of equations. A relation $R$ maps a string $x \in \{0, 1\}^*$ into a set of strings $R(x)$ (for example, $x$ might describe a set of equations, in which case $R(x)$ would correspond to the set of all solutions to $x$). We can also identify a relation $R$ with the set of pairs of strings $(x, y)$ where $y \in R(x)$. A computational process solves a relation if for every $x \in \{0, 1\}^*$, it outputs some string $y \in R(x)$.

Later on in this course we will consider even more general tasks, including interactive tasks, such as finding good strategy in a game, tasks defined using probabilistic notions, and others. However, for much of this course we will focus on the task of computing a function, and often even a Boolean function, that has only a single bit of output. It turns out that a great deal of the theory of computation can be studied in the context of this task, and the insights learned are applicable in the more general settings.

**Lecture Recap**

- We can represent essentially every object we want to compute on using binary strings.
- A representation scheme for a set of objects $\mathcal{O}$ is a one-to-one map from $\mathcal{O}$ to $\{0, 1\}^*$.
- A basic computational task is the task of computing a function $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$. This encompasses not just arithmetical computations such as multiplication, factoring, etc. but a great many other tasks arising in areas as diverse as
scientific computing, artificial intelligence, image processing, data mining and many many more.

• We will study the question of finding (or at least giving bounds on) what is the best algorithm for computing $F$ for various interesting functions $F$.

### 2.5 EXERCISES

#### R Disclaimer

Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

#### Exercise 2.1 Which one of these objects can be represented by a binary string?

1. An integer $x$
2. An undirected graph $G$.
3. A directed graph $H$
4. All of the above.

#### Exercise 2.2 — Multiplying in different representation. Recall that the grade-school algorithm for multiplying two numbers requires $O(n^2)$ operations. Suppose that instead of using decimal representation, we use one of the following representations $R(x)$ to represent a number $x$ between $0$ and $10^n − 1$. For which one of these representations you can still multiply the numbers in $O(n^2)$ operations?

1. The standard binary representation: $B(x) = (x_0, \ldots, x_k)$ where $x = \sum_{i=0}^{k} x_i 2^i$ and $k$ is the largest number s.t. $x \geq 2^k$.
2. The reverse binary representation: $B(x) = (x_k, \ldots, x_0)$ where $x_i$ is defined as above for $i = 0, \ldots, k − 1$.
3. Binary coded decimal representation: $B(x) = (y_0, \ldots, y_{n-1})$ where $y_i \in \{0, 1\}^4$ represents the $i^{th}$ decimal digit of $x$ mapping 0 to 0000, 1 to 0001, 2 to 0010, etc. (i.e. 9 maps to 1001)
4. All of the above.
Exercise 2.3 Suppose that $R: \mathbb{N} \rightarrow \{0, 1\}^*$ corresponds to representing a number $x$ as a string of $x$ 1’s, (e.g., $R(4) = 1111$, $R(7) = 1111111$, etc.). If $x, y$ are numbers between 0 and $10^n - 1$, can we still multiply $x$ and $y$ using $O(n^2)$ operations if we are given them in the representation $R(\cdot)$?

Exercise 2.4 Recall that if $F$ is a one-to-one and onto function mapping elements of a finite set $U$ into a finite set $V$ then the sizes of $U$ and $V$ are the same. Let $B: \mathbb{N} \rightarrow \{0, 1\}^*$ be the function such that for every $x \in \mathbb{N}$, $B(x)$ is the binary representation of $x$.

1. Prove that $x < 2^k$ if and only if $|B(x)| \leq k$.

2. Use a. to compute the size of the set $\{y \in \{0, 1\}^* : |y| \leq k\}$ where $|y|$ denotes the length of the string $y$.

3. Use a. and b. to prove that $2^k - 1 = 1 + 2 + 4 + \cdots + 2^{k-1}$.

Exercise 2.5 — Prefix-free encoding of tuples. Suppose that $F: \mathbb{N} \rightarrow \{0, 1\}^*$ is a one-to-one function that is prefix-free in the sense that there is no $a \neq b$ s.t. $F(a)$ is a prefix of $F(b)$.

1. Prove that $F_2 : \mathbb{N} \times \mathbb{N} \rightarrow \{0, 1\}^*$, defined as $F_2(a, b) = F(a)F(b)$ (i.e., the concatenation of $F(a)$ and $F(b)$) is a one-to-one function.

2. Prove that $F_\ast : \mathbb{N}^\ast \rightarrow \{0, 1\}^*$ defined as $F_\ast(a_1, \ldots, a_k) = F(a_1) \cdots F(a_k)$ is a one-to-one function, where $\mathbb{N}^\ast$ denotes the set of all finite-length lists of natural numbers.

Exercise 2.6 — More efficient prefix-free transformation. Suppose that $F: \mathbb{N} \rightarrow \{0, 1\}^*$ is some (not necessarily prefix-free) representation of the objects in the set $O$, and $G: \mathbb{N} \rightarrow \{0, 1\}^*$ is a prefix-free representation of the natural numbers. Define $F'(o) = G(|F(o)|)F(o)$ (i.e., the concatenation of the representation of the length $F(o)$ and $F(o)$).

1. Prove that $F'$ is a prefix-free representation of $O$.

2. Show that we can transform any representation to a prefix-free one by a modification that takes a $k$ bit string into a string of length at most $k + O(\log k)$.

3. Show that we can transform any representation to a prefix-free one by a modification that takes a $k$ bit string into a string of length at most $k + \log k + O(\log \log k)$.\(^19\)

\(^19\) Hint: Think recursively how to represent the length of the string.
**Exercise 2.7 — Kraft’s Inequality.** Suppose that $S \subseteq \{0,1\}^n$ is some finite prefix-free set.

1. For every $k \leq n$ and length-$k$ string $x \in S$, let $L(x) \subseteq \{0,1\}^n$ denote all the length-$n$ strings whose first $k$ bits are $x_0\ldots x_{k-1}$. Prove that (1) $|L(x)| = 2^{n-k|x|}$ and (2) if $x \neq x'$ then $L(x)$ is disjoint from $L(x')$.

2. Prove that $\sum_{x \in S} 2^{-|x|} \leq 1$.

3. Prove that there is no prefix-free encoding of strings with less than logarithmic overhead. That is, prove that there is no function $PF : \{0,1\}^* \to \{0,1\}^*$ s.t. $|PF(x)| \leq |x| + 0.9 \log |x|$ for every $x \in \{0,1\}^*$ and such that the set $\{PF(x) : x \in \{0,1\}^*\}$ is prefix-free. The factor 0.9 is arbitrary; all that matters is that it is less than 1.

**Exercise 2.8 — Composition of one-to-one functions.** Prove that for every two one-to-one functions $F : S \to T$ and $G : T \to U$, the function $H : S \to U$ defined as $H(x) = G(F(x))$ is one to one.

**Exercise 2.9 — Natural numbers and strings.** 1. We have shown that the natural numbers can be represented as strings. Prove that the other direction holds as well: that there is a one-to-one map $StN : \{0,1\}^* \to \mathbb{N}$. ($StN$ stands for “strings to numbers”.)

2. Recall that Cantor proved that there is no one-to-one map $RtN : \mathbb{R} \to \mathbb{N}$. Show that Cantor’s result implies Theorem 2.2.

**Exercise 2.10 — Map lists of integers to a number.** Recall that for every set $S$, the set $S^*$ is defined as the set of all finite sequences of members of $S$ (i.e., $S^* = \{(x_0\ldots x_{n-1}) \mid n \in \mathbb{N}, \forall i \in [n] x_i \in S\}$).

Prove that there is a one-one-map from $\mathbb{Z}^*$ to $\mathbb{N}$ where $\mathbb{Z}$ is the set of $\{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}$ of all integers.

### 2.6 BIBLIOGRAPHICAL NOTES

The idea that we should separate the definition or specification of a function from its implementation or computation might seem “obvious”, but it took some time for mathematicians to arrive at this viewpoint. Historically, a function $F$ was identified by rules or formulas showing how to derive the output from the input. As we discuss in greater depth in Chapter 8, in the 1800’s this somewhat informal notion of a function started “breaking at the seams” and eventually mathematicians arrived at the more rigorous definition of a function as an arbitrary assignment of input to outputs. While many functions may
be described (or computed) by one or more formulas, today we do not consider that to be an essential property of functions, and also allow functions that do not correspond to any “nice” formula.

Gromov and Pomerantz’s quotes are lifted from Doron Zeilberger’s page.

2.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

- **Succinct** data structures. These are representations that map objects from some set \( \mathcal{O} \) into strings of length not much larger than the minimum of \( \log_2 |\mathcal{O}| \) but still enable fast access to certain queries, see for example this paper.

- We’ve mentioned that all representations of the real numbers are inherently approximate. Thus an important endeavor is to understand what guarantees we can offer on the approximation quality of the output of an algorithm, as a function of the approximation quality of the inputs. This is known as the question of numerical stability.

- **The linear algebraic view of graphs**: The adjacency matrix representation of graphs is not merely a convenient way to map a graph into a binary string, but it turns out that many natural notions and operations on matrices are useful for graphs as well. (For example, Google’s PageRank algorithm relies on this viewpoint.) The notes of this course are an excellent source for this area, known as spectral graph theory. We might discuss this view much later in this course when we talk about random walks.
FINITE COMPUTATION
3

Defining computation

“there is no reason why mental as well as bodily labor should not be economized by the aid of machinery”, Charles Babbage, 1852

“If, unwarned by my example, any man shall undertake and shall succeed in constructing an engine embodying in itself the whole of the executive department of mathematical analysis upon different principles or by simpler mechanical means, I have no fear of leaving my reputation in his charge, for he alone will be fully able to appreciate the nature of my efforts and the value of their results.”, Charles Babbage, 1864

“To understand a program you must become both the machine and the program.”, Alan Perlis, 1982

People have been computing for thousands of years, with aids that include not just pen and paper, but also abacus, slide rulers, various mechanical devices, and modern electronic computers. A priori, the notion of computation seems to be tied to the particular mechanism that you use. You might think that the “best” algorithm for multiplying numbers will differ if you implement it in Python on a modern laptop than if you use pen and paper. However, as we saw in the introduction (Chapter 0), an algorithm that is asymptotically better would eventually beat a worse one regardless of the underlying technology. This gives us hope for a technology independent way of defining computation, which is what we will do in this chapter.
Figure 3.1: Calculating wheels by Charles Babbage. Image taken from the Mark I 'operating manual'.

Figure 3.2: A 1944 Popular Mechanics article on the Harvard Mark I computer.
3.1 DEFINING COMPUTATION

The name “algorithm” is derived from the Latin transliteration of Muhammad ibn Musa al-Khwarizmi’s name. Al-Khwarizmi was a Persian scholar during the 9th century whose books introduced the western world to the decimal positional numeral system, as well as the solutions of linear and quadratic equations (see Fig. 3.3). However Al-Khwarizmi’s descriptions of algorithms were rather informal by today’s standards. Rather than use “variables” such as $x, y$, he used concrete numbers such as 10 and 39, and trusted the reader to be able to extrapolate from these examples.\(^1\)

Here is how al-Khwarizmi described the algorithm for solving an equation of the form $x^2 + bx = c^2$

\[\text{[How to solve an equation of the form ] “roots and squares are equal to numbers”: For instance “one square, and ten roots of the same, amount to thirty-nine dirhems” that is to say, what must be the square which, when increased by ten of its own root, amounts to thirty-nine? The solution is this: you halve the number of the roots, which in the present instance yields five. This you multiply by itself; the product is twenty-five. Add this to thirty-nine the sum is sixty-four. Now take the root of this, which is eight, and subtract from it half the number of roots, which is five; the remainder is three. This is the root of the square which you sought for; the square itself is nine.}\]

\(^1\) Indeed, extrapolation from examples is still the way most of us first learn algorithms such as addition and multiplication, see Fig. 3.4.

\(^2\) Translation from “The Algebra of Ben-Musa”, Fredric Rosen, 1831.
For the purposes of this course, we will need a much more precise way to describe algorithms. Fortunately (or is it unfortunately?), at least at the moment, computers lag far behind school-age children in learning from examples. Hence in the 20th century people have come up with exact formalisms for describing algorithms, namely programming languages. Here is al-Khwarizmi’s quadratic equation solving algorithm described in the Python programming language:\(^3\), this is not a programming course, and it is absolutely fine if you don’t know Python. Still the code below should be fairly self-explanatory.]

```python
from math import sqrt

# Pythonspeak to enable use of the sqrt function to compute square roots.

def solve_eq(b, c):
    # return solution of x^2 + bx = c following Al
    # Khwarizmi’s instructions
    # Al Khwarizmi demonstrates this for the case
    # b=10 and c= 39

    val1 = b/2.0  # "halve the number of the roots"
    val2 = val1*val1  # "this you multiply by itself"
    val3 = val2 + c  # "Add this to thirty-nine"
    val4 = sqrt(val3)  # "take the root of this"
```

\(^3\) As mentioned in Remark 2.1.
We can define algorithms informally as follows:

**Informal definition of an algorithm:** An Algorithm is a set of instructions of how to compute an output from an input by following a sequence of "elementary steps". An algorithm A computes a function F if for every input x, if we follow the instruction of A on the input x, we obtain the output F(x).

In this chapter we will use an ultra-simple “programming language” to give a formal (that is, precise) definition of algorithms. (In fact, our programming language will be so simple that it is hardly worthy of this name.) However, it will take us some time to get there. We will start by discussing what are “elementary operations” and also how do we map a description of an algorithm into an actual physical process that produces an output from an input in the real world.

### 3.1.1 Boolean formulas with AND, OR, and NOT.

An algorithm breaks down a complex calculation into a series of simpler steps. These steps can be executed by:

- Writing down symbols on a piece of paper
- Modifying the current flowing on electrical wires.
- Binding a protein to a strand of DNA
- Response to a stimulus by a member of a collection (e.g., a bee in a colony, a trader in a market).

To formally define algorithms, let us try to “err on the side of simplicity” and model our “basic steps” as truly minimal. For example, here are some very simple functions:

- **OR**: \( \{0, 1\}^2 \rightarrow \{0, 1\} \) defined as

\[
OR(a, b) = \begin{cases} 
0 & a = b = 0 \\
1 & \text{otherwise}
\end{cases} \quad (3.1)
\]
• **AND** : \{0, 1\}^2 \rightarrow \{0, 1\} defined as

\[
\text{AND}(a, b) = \begin{cases} 
1 & a = b = 1 \\
0 & \text{otherwise} 
\end{cases}
\] (3.2)

• **NOT** : \{0, 1\} \rightarrow \{0, 1\} defined as \text{NOT}(a) = 1 - a.

The **AND**, **OR** and **NOT** functions are the basic logical operators that are used in logic and many computer system. Each one of them takes either one or two single bits as input, and produces a single bit as output. Clearly, it cannot get much more basic than these. However, the power of computation comes from *composing* simple building blocks together.

Here is an example. Consider the function **MAJ** : \{0, 1\}^3 \rightarrow \{0, 1\} that is defined as follows:

\[
\text{MAJ}(x) = \begin{cases} 
1 & x_0 + x_1 + x_2 \geq 2 \\
0 & \text{otherwise} 
\end{cases}.
\] (3.3)

That is, for every \(x \in \{0, 1\}^3\), \(\text{MAJ}(x) = 1\) if and only if the majority (i.e., at least two out of the three) of \(x\)'s coordinates are equal to 1. Can you come up with a formula involving **AND**, **OR** and **NOT** to compute **MAJ**?

It is useful for you to pause at this point and work out the formula for yourself. As a hint, although it is needed to compute some functions, you will not need to use the **NOT** operator to compute **MAJ**.

Let us first try to rephrase \(\text{MAJ}(x)\) in words: “\(\text{MAJ}(x) = 1\) if and only if there exists some pair of distinct coordinates \(i, j\) such that both \(x_i\) and \(x_j\) are equal to 1.” In other words it means that \(\text{MAJ}(x) = 1\) if either both \(x_0 = 1\) and \(x_1 = 1\), or both \(x_1 = 1\) and \(x_2 = 1\), or both \(x_0 = 1\) and \(x_2 = 1\). Since the **OR** of three conditions \(c_0, c_1, c_2\) can be written as \(\text{OR}(c_0, \text{OR}(c_1, c_2))\), we can now translate this into a formula as follows:

\[
\text{MAJ}(x_0, x_1, x_2) = \text{OR} \left( \text{AND}(x_0, x_1), \text{OR}(\text{AND}(x_1, x_2), \text{AND}(x_0, x_2)) \right).
\] (3.4)

It is common to use \(a \lor b\) for \(\text{OR}(a, b)\) and \(a \land b\) for \(\text{AND}(a, b)\), as well as write \(a \lor b \lor c\) as shorthand for \((a \lor b) \lor c\). (\(\text{NOT}(a)\) is often written as either \(\neg a\) or \(\overline{a}\); we will use both notations in this book.) With this notation, Eq. (3.4) can also be written as

\[
\text{MAJ}(x_0, x_1, x_2) = (x_0 \land x_1) \lor (x_1 \land x_2) \lor (x_0 \land x_2).
\] (3.5)
We can also write Eq. (3.4) in a “programming language” format, expressing it as a set of instructions for computing $\text{MAJ}$ given the basic operations $\text{AND}, \text{OR}, \text{NOT}$:

```python
def MAJ(X[0], X[1], X[2]):
    firstpair = AND(X[0], X[1])
    secondpair = AND(X[1], X[2])
    thirdpair = AND(X[0], X[2])
    temp = OR(secondpair, thirdpair)
    return OR(firstpair, temp)
```

Yet a third way to describe the same computation is by a Boolean circuit. Think of having wires that can carry a signal that is either the value 0 or 1. An OR gate is a gadget that has two incoming wires and one outgoing wires, and is designed so that if the signals on the incoming wires are $\mathcal{E}$ and $\mathcal{E}'$ respectively (for $\mathcal{E}, \mathcal{E}' \in \{0,1\}$), then the signal on the outgoing wire will be $\text{OR}(\mathcal{E}, \mathcal{E}')$. AND and NOT gates are defined similarly. Using this, we can express Eq. (3.4) as a circuit as well:

Example 3.1 — Computing $\text{XOR}$ from $\text{AND}, \text{OR}, \text{NOT}$. Let us see how we can obtain a different function from these building blocks. Define $\text{XOR} : \{0,1\}^2 \to \{0,1\}$ to be the function $\text{XOR}(a,b) = a + b \mod 2$. That is, $\text{XOR}(0,0) = \text{XOR}(1,1) = 0$ and $\text{XOR}(1,0) = \text{XOR}(0,1) = 1$. We claim that we can construct $\text{XOR}$ using only $\text{AND}, \text{OR},$ and $\text{NOT}$.

Here is an algorithm to compute $\text{XOR}(a,b)$ using $\text{AND}, \text{NOT}, \text{OR}$ as basic operations:

1. Compute $w1 = \text{AND}(a,b)$
2. Compute $w2 = \text{NOT}(w1)$
3. Compute $w3 = \text{OR}(a,b)$

\[\text{In practice, this is often implemented by electric potential or voltage on a wire, where for example voltage above a certain level is interpreted as a logical value of 1, and below a certain level is interpreted as a logical value of 0.}\]
4. Output $\text{AND}(w_2, w_3)$

We can also express this algorithm as a circuit:

```
X[0]  \rightarrow  AND  \rightarrow  \neg  \rightarrow  \rightarrow  \wedge  \rightarrow  Y[0]
X[1]  \rightarrow  \lor  \rightarrow  \rightarrow
```

Last but not least, we can also express it in a programming language. Specifically, the following is a Python program that computes the $XOR$ function:

```python
def AND(a, b):
    return a * b

def OR(a, b):
    return 1 - (1 - a) * (1 - b)

def NOT(a):
    return 1 - a

def XOR(a, b):
    w1 = AND(a, b)
    w2 = NOT(w1)
    w3 = OR(a, b)
    return AND(w2, w3)
```

```python
print([f"XOR({a},{b})={XOR(a,b)}" for a in [0,1]
      for b in [0,1]])
# ['XOR(0,0)=0', 'XOR(0,1)=1', 'XOR(1,0)=1', 'XOR(1,1)=0']
```

**Example 3.2 — Computing XOR on three bits.** Extending the same ideas, we can use these basic operations to compute the function $\text{XOR}_3 : \{0, 1\}^3 \rightarrow \{0, 1\}$ defined as $\text{XOR}_3(a, b, c) = a + b + c \pmod{2}$ by computing first $d = \text{XOR}(a, b)$ and then outputting $\text{XOR}(d, c)$. In Python this is done as follows:

```python
def XOR3(a, b, c):
    w1 = AND(a, b)
    w2 = NOT(w1)
    w3 = OR(a, b)
    w4 = AND(w2, w3)
    w5 = AND(w4, c)
    w6 = NOT(w5)
    w7 = OR(w4, c)
```
defining computation
127

```
return AND(w6, w7)
```

```
print({'f"XOR3((a), (b), (c))={XOR3(a, b, c)}" for a
       in [0, 1] for b in [0, 1] for c in [0, 1])
# ['XOR3(0, 0, 0)=0', 'XOR3(0, 0, 1)=1',
  'XOR3(0, 1, 0)=1', 'XOR3(0, 1, 1)=0',
  'XOR3(1, 0, 0)=1', 'XOR3(1, 0, 1)=0',
  'XOR3(1, 1, 0)=0', 'XOR3(1, 1, 1)=1']
```

Try to generalize the above examples to obtain a way to compute XORₙ : {0, 1}ⁿ → {0, 1} for every n using at most 4n basic steps involving applications of a function in {AND, OR, NOT} to outputs or previously computed values.

3.1.2 The NAND function

Here is another function we can compute using AND, OR, NOT. The NAND function maps {0, 1}² to {0, 1} and is defined as

```
NAND(a, b) = \begin{cases} 
0 & a = b = 1 \\
1 & \text{otherwise}
\end{cases}
```

(3.6)

As its name implies, NAND is the NOT of AND (i.e., NAND(a, b) = NOT(AND(a, b))), and so we can clearly compute NAND using AND and NOT. Interestingly, the opposite direction also holds:

**Theorem 3.3** — NAND computes AND, OR, NOT. We can compute AND, OR, and NOT by composing only the NAND function.

**Proof.** We start with the following observation. For every a ∈ {0, 1}, AND(a, a) = a. Hence, NAND(a, a) = NOT(AND(a, a)) = NOT(a).

This means that NAND can compute NOT, and since by the principle of “double negation”, AND(a, b) = NOT(NOT(AND(a, b))) this means that we can use NAND to compute AND as well. Once we can compute AND and NOT, we can compute OR using the so called “De Morgan’s Law”: OR(a, b) = NOT(AND(NOT(a), NOT(b))) (which can also be written as a ∨ b = ¬a ∧ ¬b) for every a, b ∈ {0, 1}.

**Theorem 3.3**’s proof is very simple, but you should make sure that (i) you understand the statement of the theorem, and (ii) you follow its proof completely. In particular, you should make sure you understand
why De Morgan’s law is true.

Verify NAND’s universality by Python (optional)  If you are so inclined, you can also verify the proof of Theorem 3.3 by Python:

def NAND(a, b):
    return 1 - a * b

def ORwithNAND(a, b):
    return NAND(NAND(a, a), NAND(b, b))

print([f"Test {a}, {b}: {ORwithNAND(a, b) == OR(a, b)}" for a in [0, 1] for b in [0, 1]])

Solved Exercise 3.1 — Compute majority with NAND. Let \( MAJ : \{0, 1\}^3 \to \{0, 1\} \) be the function that on input \( a, b, c \) outputs 1 iff \( a + b + c \geq 2 \). Show how to compute \( MAJ \) using a composition of \( NAND \)'s.

Solution: Recall that Eq. (3.4) stated that

\[
MAJ(x_0, x_1, x_2) = OR(AND(x_0, x_1), OR(AND(x_1, x_2), AND(x_0, x_2))).
\]

(3.7)

We can use Theorem 3.3 to replace all the occurrences of \( AND \) and \( OR \) with \( NAND \)'s. Specifically, we can use the equivalence \( AND(a, b) = NOT(NAND(a, b)), OR(a, b) = NAND(NOT(a), NOT(b)), and NOT(a) = NAND(a, a) \) to replace the righthand side of Eq. (3.7) with an expression involving only \( NAND \), yielding that \( MAJ(a, b, c) \) is equivalent the (somewhat unwieldy) expression

\[
NAND\left( NAND\left( NAND(NAND(a, b), NAND(a, c)),
NAND(NAND(a, b), NAND(a, c)) \right),
NAND(b, c) \right)
\]

(3.8)

This corresponds to the following circuit with \( NAND \) gates:
3.2 INFORMALLY DEFINING “BASIC OPERATIONS” AND “ALGORITHMS”

Theorem 3.3 tells us that we can use applications of the single function NAND to obtain AND, OR, NOT, and so by extension all the other functions that can be built up from them. So, if we wanted to decide on a “basic operation”, we might as well choose NAND, as we’ll get “for free” the three other operations AND, OR and NOT. This suggests the following definition of an “algorithm”:

**Semi-formal definition of an algorithm:** An algorithm consists of a sequence of steps of the form “store the NAND of variables bar and blah in variable foo”.

An algorithm A computes a function F if for every input x to F, if we feed x as input to the algorithm, the value computed in its last step is F(x).

There are several concerns that are raised by this definition:

1. First and foremost, this definition is indeed too informal. We do not specify exactly what each step does, nor what it means to “feed x as input”.

2. Second, the choice of NAND as a basic operation seems arbitrary. Why just NAND? Why not AND, OR or NOT? Why not allow operations like addition and multiplication? What about any other logical constructions such if/then or while?

3. Third, do we even know that this definition has anything to do with actual computing? If someone gave us a description of such an algorithm, could we use it to actually compute the function in the real world?
These concerns will to a large extent guide us in the upcoming chapters. Thus you would be well advised to re-read the above informal definition and see what you think about these issues.

A large part of this course will be devoted to addressing the above issues. We will see that:

1. We can make the definition of an algorithm fully formal, and so give a precise mathematical meaning to statements such as “Algorithm \( A \) computes function \( F \).”

2. While the choice of \( \text{NAND} \) is arbitrary, and we could just as well chose some other functions, we will also see this choice does not matter much. Our notion of an algorithm is not more restrictive because we only think of \( \text{NAND} \) as a basic step. We have already seen that allowing \( \text{AND}, \text{OR}, \text{NOT} \) as basic operations will not add any power (because we can compute them from \( \text{NAND} \)'s via Theorem 3.3). We will see that the same is true for addition, multiplication, and essentially every other operation that could be reasonably thought of as a basic step.

3. It turns out that we can and do compute such “\( \text{NAND} \) based algorithms” in the real world. First of all, such an algorithm is clearly well specified, and so can be executed by a human with a pen and paper. Second, there are a variety of ways to mechanize this computation. We’ve already seen that we can write Python code that corresponds to following such a list of instructions. But in fact we can directly implement operations such as \( \text{NAND}, \text{AND}, \text{OR}, \text{NOT} \) etc. via electronic signals using components known as transistors. This is how modern electronic computers operate.

In the remainder of this chapter, we will begin to answer some of these questions. We will see more examples of the power of simple operations like \( \text{NAND} \) (or equivalently, \( \text{AND}, \text{OR}, \text{NOT} \), as well as many other choices) to compute more complex operations including addition, multiplication, sorting and more. We will then discuss how to physically implement simple operations such as NAND using a variety of technologies. Finally we will define the \( \text{NAND} \) programming language that will be our formal model of computation.

### 3.3 FROM NAND TO INFINITY AND BEYOND...

We have seen that using \( \text{NAND} \), we can compute \( \text{AND}, \text{OR}, \text{NOT} \) and \( \text{XOR} \). But this still seems a far cry from being able to add and multiply numbers, not to mention more complex programs such as
sorting and searching, solving equations, manipulating images, and so on. We now give a few examples demonstrating how we can use these simple operations to do some more complicated tasks. While we will not go as far as implementing Call of Duty using NAND, we will at least show how we can compose NAND operations to obtain tasks such as addition, multiplications, and comparisons.

### 3.3.1 NAND Circuits

We can describe the computation of a function \( F : \{0,1\}^n \rightarrow \{0,1\} \) via a composition of NAND operations in terms of a circuit, as was done in Example 3.1. Since in our case, all the gates are the same function (i.e., NAND), the description of the circuit is even simpler. We can think of the circuit as a directed graph. It has a vertex for every one of the input bits, and also for every intermediate value we use in our computation. If we compute a value \( u \) by applying NAND to \( v \) and \( w \) then we put a directed edges from \( v \) to \( u \) and from \( w \) to \( u \). We will follow the convention of using “\( x \)” for inputs and “\( y \)” for outputs, and hence write \( x_0, x_1, \ldots \) for our inputs and \( y_0, y_1, \ldots \) for our outputs. (We will sometimes also write these as \( X[0], X[1], \ldots \) and \( Y[0], Y[1], \ldots \) respectively.) Here is a more formal definition:

---

Before reading the formal definition, it would be an extremely good exercise for you to pause here and try to think how you would formally define the notion of a NAND circuit. Sometimes working out the definition for yourself is easier than parsing its text.

---

**Definition 3.4 — NAND circuits.** Let \( n, m, s > 0 \). A NAND circuit \( C \) with \( n \) inputs, \( m \) outputs, and \( s \) gates is a labeled directed acyclic graph (DAG) with \( n + s \) vertices such that:

- \( C \) has \( n \) vertices with no incoming edges, which are called the *input vertices* and are labeled with \( X[0], \ldots, X[n−1] \).

- \( C \) has \( s \) vertices each with exactly two (possibly parallel) incoming edges, which are called the *gates*.

- \( C \) has \( m \) gates which are called the *output vertices* and are labeled with \( Y[0], \ldots, Y[m−1] \). The output vertices have no outgoing edges.

For \( x \in \{0,1\}^n \), the *output* of \( C \) on input \( x \), denoted by \( C(X) \), is computed in the natural way. For every \( i \in [n], \) we assign to the
input vertex $X[i]$ the value $x_i$, and then continuously assign to every gate the value which is the NAND of the values assigned to its two incoming neighbors. The output is the string $y \in \{0, 1\}^m$ such that for every $j \in [m]$, $y_j$ is the value assigned to the output gate labeled with $Y[j]$.

Definition 3.4 is perhaps our first encounter with a somewhat complicated definition. When you are faced with such a definition, there are several strategies to try to understand it:

1. First, as we suggested above, you might want to see how you would formalize the intuitive notion that the definitions tries to capture. If we made different choices than you would, try to think why is that the case.

2. Then, you should read the definition carefully, making sure you understand all the terms that it uses, and all the conditions it imposes.

3. Finally, try to how the definition corresponds to simple examples such as the NAND circuit presented in ??, as well as the examples illustrated below.

We now present some examples of NAND circuits for various natural problems:

Example 3.5 — NAND circuit for XOR. Recall the XOR function which maps $x_0, x_1 \in \{0, 1\}$ to $x_0 + x_1 \mod 2$. We have seen in Example 3.1 that we can compute this function using AND, OR, and NOT, and so by Theorem 3.3 we can compute it using only NAND’s. However, the following is a direct construction of computing XOR by a sequence of NAND operations:

1. Let $u = NAND(x_0, x_1)$.

2. Let $v = NAND(x_0, u)$

3. Let $w = NAND(x_1, u)$

4. The XOR of $x_0$ and $x_1$ is $y_0 = NAND(v, w)$.

(We leave it to you to verify that this algorithm does indeed compute XOR.)

We can also represent this algorithm graphically as a circuit:
We now present a few more examples of computing natural functions by a sequence of \textit{NAND} operations.

\textbf{Example 3.6 — NAND circuit for incrementing.} Consider the task of computing, given as input a string \(x \in \{0,1\}^n\) that represents a natural number \(X \in \mathbb{N}\), the representation of \(X + 1\). That is, we want to compute the function \(\text{INC}_n : \{0,1\}^n \to \{0,1\}^{n+1}\) such that for every \(x_0, \ldots, x_{n-1}\), \(\text{INC}_n(x) = y\) which satisfies
\[\sum_{i=0}^{n} y_i 2^i = (\sum_{i=0}^{n-1} x_i 2^i) + 1.\]

The increment operation can be very informally described as follows: \textit{“Add 1 to the least significant bit and propagate the carry”}. A little more precisely, in the case of the binary representation, to obtain the increment of \(x\), we scan \(x\) from the least significant bit onwards, and flip all 1’s to 0’s until we encounter a bit equal to 0, in which case we flip it to 1 and stop. (Please verify you understand why this is the case.)

Thus we can compute the increment of \(x_0, \ldots, x_{n-1}\) by doing the following:

1. Set \(c_0 = 1\) (we pretend we have a “carry” of 1 initially)
2. For \(i = 0, \ldots, n - 1\) do the following:
   1. Let \(y_i = \text{XOR}(x_i, c_i)\).
2. If $c_i = x_i = 1$ then $c_{i+1} = 1$, else $c_{i+1} = 0$.

1. Set $y_n = c_n$.

The above is a very precise description of an algorithm to compute the increment operation, and can be easily transformed into Python code that performs the same computation, but it does not seem to directly yield a NAND circuit to compute this. However, we can transform this algorithm line by line to a NAND circuit. For example, since for every $a$, $NAND(a, NOT(a)) = 1$, we can replace the initial statement $c_0 = 1$ with $c_0 = NAND(x_0, NAND(x_0, x_0))$.

We already know how to compute $XOR$ using NAND, so line 2.a can be replaced by some NAND operations. Next, we can write line 2.b as simply saying $c_{i+1} = NAND(y_i, x_i)$, or in other words $c_{i+1} = NAND(NAND(y_i, x_i), NAND(y_i, x_i))$. Finally, the assignment $y_n = c_n$ can be written as $y_n = NAND(NAND(c_n, c_n), NAND(c_n, c_n))$.

Combining these observations yields for every $n \in \mathbb{N}$, a NAND circuit to compute $INC_n$. For example, this is how this circuit looks like for $n = 4$. 


Example 3.7 — Addition using NANDs. Once we have the increment operation, we can certainly compute addition by repeatedly incrementing (i.e., compute $x + y$ by performing $INC(x)$ $y$ times). However, that would be quite inefficient and unnecessary. With the same idea of keeping track of carries we can implement the "grade-school" algorithm for addition to compute the function $ADD_n : \{0, 1\}^{2n} \to \{0, 1\}^{n+1}$ that on input $x \in \{0, 1\}^{2n}$ outputs the binary representation of the sum of the numbers represented.
by $x_0, \ldots, x_{n-1}$ and $x_{n+1}, \ldots, x_n$:

1. Set $c_0 = 0$.
2. For $i = 0, \ldots, n - 1$:
   
   (a) Let $y_i = x_i + x_{i+1} + c_i \pmod{2}$.
   (b) If $x_i + x_{i+1} + c_i \geq 2$ then $c_{i+1} = 1$.
3. Let $y_n = c_n$

Once again, this can be translated into a NAND circuit. To transform Step 2.b to a NAND circuit we use the fact (shown in Solved Exercise 3.1) that the function $MAJ_3 : \{0,1\}^3 \to \{0,1\}$ can be computed using NANDs.

### 3.4 Physical Implementations of Computing Devices.

Computation is an abstract notion, that is distinct from its physical implementations. While most modern computing devices are obtained by mapping logical gates to semi-conductor based transistors, over history people have computed using a huge variety of mechanisms, including mechanical systems, gas and liquid (known as fluidics), biological and chemical processes, and even living creatures (e.g., see Fig. 3.5 or this video for how crabs or slime mold can be used to do computations).

In this section we will review some of these implementations, both so you can get an appreciation of how it is possible to directly translate NAND programs to the physical world, without going through the entire stack of architecture, operating systems, compilers, etc. as well as to emphasize that silicon-based processors are by no means the only way to perform computation. Indeed, as we will see much later in this course, a very exciting recent line of works involves using different media for computation that would allow us to take advantage of quantum mechanical effects to enable different types of algorithms.

#### 3.4.1 Transistors and physical logic gates

A transistor can be thought of as an electric circuit with two inputs, known as source and gate and an output, known as the sink. The gate controls whether current flows from the source to the sink. In a standard transistor, if the gate is “ON” then current can flow from the source to the sink and if it is “OFF” then it can’t. In a complementary transistor this is reversed: if the gate is “OFF” then current can flow from the source to the sink and if it is “ON” then it can’t.

There are several ways to implement the logic of a transistor. For example, we can use faucets to implement it using water pressure
Figure 3.5. Crab-based logic gates from the paper “Robust soldier-crab ball gate” by Gunji, Nishiyama and Adamatzky. This is an example of an AND gate that relies on the tendency of two swarms of crabs arriving from different directions to combine to a single swarm that continues in the average of the directions.

Figure 3.6. We can implement the logic of transistors using water. The water pressure from the gate closes or opens a faucet between the source and the sink.
However, the standard implementation uses electrical current. One of the original implementations used vacuum tubes. As its name implies, a vacuum tube is a tube containing nothing (i.e., vacuum) and where a priori electrons could freely flow from source (a wire) to the sink (a plate). However, there is a gate (a grid) between the two, where modulating its voltage can block the flow of electrons.

Early vacuum tubes were roughly the size of lightbulbs (and looked very much like them too). In the 1950’s they were supplanted by transistors, which implement the same logic using semiconductors which are materials that normally do not conduct electricity but whose conductivity can be modified and controlled by inserting impurities (“doping”) and an external electric field (this is known as the field effect). In the 1960’s computers were started to be implemented using integrated circuits which enabled much greater density. In 1965, Gordon Moore predicted that the number of transistors per circuit would double every year (see Fig. 3.7), and that this would lead to “such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment”. Since then, (adjusted versions of) this so-called “Moore’s law” has been running strong, though exponential growth cannot be sustained forever, and some physical limitations are already becoming apparent.
Figure 3.8: Gordon Moore’s cartoon “predicting” the implications of radically improving transistor density.

Figure 3.9: The exponential growth in computing power over the last 120 years. Graph by Steve Jurvetson, extending a prior graph of Ray Kurzweil.
3.4.2 NAND gates from transistors
We can use transistors to implement a NAND gate, which would be a system with two input wires \(x, y\) and one output wire \(z\), such that if we identify high voltage with “1” and low voltage with “0”, then the wire \(z\) will equal to “1” if and only if the NAND of the values of the wires \(x\) and \(y\) is 1 (see Fig. 3.10).

![Figure 3.10: Implementing a NAND gate using transistors.](image)

This means that there exists a NAND circuit to compute a function \(F : \{0, 1\}^n \rightarrow \{0, 1\}^m\), then we can compute \(F\) in the physical world using transistors as well.

3.5 Basing computing on other media (optional)
Electronic transistors are in no way the only technology that can implement computation. There are many mechanical, chemical, biological, or even social systems that can be thought of as computing devices. We now discuss some of these examples.

3.5.1 Biological computing
Computation can be based on biological or chemical systems. For example the lac operon produces the enzymes needed to digest lactose only if the conditions \(x \land (\neg y)\) hold where \(x\) is “lactose is present” and \(y\) is “glucose is present”. Researchers have managed to create transistors, and from them the NAND function and other logic gates, based on DNA molecules (see also Fig. 3.11). One motivation for DNA computing is to achieve increased parallelism or storage density; another
is to create “smart biological agents” that could perhaps be injected into bodies, replicate themselves, and fix or kill cells that were damaged by a disease such as cancer. Computing in biological systems is not restricted of course to DNA. Even larger systems such as flocks of birds can be considered as computational processes.

![Figure 3.11: Performance of DNA-based logic gates. Figure taken from paper of Bonnet et al, Science, 2013.](image)

**3.5.2 Cellular automata and the game of life**

*Cellular automata* is a model of a system composed of a sequence of cells, which of which can have a finite state. At each step, a cell updates its state based on the states of its neighboring cells and some simple rules. As we will discuss later in this course, cellular automata such as Conway’s “Game of Life” can be used to simulate computation gates (see Fig. 3.12).

**3.5.3 Neural networks**

One computation device that we all carry with us is our own brain. Brains have served humanity throughout history, doing computations that range from distinguishing prey from predators, through making scientific discoveries and artistic masterpieces, to composing witty 280 character messages. The exact working of the brain is still not fully understood, but it seems that to a first approximation it can be modeled by a (very large) neural network.

A neural network is a Boolean circuit that instead of $NAND$ (or
even AND/OR/NOT) uses some other gates as the basic basis. For example, one particular basis we can use are threshold gates. For every vector \( w = (w_0, \ldots, w_{k-1}) \) of integers and integer \( t \) (some or all of whom could be negative), the threshold function corresponding to \( w, t \) is the function \( T_{w,t} : \{0,1\}^k \rightarrow \{0,1\} \) that maps \( x \in \{0,1\}^k \) to 1 if and only if \( \sum_{i=0}^{k-1} w_i x_i \geq t \). For example, the threshold function \( T_{w,t} \) corresponding to \( w = (1,1,1,1,1) \) and \( t = 3 \) is simply the majority function \( MAJ_5 \) on \( \{0,1\}^5 \). The function \( NAND : \{0,1\}^2 \rightarrow \{0,1\} \) is the threshold function corresponding to \( w = (-1, -1) \) and \( t = -1 \), since \( NAND(x_0, x_1) = 1 \) if and only if \( x_0 + x_1 \leq 1 \) or equivalently, \( -x_0 - x_1 \geq -1 \).

Threshold gates can be thought of as an approximation for neuron cells that make up the core of human and animal brains. To a first approximation, a neuron has \( k \) inputs and a single output and the neurons “fires” or “turns on” its output when those signals pass some threshold. Unlike the cases above, when we considered the number of inputs to a gate \( k \) to be a small constant, in such neural networks we often do not put any bound on the number of inputs. However, since any threshold function on \( k \) inputs can be computed by a NAND circuit of at most \( poly(k) \) gates (see Exercise 3.3), NAND circuits are no less powerful than neural networks.

### 3.5.4 The marble computer

We can implement computation using many other physical media, without need for any electronic, biological, or chemical components. Many suggestions for mechanical computers have been put forward, starting with Charles Babbage’s 1837 plan for a mechanical “Analytical Engine”.

---

\( ^{\text{6}} \) Threshold is just one example of gates that can used by neural networks.
More generally, a neural network is often described as operating on signals that are real numbers, rather than 0/1 values, and where the output of a gate on inputs \( x_0, \ldots, x_{k-1} \) is obtained by applying \( f(\sum_i w_i x_i) \) where \( f : \mathbb{R} \rightarrow \mathbb{R} \) is an activation function such as rectified linear unit (ReLU), Sigmoid, or many others. However, for the purpose of our discussion, all of the above are equivalent. In particular we can reduce the real case to the binary case by a real number in the binary basis, and multiplying the weight of the bit corresponding to the \( i^{th} \) digit by \( 2^i \).
As one example, Fig. 3.13 shows a simple implementation of a NAND gate using marbles going through pipes. We represent a logical value in \( \{0, 1\} \) by a pair of pipes, such that there is a marble flowing through exactly one of the pipes. We call one of the pipes the “0 pipe” and the other the “1 pipe”, and so the identity of the pipe containing the marble determines the logical value. A NAND gate would correspond to some mechanical object with two pairs of incoming pipes and one pair of outgoing pipes, such that for every \( a, b \in \{0, 1\} \), if two marble are rolling toward the object in the \( a \) pipe of the first pair and the \( b \) pipe of the second pair, then a marble will roll out of the object in the \( \text{NAND}(a, b) \)-pipe of the outgoing pair.

As shown in Fig. 3.13, we can achieve such a NAND gate in a fairly straightforward way, together with a gadget that ensures that at most one marble flows in each wire. Such NAND gates can be combined together to form for every \( n \)-input NAND circuit \( \mathcal{P} \) a physical computer that simulates \( \mathcal{P} \) in the sense that if the marbles are placed in its incoming pipes according to some input \( x \in \{0, 1\}^n \), then eventually marbles will come out of its outgoing pipes according to the output \( P(x) \).

Figure 3.13: A physical implementation of a NAND gate using marbles. Each wire in a Boolean circuit is modeled by a pair of pipes representing the values 0 and 1 respectively, and hence a gate has four input pipes (two for each logical input) and two output pipes. If one of the input pipes representing the value 0 has a marble in it then that marble will flow to the output pipe representing the value 1. (The dashed line represent a gadget that will ensure that at most one marble is allowed to flow onward in the pipe.) If both the input pipes representing the value 1 have marbles in them, then the first marble will be stuck but the second one will flow onwards to the output pipe representing the value 0.
We follow the common programming languages convention of using names such as foo, bar, baz, blah as stand-ins for generic identifiers. Generally a variable identifier in the NAND programming language can be any combination of letters and numbers, and we will also sometimes have identifiers such as Foo[12] that end with a number inside square brackets. Later in the course we will introduce programming languages where such identifiers carry special meaning as arrays. At the moment you can treat them as simply any other identifier. The appendix contains a full formal specification of the NAND programming language.

---

**Figure 3.14**: A “gadget” in a pipe that ensures that at most one marble can pass through it. The first marble that passes causes the barrier to lift and block new ones.

### 3.6 THE NAND PROGRAMMING LANGUAGE

We now turn to formally defining the notion of algorithm. We use a programming language to do so. We define the NAND Programming Language to be a programming language where every line has the following form:

\[ \text{foo} = \text{NAND}(\text{bar}, \text{blah}) \]

where foo, bar and blah are variable identifiers.\(^8\)

---

**Example 3.8 — Our first NAND program.** Here is an example of a NAND program:

- \[ u = \text{NAND}(X[0], X[1]) \]
- \[ v = \text{NAND}(X[0], u) \]
- \[ w = \text{NAND}(X[1], u) \]
- \[ Y[0] = \text{NAND}(v, w) \]

Do you know what function this program computes? Hint: you have seen it before.

---

As you might have guessed from this example, we have two special types of variables in the NAND language: input variables have the form \(X[i]\) where \(i\) is a natural number, and output variables have the form \(Y[j]\) where \(j\) is a natural number. When a NAND program is executed on input \(x \in \{0, 1\}^n\), the variable \(X[i]\) is assigned the value \(x_i\) for all \(i \in [n]\). The output of the program is the list of \(m\) values \(Y[0]...Y[m - 1]\), where \(m - 1\) is the largest index for which the variable \(Y[m - 1]\) is assigned a value in the program. If a line of

---

\(^8\) We follow the common programming languages convention of using names such as foo, bar, baz, blah as stand-ins for generic identifiers. Generally a variable identifier in the NAND programming language can be any combination of letters and numbers, and we will also sometimes have identifiers such as Foo[12] that end with a number inside square brackets. Later in the course we will introduce programming languages where such identifiers carry special meaning as arrays. At the moment you can treat them as simply any other identifier. The appendix contains a full formal specification of the NAND programming language.
the form \( \text{foo} = \text{NAND}(\text{bar}, \text{blah}) \) appears in the program, then if \( \text{bar} \) is not an input variable of the form \( X[i] \), then it must have been assigned a value in a previous line, and the same holds for \( \text{blah} \). We also forbid assigning a value to an input variable, and applying the NAND operation to an output variable.

We can now formally define the notion of a function being computed by a NAND program:

**Definition 3.9 — Computing by a NAND program.** Let \( F : \{0, 1\}^n \rightarrow \{0, 1\}^m \) be some function, and let \( P \) be a NAND program. We say that \( P \) computes the function \( F \) if:

1. \( P \) has \( n \) input variables \( X[0], \ldots, X[n-1] \) and \( m \) output variables \( Y[0], \ldots,Y[m-1] \).

2. For every \( x \in \{0, 1\}^n \), if we execute \( P \) when we assign to \( X[0], \ldots, X[n-1] \) the values \( x_0, \ldots, x_{n-1} \), then at the end of the execution, the output variables \( Y[0], \ldots,Y[m-1] \) have the values \( y_0, \ldots, y_{m-1} \) where \( y = F(x) \).

**Definition 3.9** is one of the most important definitions in this book. Please make sure to read it time and again until you are sure that you understand it. A full formal specification of the execution model of NAND programs appears in the appendix.

**Is the NAND programming language Turing Complete? (optional note)** You might have heard of a term called “Turing Complete” to describe programming languages. (If you haven’t, feel free to ignore the rest of this remark: we will encounter this term later in this course and define it properly.) If so, you might wonder if the NAND programming language has this property. The answer is no, or perhaps more accurately, the term is not really applicable for the NAND programming language. The reason is that, by design, the NAND programming language can only compute **finite** functions \( F : \{0, 1\}^n \rightarrow \{0, 1\}^m \) that take a fixed number of input bits and produce a fixed number of outputs bits. The term “Turing Complete” is really only applicable to programming languages for **infinite** functions that can take inputs of arbitrary length. We will come back to this distinction later on in the course.
3.6.1 NAND programs and NAND circuits

So far we have described two models of computation:

- **NAND circuits**, which are obtained by applying NAND gates to inputs.
- **NAND programs**, which are obtained by repeatedly applying operations of the form $\text{foo} = \text{NAND}(\text{bar}, \text{blah})$.

A central result is that these two models are actually equivalent:

**Theorem 3.10** — Circuit and straightline program equivalence. Let $F: \{0,1\}^n \to \{0,1\}^m$ and $s \in \mathbb{N}$. Then $F$ is computable by a NAND program of $s$ lines if and only if it is computable by a NAND circuit of $s$ gates.

**Proof Idea:** To understand the proof, you can first work out for yourself the equivalence between the NAND program of Example 3.8 and the circuit we have seen in Example 3.5, see also Fig. 3.15. Generally, if we have a NAND program, we can transform it into a circuit by mapping every line $\text{foo} = \text{NAND}(\text{bar}, \text{blah})$ of the program into a gate $\text{foo}$ that is applied to the result of the previous gates $\text{bar}$ and $\text{blah}$. (Since we always assign a variable to variables that have been assigned before or are input variables, we can assume that $\text{bar}$ and $\text{blah}$ are either gates we already constructed or are inputs to the circuit.) In the reverse direction, to map a circuit $C$ into a program $P$ we use topological sorting to sort the vertices of the graph of $C$ into an order $v_0, v_1, \ldots, v_s$ such that if there is an edge from $v_i$ to $v_j$ then $j > i$. Thus we can transform every gate (i.e. non input vertex) of the circuit into a line in a program in an analogous way: if $v$ is a gate that has two incoming edges from $u$ and $w$, then we add a variable $\text{foo}$ corresponding to $v$ and a line $\text{foo} = \text{NAND}(\text{bar}, \text{blah})$ where $\text{bar}$ and $\text{blah}$ are the variables corresponding to $u$ and $w$.

**Proof of Theorem 3.10.** Let $F: \{0,1\}^n \to \{0,1\}^m$ be a function. Suppose that there exists a program $P$ of $s$ lines that computes $F$. We construct a NAND circuit $C$ to compute $F$ as follows: the circuit will include $n$ input vertices, and will include $s$ gates, one for each of the lines of $P$. We let $I(0), \ldots, I(n-1)$ denotes the vertices corresponding to the inputs and $G(0), \ldots, G(s-1)$ denote the vertices corresponding to the lines. We connect our gates in the natural way as follows:

If the $\ell$-th line of $P$ has the form $\text{foo} = \text{NAND}(\text{bar}, \text{blah})$ where $\text{bar}$ and $\text{blah}$ are variables not of the form $X[i]$, then $\text{bar}$ and $\text{blah}$ must have been assigned a value before. We let $j$ and $k$ be the last lines before the $\ell$-th line in which the variables $\text{bar}$ and $\text{blah}$ respectively
Figure 3.15: The NAND code and the corresponding circuit for a program to compute the increment function that maps a string $x \in \{0,1\}^3$ (which we think of as a number in $\mathbb{Z}_7$) to the string $y \in \{0,1\}^4$ that represents $x + 1$. Note how every line in the program corresponds to a gate in the circuit.

were assigned a value. In such a case, we will add the edges $G(j) \rightarrow G(\ell)$ and $G(k) \rightarrow G(\ell)$ to our circuit $C$. That is, we will apply the gate $G(\ell)$ to the outputs of the gates $G(j)$ and $G(k)$. If $bar$ is an input variable of the form $X[i]$ then we connect $G(\ell)$ to the corresponding input vertex $I(i)$, and do the analogous step if $blah$ is an input variable. Finally, for every $j \in \{0,1,\ldots,m\}$ if $\ell(j)$ is the last line which assigns a value to $Y[j]$, then we mark the gate $G(j)$ as the $j$-th output gate of the circuit $C$.

We claim that the circuit $C$ computes the same function as the program $P$. Indeed, one can show by induction on $\ell$ that for every input $x \in \{0,1\}^n$, if we execute $P$ on input $x$, then the value assigned to the variable in the $\ell$-th line is the same as the value output by the gate $G(\ell)$ in the circuit $C$. (To see this note that by the induction hypothesis, this is true for the values that the $\ell$-th line uses, as they were assigned a value in earlier lines or are inputs, and both the gate and the line compute the NAND function on these values.) Hence in particular the output variables of the program will have the same value as the output gates of the circuits.

In the other direction, given a circuit $C$ of $s$ gates that computes $F$, we can construct a program of $s$ lines that computes the same function. We use a topological sort to ensure that the $n + s$ vertices of the graph of $C$ are sorted so that all edges go from earlier vertices to later ones, and ensure the first $n$ vertices $0, 1, \ldots, n - 1$ correspond to the $n$ inputs. (This can be ensured as input vertices have no incoming edges.) Then for every $\ell \in [s]$, the $\ell$-th line of the program $P$ will correspond to the vertex $n + \ell$ of the circuit. If vertex $n + \ell$'s incoming
There is a minor technical complication when using gates corresponding to non-symmetric functions. A function \( f : \{0,1\}^n \rightarrow \{0,1\} \) is symmetric if reordering its inputs does not make a difference to the output. For example, the functions \( \land, \lor, \lnot \) are symmetric. If we consider circuits with gates that are non-symmetric functions, then we need to label each wire entering a gate as to which parameter of the function it correspond to.

The proof of Theorem 3.10 is constructive, in the sense that it yields an explicit transformation from a program to a circuit and vice versa. The appendix contains code of a Python function that outputs the circuit corresponding to a program.

\[ \text{R Constructive proof} \]

**3.6.2 Circuits with other gate sets (optional)**

There is nothing special about NAND. For every set of functions \( \mathcal{G} = \{G_0, \ldots, G_{k-1}\} \), we can define a notion of circuits that use elements of \( \mathcal{G} \) as gates, and a notion of a “\( \mathcal{G} \) programming language” where every line involves assigning to a variable \( \text{foo} \) the result of applying some \( G_i \in \mathcal{G} \) to previously defined or input variables. Specifically, we can make the following definition:

**Definition 3.11 — General straightline programs.** Let \( \mathcal{F} = \{f_0, \ldots, f_{l-1}\} \) be a finite collection of Boolean functions, such that \( f_i : \{0,1\}^{k_i} \rightarrow \{0,1\} \) for some \( k_i \in \mathbb{N} \). An \( \mathcal{F} \) program is a sequence of lines, each of which assigns to some variable the result of applying some \( f_i \in \mathcal{F} \) to \( k_i \) other variables. As above, we use \( X[i] \) and \( Y[j] \) to denote the input and output variables.

\( \text{NAND programs} \) corresponds to \( \mathcal{F} \) programs for the set \( \mathcal{F} \) that only contains the \( \text{NAND} \) function, but we can talk about \( \{\text{AND, OR, NOT}\} \) programs, \( \{\text{XOR, 0, 1}\} \) programs, or use any other set. We can also define \( \mathcal{F} \) circuits, which will be directed graphs in which the gates corresponds to applying a function \( f_i \in \mathcal{F} \), and will each have \( k_i \) incoming wires and a single outgoing wire. As in Theorem 3.10, we can show that \( \mathcal{F} \) circuits and \( \mathcal{F} \) programs are equivalent. We have seen that for \( \mathcal{F} = \{\text{AND, OR, NOT}\} \), the resulting circuits/programs are equivalent in power to the NAND programming language, as we can compute \( \text{NAND} \) using \( \text{AND/OR/NOT} \) and vice versa. This turns out to be a special case of a general phenomena— the universality of \( \text{NAND} \) and other gate sets— that we will explore more in depth later in this course. However,
there are some sets $\mathcal{F}$ that are not equivalent in power to $\textit{NAND}$: see Exercise 3.1 for more.

### Lecture Recap

- **Algorithm** is a recipe for performing a computation as a sequence of “elementary” or “simple” operations.
- One candidate definition for an “elementary” operation is the $\textit{NAND}$ operation. It is an operation that is easily implementable in the physical world in a variety of methods including by electronic transistors.
- We can use $\textit{NAND}$ to compute many other functions, including majority, increment, and others.
- There are other equivalent choices, including the set $\{\textit{AND}, \textit{OR}, \textit{NOT}\}$.
- We can formally define the notion of a function $F : \{0,1\}^n \to \{0,1\}^m$ being computable using the $\textit{NAND}$ Programming language.
- The notions of being computable by a $\textit{NAND}$ circuit and being computable by a $\textit{NAND}$ program are equivalent.

### 3.7 Exercises

#### Disclaimer

Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

**Exercise 3.1 — Universal basis.** Define a set $\mathcal{F}$ of functions to be a universal basis if we can compute $\textit{NAND}$ using $\mathcal{F}$. For every one of the following sets, either prove that it is a universal basis or prove that it is not.

1. $\mathcal{F} = \{\textit{AND}, \textit{OR}, \textit{NOT}\}$.
2. $\mathcal{F} = \{\textit{AND}, \textit{OR}\}$.
3. $\mathcal{F} = \{\textit{OR}, \textit{NOT}\}$.
4. $\mathcal{F} = \{\textit{NOR}\}$ where $\textit{NOR}(a, b) = \textit{NOT}(\textit{OR}(a, b))$. 
5. \( \mathcal{F} = \{XOR, 0, 1\} \) where 0 and 1 are the constant functions that take no input and output 0 and 1.

6. \( \mathcal{F} = \{LOOKUP_1, 0, 1\} \) where 0 and 1 are the constant functions as above and \( \text{LOOKUP}_1 : \{0, 1\}^3 \to \{0, 1\} \) satisfies \( \text{LOOKUP}_1(a, b, c) \) equals \( a \) if \( c = 0 \) and equals \( b \) if \( c = 1 \).

Exercise 3.2 — Bound on universal basis size (challenge). Prove that for every subset \( B \) of the functions from \( \{0, 1\}^k \) to \( \{0, 1\} \), if \( B \) is universal then there is a \( B \)-circuit of at most \( O(k) \) gates to compute the NAND function (you can start by showing that there is a \( B \) circuit of at most \( O(k^{16}) \) gates).

Exercise 3.3 — Threshold using NANDs. Prove that for every \( w, t \), the function \( T_{w, t} \) can be computed by a NAND program of at most \( O(k^3) \) lines.

3.8 BIOGRAPHICAL NOTES

3.9 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

- Efficient constructions of circuits: finding circuits of minimal size that compute certain functions.
4

Syntactic sugar, and computing every function

“[In 1951] I had a running compiler and nobody would touch it because, they carefully told me, computers could only do arithmetic; they could not do programs.”, Grace Murray Hopper, 1986.

“Syntactic sugar causes cancer of the semicolon.”, Alan Perlis, 1982.

The NAND programing language is pretty much as “bare bones” as programming languages come. After all, it only has a single operation. But, it turns out we can implement some “added features” on top of it. That is, we can show how we can implement those features using the underlying mechanisms of the language.

Let’s start with a simple example. One of the most basic operations a programming language has is to assign the value of one variable into another. And yet in NAND, we cannot even do that, as we only allow assignments of the result of a NAND operation. Yet, it is possible to “pretend” that we have such an assignment operation, by transforming code such as

\[
\text{foo} = \text{COPY}(\text{bar})
\]

into the valid NAND code:

\[
\text{notbar} = \text{NAND}(\text{bar}, \text{bar}) \\
\text{foo} = \text{NAND}(\text{notbar}, \text{notbar})
\]

the reason being that for every \( a \in \{0, 1\} \), \( \text{NAND}(a, a) = \text{NOT}(a) \text{AND}a = \text{NOT}(a) \) and so in these two lines \( \text{notbar} \) is assigned the negation of \( \text{bar} \) and so \( \text{foo} \) is assigned the negation of the negation of \( \text{bar} \), which is simply \( \text{bar} \).
Thus in describing NAND programs we can (and will) allow ourselves to use the variable assignment operation, with the understanding that in actual programs we will replace every line of the first form with the two lines of the second form. In programming language parlance this is known as “syntactic sugar”, since we are not changing the definition of the language, but merely introducing some convenient notational shortcuts.¹ We will use several such “syntactic sugar” constructs to make our descriptions of NAND programs shorter and simpler. However, these descriptions are merely shorthand for the equivalent standard or “sugar free” NAND program that is obtained after removing the use of all these constructs. In particular, when we say that a function \( F \) has an \( s \)-line NAND program, we mean a standard NAND program, that does not use any syntactic sugar. The website [http://www.nandpl.org](http://www.nandpl.org) contains an online “unsweetener” that can take a NAND program that uses these features and modifies it to an equivalent program that does not use them.

### 4.1 SOME USEFUL SYNTACTIC SUGAR

In this section, we will list some additional examples of “syntactic sugar” transformations. Going over all these examples can be somewhat tedious, but we do it for two reasons:

1. To convince you that despite its seeming simplicity and limitations, the NAND programming language is actually quite powerful and can capture many of the fancy programming constructs such as \( \text{if} \) statements and function definitions that exists in more fashionable languages.

2. So you can realize how lucky you are to be taking a theory of computation course and not a compilers course… :)

#### 4.1.1 Constants

We can create variables \( \text{zero} \) and \( \text{one} \) that have the values 0 and 1 respectively by adding the lines

\[
\begin{align*}
\text{temp} & = \text{NAND}(x[0], x[0]) \\
\text{one} & = \text{NAND}(\text{temp}, x[0]) \\
\text{zero} & = \text{NAND}(\text{one}, \text{one})
\end{align*}
\]

Note that since for every \( x \in \{0,1\}, \text{NAND}(x, \overline{x}) = 1 \), the variable \( \text{one} \) will get the value 1 regardless of the value of \( x_0 \), and the variable \( \text{zero} \) will get the value \( \text{NAND}(1, 1) = 0 \).² We can combine the above two techniques to enable assigning constants to variables in our programs.

¹ This concept is also known as “macros” or “meta-programming” and is sometimes implemented via a preprocessor or macro language in a programming language or a text editor. One modern example is the Babel JavaScript syntax transformer, that converts JavaScript programs written using the latest features into a format that older Browsers can accept. It even has a plug-in architecture, that allows users to add their own syntactic sugar to the language.

² We could have saved a couple of lines using the convention that uninitialized variables default to 0, but it’s always nice to be explicit.
4.1.2 Functions / Macros

Another staple of almost any programming language is the ability to execute functions. However, we can achieve the same effect as (non recursive) functions using the time honored technique of “copy and paste”. That is, we can replace code such as

```python
def Func(a, b):
    function_code
    return c
some_code
f = Func(e, d)
some_more_code

some_code
function_code'
some_more_code
```

where `function_code'` is obtained by replacing all occurrences of `a` with `d`, `b` with `e`, `c` with `f`. When doing that we will need to ensure that all other variables appearing in `function_code` don’t interfere with other variables by replacing every instance of a variable `foo` with `upfoo` where `up` is some unique prefix.

4.1.3 Example: Computing Majority via NAND’s

Function definition allow us to express NAND programs much more cleanly and succinctly. For example, because we can compute AND, OR, NOT using NANDs, we can compute the Majority function as well.

```python
def NOT(a): return NAND(a, a)
def AND(a, b): return NOT(NAND(a, b))
def OR(a, b): return NAND(NOT(a), NOT(b))

def MAJ(a, b, c):
    return OR(OR(AND(a, b), AND(b, c)), AND(a, c))

print(MAJ(0, 1, 1))
# 1
```

This is certainly much more pleasant than the full NAND alternative:

```python
Temp[0] = NAND(X[0], X[1])
Temp[1] = NAND(Temp[0], Temp[0])
Temp[2] = NAND(X[1], X[2])
Temp[3] = NAND(Temp[2], Temp[2])
```
Temp[4] = NAND(Temp[1], Temp[1])
Temp[5] = NAND(Temp[3], Temp[3])
Temp[6] = NAND(Temp[4], Temp[5])
Temp[7] = NAND(X[0], X[2])
Temp[8] = NAND(Temp[7], Temp[7])
Temp[9] = NAND(Temp[6], Temp[6])
Temp[10] = NAND(Temp[8], Temp[8])
Y[0] = NAND(Temp[9], Temp[10])

4.1.4 Conditional statements
Another sorely missing feature in NAND is a conditional statement such as the if/then constructs that are found in many programming languages. However, using functions, we can obtain an ersatz if/then construct. First we can compute the function $IF : \{0,1\}^3 \to \{0,1\}$ such that $IF(a, b, c)$ equals $b$ if $a = 1$ and $c$ if $a = 0$.

```python
def IF(cond,a,b):
    notcond = NAND(cond,cond)
    temp = NAND(b,notcond)
    temp1 = NAND(a,cond)
    return NAND(temp,temp1)
```

```
print(IF(0,1,0))
# 0
print(IF(1,1,0))
# 1
```

The $IF$ function is also known as the multiplexing function, since $cond$ can be thought of as a switch that controls whether the output is connected to $a$ or $b$. We leave it as Exercise 4.2 to verify that this program does indeed compute this function.

Using the $IF$ function, we can implement conditionals in NAND:

```python
if (cond):
    a = ...
    b = ...
    c = ...
```

we can use code of the following form
a = IF(cond, ..., a)
b = IF(cond, ..., b)
c = IF(cond, ..., c)

or even

a, b, c = IF(cond, ...., a, b, c)

using an extension of the IF function to more inputs and outputs.

4.1.5 Bounded loops
We can use “copy paste” to implement a bounded variant of loops, as long we only need to repeat the loop a fixed number of times. For example, we can use code such as:

```
for i in [7, 9, 12]:
    Foo[i] = NAND(Bar[2*i], Blah[3*i+1])
```

as shorthand for

```
Foo[7]  = NAND(Bar[14], Blah[22])
Foo[9]  = NAND(Bar[18], Blah[28])
Foo[12] = NAND(Bar[24], Blah[37])
```

One can also consider fancier versions, including inner loops and so on. The crucial point is that (unlike most programming languages) we do not allow the number of times the loop is executed to depend on the input, and so it is always possible to “expand out” the loop by simply copying the code the requisite number of times. We will use standard Python syntax such as `range(n)` for the sets we can range over.

4.1.6 Example: Adding two integers
Using the above features, we can write the integer addition function as follows:

```
# Add two n-bit integers
def ADD(A,B):
    n = len(A)
    Result = [0]*(n+1)
    Carry = [0]*(n+1)
    Carry[0] = zero(A[0])
    for i in range(n):
        Result[i] = XOR(Carry[i], XOR(A[i], B[i]))
        Carry[i+1] = MAJ(Carry[i], A[i], B[i])
    Result[n] = Carry[n]
    return Result
```
ADD([1,1,1,0],[1,0,0,0])
# [0, 0, 0, 1, 0, 0]

where zero is the constant zero function, and MAJ and XOR correspond to the majority and XOR functions respectively. This “sugared” version is certainly easier to read than even the two bit NAND addition program (obtained by restricting the above to the case \( n = 2 \)):

Temp[0] = NAND(X[0], X[0])
Temp[1] = NAND(X[0], Temp[0])
Temp[2] = NAND(Temp[1], Temp[1])
Temp[3] = NAND(X[0], X[2])
Temp[4] = NAND(X[0], Temp[3])
Temp[5] = NAND(X[2], Temp[3])
Temp[6] = NAND(Temp[4], Temp[5])
Temp[7] = NAND(Temp[2], Temp[6])
Temp[8] = NAND(Temp[2], Temp[7])
Temp[9] = NAND(Temp[6], Temp[7])
Y[0] = NAND(Temp[8], Temp[9])
Temp[11] = NAND(Temp[2], X[0])
Temp[13] = NAND(X[0], X[2])
Temp[14] = NAND(Temp[13], Temp[13])
Temp[15] = NAND(Temp[12], Temp[12])
Temp[16] = NAND(Temp[14], Temp[14])
Temp[17] = NAND(Temp[15], Temp[16])
Temp[18] = NAND(Temp[2], X[2])
Temp[19] = NAND(Temp[18], Temp[18])
Temp[20] = NAND(Temp[17], Temp[17])
Temp[21] = NAND(Temp[19], Temp[19])
Temp[22] = NAND(Temp[20], Temp[21])
Temp[23] = NAND(X[1], X[3])
Temp[24] = NAND(X[1], Temp[23])
Temp[25] = NAND(X[3], Temp[23])
Temp[26] = NAND(Temp[24], Temp[25])
Temp[27] = NAND(Temp[22], Temp[26])
Temp[28] = NAND(Temp[22], Temp[27])
Temp[29] = NAND(Temp[26], Temp[27])
Y[1] = NAND(Temp[28], Temp[29])
Temp[31] = NAND(Temp[22], X[1])
Temp[32] = NAND(Temp[31], Temp[31])
Temp[33] = NAND(X[1], X[3])
Temp[34] = NAND(Temp[33], Temp[33])
Temp[35] = NAND(Temp[32], Temp[32])
Temp[36] = NAND(Temp[34], Temp[34])
Temp[37] = NAND(Temp[35], Temp[36])
Temp[38] = NAND(Temp[22], X[3])
Temp[39] = NAND(Temp[38], Temp[38])
Temp[40] = NAND(Temp[37], Temp[37])
Temp[41] = NAND(Temp[39], Temp[39])
Y[2] = NAND(Temp[40], Temp[41])

Which corresponds to the following circuit:
4.2 EVEN MORE SUGAR (OPTIONAL)

We can go even beyond this, and add more “syntactic sugar” to NAND. The key observation is that all of these are not extra features to NAND, but only ways that make it easier for us to write programs.

4.2.1 More indices

As stated, the NAND programming language only allows for “one dimensional arrays”, in the sense that we can use variables such as Foo[7] or Foo[29] but not Foo[5][15]. However we can easily embed two dimensional arrays in one-dimensional ones using a one-to-one function \( PAIR : \mathbb{N}^2 \to \mathbb{N} \). (For example, we can use \( PAIR(x, y) = 2^x3^y \), but there are also more efficient embeddings, see Exercise 4.1.) Hence we can replace any variable of the form Foo\([i][j]\) with foo\([PAIR(i, j)]\), and similarly for three dimensional arrays.

4.2.2 Non-Boolean variables, lists and integers

While the basic variables in NAND++ are Boolean (only have 0 or 1), we can easily extend this to other objects using encodings. For example, we can encode the alphabet \( \{a, b, c, d, e, f\} \) using three bits as 000, 001, 010, 011, 100, 101. Hence, given such an encoding, we could use the code

\[
\text{Foo} = \text{REPRES}("b")
\]

would be a shorthand for the program

\[
\text{Foo}[0] = \text{zero}(); \\
\text{Foo}[1] = \text{zero}(); \\
\text{Foo}[2] = \text{one}();
\]

(Where we use the constant functions \text{zero} and \text{one}, which we can apply to any variable.) Using our notion of multi-indexed arrays, we can also use code such as

\[
\text{Foo} = \text{COPY}("be")
\]

as a shorthand for

\[
\text{Foo}[0][0] = \text{zero}(); \\
\text{Foo}[0][1] = \text{one}(); \\
\text{Foo}[0][2] = \text{one}(); \\
\text{Foo}[1][0] = \text{one}(); \\
\text{Foo}[1][1] = \text{zero}(); \\
\text{Foo}[1][2] = \text{zero}();
\]

which can then in turn be mapped to standard NAND code using a one-to-one embedding \( pair : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \) as above.
4.2.3 Storing integers

We can also handle non-finite alphabets, such as integers, by using some prefix-free encoding and encoding the integer in an array. For example, to store non-negative integers, we can use the convention that 01 stands for 0, 11 stands for 1, and 00 is the end marker. To store integers that could be potentially negative we can use the convention 10 in the first coordinate stands for the negative sign.\(^3\) So, code such as

\[
\text{Foo} = \text{REPRES}(5) \quad \# (1,0,1) \text{ in binary}
\]

will be shorthand for

\[
\begin{align*}
\text{Foo}[0] &= \text{one}(). \\
\text{Foo}[1] &= \text{one}(). \\
\text{Foo}[2] &= \text{zero}(). \\
\text{Foo}[3] &= \text{one}(). \\
\text{Foo}[4] &= \text{one}(). \\
\text{Foo}[5] &= \text{one}(). \\
\text{Foo}[6] &= \text{zero}(). \\
\text{Foo}[7] &= \text{zero}().
\end{align*}
\]

Using multidimensional arrays, we can use arrays of integers and hence replace code such as

\[
\text{Foo} = \text{REPRES}([12,7,19,33])
\]

with the equivalent NAND expressions.

4.2.4 Example: Multiplying \(n\) bit numbers

We have seen in Section 4.1.6 how to use the grade-school algorithm to show that NAND programs can add \(n\)-bit numbers for every \(n\). By following through this example, we can obtain the following result

---

**Theorem 4.1 — Addition using NAND programs.** For every \(n\), let 

\[
ADD_n : \{0,1\}^{2n} \rightarrow \{0,1\}^{n+1}
\]

be the function that, given \(x, x' \in \{0,1\}^n\) computes the representation of the sum of the numbers that \(x\) and \(x'\) represent. Then there is a NAND program that computes the function \(ADD_n\). Moreover, the number of lines in this program is smaller than \(100n\).

---

We omit the full formal proof of Theorem 4.1, but it can be obtained by going through the code in Section 4.1.6 and:

1. Proving that for every \(n\), this code does indeed compute the addition of two \(n\) bit numbers.
2. Proving that for every $n$, if we expand the code out to its “unsweetened” version (i.e., to a standard NAND program), then the number of lines will be at most $100n$.

See Fig. 4.1 for a figure illustrating the number of lines our program has as a function of $n$. It turns out that this implementation of $ADD_n$ uses about $13n$ lines.

![Figure 4.1](image)

**Figure 4.1**: The number of lines in our NAND program to add two $n$ bit numbers, as a function of $n$, for $n$’s between 1 and 100.

Once we have addition, we can use the grade-school algorithm to obtain multiplication as well, thus obtaining the following theorem:

**Theorem 4.2 — Multiplication NAND programs.** For every $n$, let $MULT_n : \{0,1\}^{2n} \rightarrow \{0,1\}^n$ be the function that, given $x, x' \in \{0,1\}^n$ computes the representation of the product of the numbers that $x$ and $x'$ represent. Then there is a NAND program that computes the function $MULT_n$. Moreover, the number of lines in this program is smaller than $1000n^2$.

We omit the proof, though in Exercise 4.6 we ask you to supply a “constructive proof” in the form of a program (in your favorite programming language) that on input a number $n$, outputs the code of a NAND program of at most $1000n^2$ lines that computes the $MULT_n$ function. In fact, we can use Karatsuba’s algorithm to show that there is a NAND program of $O(n \log_2 3)$ lines to compute $MULT_n$ (and one can even get further asymptotic improvements using the newer algorithms).
4.3 FUNCTIONS BEYOND ARITHMETIC AND LOOKUP

We have seen that NAND programs can add and multiply numbers. But can they compute other type of functions, that have nothing to do with arithmetic? Here is one example:

**Definition 4.3 — Lookup function.** For every \( k \), the lookup function \( \text{LOOKUP}_k : \{0, 1\}^{2^k} \to \{0, 1\} \) is defined as follows: For every \( x \in \{0, 1\}^{2^k} \) and \( i \in \{0, 1\}^k \),

\[
\text{LOOKUP}_k(x, i) = x_i
\]

(4.1)

where \( x_i \) denotes the \( i \)th entry of \( x \), using the binary representation to identify \( i \) with a number in \( \{0, \ldots, 2^k - 1\} \).

The function \( \text{LOOKUP}_1 : \{0, 1\}^3 \to \{0, 1\} \) maps \( (x_0, x_1, i) \in \{0, 1\}^3 \) to \( x_{i_1} \). It is actually the same as the \( IF/MUX \) function we have seen above, that has a 4 line NAND program. However, can we compute higher levels of \( \text{LOOKUP} \)? This turns out to be the case:

**Theorem 4.4 — Lookup function.** For every \( k \), there is a NAND program that computes the function \( \text{LOOKUP}_k : \{0, 1\}^{2^k} \to \{0, 1\} \). Moreover, the number of lines in this program is at most \( 4 \cdot 2^k \).

### 4.3.1 Constructing a NAND program for \( \text{LOOKUP} \)

We now prove Theorem 4.4. We will do so by induction. That is, we show how to use a NAND program for computing \( \text{LOOKUP}_k \) to compute \( \text{LOOKUP}_{k+1} \). Let us first see how we do this for \( \text{LOOKUP}_2 \). Given input \( x = (x_0, x_1, x_2, x_3) \) and an index \( i = (i_0, i_1) \), if the most significant bit \( i_1 \) of the index is 0 then \( \text{LOOKUP}_2(x, i) \) will equal \( x_0 \) if \( i_0 = 0 \) and equal \( x_1 \) if \( i_0 = 1 \). Similarly, if the most significant bit \( i_1 \) is 1 then \( \text{LOOKUP}_2(x, i) \) will equal \( x_2 \) if \( i_0 = 0 \) and will equal \( x_3 \) if \( i_0 = 1 \). Another way to say this is that

\[
\text{LOOKUP}_2(x_0, x_1, x_2, x_3, i_0, i_1) = \text{LOOKUP}_1(\text{LOOKUP}_1(x_0, x_1, i_0), \text{LOOKUP}_1(x_2, x_3, i_0, i_1))
\]

(4.2)

That is, we can compute \( \text{LOOKUP}_2 \) using three invocations of \( \text{LOOKUP}_1 \). The “pseudocode” for this program will be

\[
\begin{align*}
Z[0] &= \text{LOOKUP}_1(X[0], X[1], X[4]) \\
Z[1] &= \text{LOOKUP}_1(X[0], X[1], X[4]) \\
Z[2] &= \text{LOOKUP}_1(X[2], X[3], X[4]) \\
Y[0] &= \text{LOOKUP}_1(Z[0], Z[1], X[5])
\end{align*}
\]

(Note that since we call this function with \( (x_0, x_1, x_2, x_3, i_0, i_1) \), the inputs \( x_4 \) and \( x_5 \) correspond to \( i_0 \) and \( i_1 \).) We can obtain an actual
“sugar free” NAND program of at most 12 lines by replacing the calls to `LOOKUP_1` by an appropriate copy of the program above.

We can generalize this to compute `LOOKUP_3` using two invocations of `LOOKUP_2` and one invocation of `LOOKUP_1`. That is, given input \( x = (x_0, \ldots, x_7) \) and \( i = (i_0, i_1, i_2) \) for `LOOKUP_3`, if the most significant bit of the index \( i_2 \) is 0, then the output of `LOOKUP_3` will equal `LOOKUP_2(x_0, x_1, x_2, i_0, i_1)`, while if this index \( i_2 \) is 1 then the output will be `LOOKUP_2(x_4, x_5, x_6, i_0, i_1)`, meaning that the following pseudocode can compute `LOOKUP_3`,

\[
\begin{align*}
Z[0] &= \text{LOOKUP}_2(x[0], x[1], x[2], x[3], x[8], x[9]) \\
Z[1] &= \text{LOOKUP}_2(x[4], x[5], x[6], x[7], x[8], x[9]) \\
Y[0] &= \text{LOOKUP}_1(Z[0], Z[1], x[10])
\end{align*}
\]

where again we can replace the calls to `LOOKUP_2` and `LOOKUP_1` by invocations of the process above.

Formally, we can prove the following lemma:

**Lemma 4.5 — Lookup recursion.** For every \( k \geq 2 \), `LOOKUP_k(x_0, \ldots, x_{2^k - 1}, i_0, \ldots, i_{k-1})` is equal to

\[
\begin{align*}
\text{LOOKUP}_1(\text{LOOKUP}_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-2}), \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-2}), i_{k-1})
\end{align*}
\] (4.3)

**Proof.** If the most significant bit \( i_{k-1} \) of \( i \) is zero, then the index \( i \) is in \( \{0, \ldots, 2^{k-1} - 1\} \) and hence we can perform the lookup on the “first half” of \( x \) and the result of `LOOKUP_k(x, i)` will be the same as \( \text{LOOKUP}_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_0, \ldots, i_{k-1}) \). On the other hand, if this most significant bit \( i_{k-1} \) is equal to 1, then the index is in \( \{2^{k-1}, \ldots, 2^k - 1\} \), in which case the result of `LOOKUP_k(x, i)` is the same as \( b = \text{LOOKUP}_{k-1}(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1}) \). Thus we can compute `LOOKUP_k(x, i)` by first computing \( a \) and \( b \) and then outputting `LOOKUP_1(a, b, i_{k-1})`.■

**Lemma 4.5** directly implies **Theorem 4.4**. We prove by induction on \( k \) that there is a NAND program of at most \( 4 \cdot 2^k \) lines for `LOOKUP_k`. For \( k = 1 \) this follows by the four line program for `LOOKUP_1` we’ve seen before. For \( k > 1 \), we use the following pseudocode

\[
\begin{align*}
a &= \text{LOOKUP}_1(k-1)(x[0], \ldots, x[2^{k-1} - 1], i[0], \ldots, i[k-2]) \\
b &= \text{LOOKUP}_1(k-1)(x[2^{k-1}], \ldots, z[2^{k-1} - 1], i[0], \ldots, i[k-2]) \\
y_0 &= \text{LOOKUP}_1(a, b, i[k-1])
\end{align*}
\]

In Python, this can be described as follows
```python
def LOOKUP(X, i):
    k = len(i)
    if k==1: return IF(i[0], X[1], X[0])
    return IF(i[k-1], LOOKUP(X[2**k-1:], i[1:-1]), LOOKUP(X[2**k-1:], i[1:-1]))
```

If we let \( L(k) \) be the number of lines required for \( LOOKUP_k \), then the above shows that

\[
L(k) \leq 2L(k-1) + 4.
\]  
(4.4)

We will prove by induction that \( L(k) \leq 4(2^k - 1) \). This is true for \( k = 1 \) by our construction. For \( k > 1 \), using the inductive hypothesis and Eq. (4.4), we get that

\[
L(k) \leq 2 \cdot 4 \cdot (2^{k-1} - 1) + 4 = 4 \cdot 2^k - 8 + 4 = 4(2^k - 1)
\]  
(4.5)

completing the proof of Theorem 4.4. (See Fig. 4.2 for a plot of the actual number of lines in our implementation of \( LOOKUP_k \).

![Figure 4.2](image)

Figure 4.2: The number of lines in our implementation of the `LOOKUP_k` function as a function of \( k \) (i.e., the length of the index). The number of lines in our implementation is roughly \( 3 \cdot 2^k \).

### 4.4 Computing Every Function

At this point we know the following facts about NAND programs:

1. They can compute at least some non trivial functions.
2. Coming up with NAND programs for various functions is a very tedious task.
Thus I would not blame the reader if they were not particularly looking forward to a long sequence of examples of functions that can be computed by NAND programs. However, it turns out we are not going to need this, as we can show in one fell swoop that NAND programs can compute every finite function:

**Theorem 4.6 — Universality of NAND.** For every \( n, m \) and function \( F : \{0, 1\}^n \to \{0, 1\}^m \), there is a NAND program that computes the function \( F \). Moreover, there is such a program with at most \( O(m2^n) \) lines.

The implicit constant in the \( O(\cdot) \) notation can be shown to be at most 10. We also note that the bound of Theorem 4.6 can be improved to \( O(m2^n / n) \), see Section 4.4.2.

### 4.4.1 Proof of NAND’s Universality

To prove Theorem 4.6, we need to give a NAND program for every possible function. We will restrict our attention to the case of Boolean functions (i.e., \( m = 1 \)). In Exercise 4.8 you will show how to extend the proof for all values of \( m \). A function \( F : \{0, 1\}^n \to \{0, 1\} \) can be specified by a table of its values for each one of the \( 2^n \) inputs. For example, the table below describes one particular function \( G : \{0, 1\}^4 \to \{0, 1\} \):

<table>
<thead>
<tr>
<th>Input (( x ))</th>
<th>Output (( G(x) ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>

We can see that for every \( x \in \{0, 1\}^4 \), \( G(x) = LOOKUP_t(1100100100001111, x) \). Therefore the following is NAND “pseudocode” to compute \( G \):

\[
G0000 = 1
\]
G1000 = 1
G0100 = 0
G1100 = 0
G0010 = 1
G1010 = 0
G0110 = 0
G1110 = 1
G0001 = 0
G1001 = 0
G0101 = 0
G1101 = 0
G0011 = 1
G1011 = 1
G1111 = 1
Y[0] = \text{LOOKUP}(G0000, G1000, G0100, G1100, G0010, 
G1010, G0110, G1110, G0001, G1001, 
G0101, G1101, G0011, G1011, G1111, 
X[0], X[1], X[2], X[3])

Recall that we can translate this pseudocode into an actual NAND program by adding three lines to define variables \texttt{zero} and \texttt{one} that are initialized to 0 and 1 respectively, and then replacing a statement such as \texttt{Gxxx = 0} with \texttt{Gxxx = NAND}(\texttt{one}, \texttt{one}) and a statement such as \texttt{Gxxx = 1} with \texttt{Gxxx = NAND}(\texttt{zero}, \texttt{zero}). The call to \texttt{LOOKUP} will be replaced by the NAND program that computes $\mathcal{G}$, but we will replace the variables $X[16], \ldots, X[19]$ in this program with $X[0], \ldots, X[3]$ and the variables $X[0], \ldots, X[15]$ with $G000, \ldots, G1111$.

There was nothing about the above reasoning that was particular to this program. Given every function $F : \{0,1\}^n \rightarrow \{0,1\}$, we can write a NAND program that does the following:

1. Initialize $2^n$ variables of the form $F00\ldots0$ till $F11\ldots1$ so that for every $z \in \{0,1\}^n$, the variable corresponding to $z$ is assigned the value $F(z)$.

2. Compute $\text{LOOKUP}_n$ on the $2^n$ variables initialized in the previous step, with the index variable being the input variables $X[0]$ $\ldots$, $X[2^n - 1]$. That is, just like in the pseudocode for $G$ above, we use $Y[0] = \text{LOOKUP}(F00\ldots0, F10\ldots0, \ldots, F11\ldots1, X[0], \ldots, x[(n-1)])$

The total number of lines in the program will be $2^n$ plus the $4 \cdot 2^n$ lines that we pay for computing $\text{LOOKUP}_n$. This completes the proof of Theorem 4.6.
The NAND programming language website allows you to construct a NAND program for an arbitrary function.

**Result in perspective** While Theorem 4.6 seems striking at first, in retrospect, it is perhaps not that surprising that every finite function can be computed with a NAND program. After all, a finite function \( f : \{0,1\}^n \rightarrow \{0,1\}^m \) can be represented by simply the list of its outputs for each one of the \( 2^n \) input values. So it makes sense that we could write a NAND program of similar size to compute it. What is more interesting is that some functions, such as addition and multiplication, have a much more efficient representation: one that only requires \( \Omega(2^n) \) or even smaller number of lines.

### 4.4.2 Improving by a factor of \( n \) (optional)

By being a little more careful, we can improve the bound of Theorem 4.6 and show that every function \( F : \{0,1\}^n \rightarrow \{0,1\}^m \) can be computed by a NAND program of at most \( O(m2^n/n) \) lines. As before, it is enough to prove the case that \( m = 1 \). The idea is to use the technique known as memoization. Let \( k = \log(n - 2 \log n) \) (the reasoning behind this choice will become clear later on). For every \( a \in \{0,1\}^{n-k} \) we define \( F_a : \{0,1\}^k \rightarrow \{0,1\} \) to be the function that maps \( w_0, \ldots, w_{k-1} \) to \( F(a_0, \ldots, a_{n-k-1}, w_0, \ldots, w_{k-1}) \). On input \( x = x_0, \ldots, x_{n-1} \), we can compute \( F(x) \) as follows: First we compute a \( 2^{n-k} \) long string \( \mathcal{S} \) whose \( x \) entry (identifying \( \{0,1\}^n \) with \( [2^n] \)) equals \( F_a(x_0, \ldots, x_{n-1}) \). On input \( x = x_0, \ldots, x_{n-1} \), we can compute \( \mathcal{S} \) using \( \Omega(2^{n-k}) \) lines. One can verify that \( F(x) = \texttt{LOOKUP}_{n-k}(P, x_0, \ldots, x_{n-1}) \).

If we can compute \( \texttt{LOOKUP}_{n-k} \) using \( O(2^{n-k}) \) lines, if we can compute the string \( P \) (i.e., compute variables \( P_{\cdot}(0), \ldots, P_{\cdot}(2^{n-k} - 1) \)) using \( T \) lines, then we can compute \( F \) in \( O(2^{n-k}) + T \) lines. The trivial way to compute the string \( P \) would be to use \( O(2^k) \) lines to compute for every \( a \) the map \( x_0, \ldots, x_{k-1} \mapsto F_a(x_0, \ldots, x_{k-1}) \) as in the proof of Theorem 4.6. Since there are \( 2^{n-k} a' \)’s, that would be a total cost of \( O(2^{n-k} \cdot 2^k) = O(2^n) \) which would not improve at all on the bound of Theorem 4.6. However, a more careful observation shows that we are making some redundant computations. After all, there are only \( 2^k \) distinct functions mapping \( k \) bits to one bit. If \( a \) and \( a' \) satisfy that \( F_a = F_{a'} \), then we don’t need to spend \( 2^k \) lines computing both \( F_a(x) \) and \( F_{a'}(x) \) but rather can only compute the variable \( P_{\cdot}(a) \) and then copy \( P_{\cdot}(a) \) to \( P_{\cdot}(a') \) using \( O(1) \) lines. Since we have \( 2^{2k} \) unique functions, we can bound the total cost to compute \( P \) by \( O(2^{2k}2^k) + O(2^{n-k}) \). Now it just becomes a matter of calculation. By our choice of \( k \), \( 2^k = n - 2 \log n \) and hence \( 2^{2k} = \frac{2n}{n} \). Since \( n/2 \leq 2^k \leq n \), we can bound the total cost of computing \( F(x) \) (inclu-
ing also the additional $O(2^{n-k})$ cost of computing $LOOKUP_{n-k}$) by $O(\frac{2^n}{n^2} \cdot n) + O(2^n/n)$, which is what we wanted to prove.

### 4.4.3 The class $SIZE_{n,m}(T)$

For every $n, m, T \in \mathbb{N}$, we denote by $SIZE_{n,m}(T)$, the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$ that can be computed by NAND programs of at most $T$ lines. Theorem 4.6 shows that $SIZE_{n,m}(4m2^n)$ is the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$. The results we’ve seen before can be phrased as showing that $ADD_n \in SIZE_{2n,n+1}(100n)$ and $MULT_n \in SIZE_{2n,2n}(10000n\log_2 n)$. See Fig. 4.3.

Note that $SIZE_{n,m}(T)$ does not correspond to a set of programs! Rather, it is a set of functions. This distinction between programs and functions will be crucial for us in this course. You should always remember that while a program computes a function, it is not equal to a function. In particular, as we’ve seen, there can be more than one program to compute the same function.

**Figure 4.3**: A rough illustration of the relations between the different classes of functions computed by NAND programs of given size. For every $n, m$, the class $SIZE_{n,m}(T)$ is a subset of the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$, and if $T \leq T'$ then $SIZE_{n,m}(T) \subseteq SIZE_{n,m}(T')$. Theorem 4.6 shows that $SIZE_{n,m}(O(m \cdot 2^n))$ is equal to the set of all functions, and using Section 4.4.2 this can be improved to $O(m \cdot 2^n/n)$. If we consider all functions mapping $n$ bits to $n$ bits, then addition of two $n/2$ bit numbers can be done in $O(n)$ lines, while we don’t know of such a program for multiplying two $n$ bit numbers, though we do know it can be done in $O(n^2)$ and in fact even better size. In the above $FACTOR_n$ corresponds to the inverse problem of multiplying - finding the prime factorization of a given number.

At the moment we do not know of any NAND program with a polynomial (or even sub-exponential) number of lines that can compute $FACTOR_n$.

**Finite vs infinite functions** A NAND program $P$ can only compute a function with a certain number $n$ of inputs and a certain number $m$ of outputs.
Hence for example there is no single NAND program that can compute the increment function $\text{INC} : \{0,1\}^* \to \{0,1\}^*$ that maps a string $x$ (which we identify with a number via the binary representation) to the string that represents $x + 1$. Rather for every $n > 0$, there is a NAND program $P_n$ that computes the restriction $\text{INC}_n$ of the function $\text{INC}$ to inputs of length $n$. Since it can be shown that for every $n > 0$ such a program $P_n$ exists of length at most $10n$, $\text{INC}_n \in \text{SIZE}(10n)$ for every $n > 0$.

If $T : \mathbb{N} \to \mathbb{N}$ and $F : \{0,1\}^* \to \{0,1\}$, we will sometimes slightly abuse notation and write $F \in \text{SIZE}(T(n))$ to indicate that for every $n$ the restriction $F_n$ of $F$ to inputs in $\{0,1\}^n$ is in $\text{SIZE}(T(n))$. Hence we can write $\text{INC} \in \text{SIZE}(10n)$. We will come back to this issue of finite vs infinite functions later in this course.

Solved Exercise 4.1 — $\text{SIZE}$ closed under complement. In this exercise we prove a certain “closure property” of the class $\text{SIZE}(T(n))$. That is, we show that if $f$ is in this class then (up to some small additive term) so is the complement of $f$, which is the function $g(x) = 1 - f(x)$.

Prove that there is a constant $c$ such that for every $f : \{0,1\}^n \to \{0,1\}$ and $s \in \mathbb{N}$, if $f \in \text{SIZE}(s)$ then $1 - f \in \text{SIZE}(s + c)$.

Solution: If $f \in \text{SIZE}(s)$ then there is an $s$-line program $P$ that computes $f$. We can rename the variable $Y[0]$ in $P$ to a unique variable $\text{unique\_temp}$ and add the line

$$Y[0] = \text{NAND}(\text{unique\_temp, unique\_temp})$$

at the very end to obtain a program $P'$ that computes $1 - f$.

Lecture Recap

- We can define the notion of computing a function via a simplified “programming language”, where computing a function $F$ in $T$ steps would correspond to having a $T$-line NAND program that computes $F$.
- While the NAND programming only has one operation, other operations such as functions and conditional execution can be implemented using it.
- Every function $F : \{0,1\}^n \to \{0,1\}^m$ can be computed by a NAND program of at most $O(m2^n)$ lines (and in fact at most $O(m2^n/n)$ lines).
- Sometimes (or maybe always?) we can translate an efficient algorithm to compute $F$ into a NAND
program that computes \( F \) with a number of lines comparable to the number of steps in this algorithm.

4.5 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 4.1 — Pairing. 1. Prove that the map \( F(x, y) = 2^x 3^y \) is a one-to-one map from \( \mathbb{N}^2 \) to \( \mathbb{N} \).

2. Show that there is a one-to-one map \( F : \mathbb{N}^2 \to \mathbb{N} \) such that for every \( x, y \), \( F(x, y) \leq 100 \cdot \max\{x, y\}^2 + 100 \).

3. For every \( k \), show that there is a one-to-one map \( F : \mathbb{N}^k \to \mathbb{N} \) such that for every \( x_0, \ldots, x_{k-1} \in \mathbb{N} \), \( F(x_0, \ldots, x_{k-1}) \leq 100 \cdot (x_0 + x_1 + \ldots + x_{k-1} + 100k)^{\alpha} \).

Exercise 4.2 — Computing MUX. Prove that the NAND program below computes the function \( MUX \) (or \( LOOKUP_1 \)) where \( MUX(a, b, c) \) equals \( a \) if \( c = 0 \) and equals \( b \) if \( c = 1 \):

\[
\begin{align*}
t &= \text{NAND}(X[2], X[2]) \\
u &= \text{NAND}(X[0], t) \\
v &= \text{NAND}(X[1], X[2]) \\
Y[0] &= \text{NAND}(u, v)
\end{align*}
\]

Exercise 4.3 — At least two / Majority. Give a NAND program of at most 6 lines to compute \( MAJ : \{0, 1\}^3 \to \{0, 1\} \) where \( MAJ(a, b, c) = 1 \) iff \( a + b + c \geq 2 \).

Exercise 4.4 — Conditional statements. In this exercise we will show that even though the NAND programming language does not have an \textbf{if} . . then . . \textbf{else} . . statement, we can still implement it. Suppose that there is an \( s \)-line NAND program to compute \( F : \{0, 1\}^n \to \{0, 1\} \) and an \( s' \)-line NAND program to compute \( F' : \{0, 1\}^n \to \{0, 1\} \).

Prove that there is a program of at most \( s + s' + 10 \) lines to compute the function \( G : \{0, 1\}^{n+1} \to \{0, 1\} \) where \( G(x_0, \ldots, x_{n-1}, x_n) \) equals \( F(x_0, \ldots, x_{n-1}) \) if \( x_n = 0 \) and equals \( F'(x_0, \ldots, x_{n-1}) \) otherwise.
Exercise 4.5 — Addition. Write a program using your favorite programming language that on input an integer $n$, outputs a NAND program that computes $ADD_n$. Can you ensure that the program it outputs for $ADD_n$ has fewer than $10n$ lines?

Exercise 4.6 — Multiplication. Write a program using your favorite programming language that on input an integer $n$, outputs a NAND program that computes $MULT_n$. Can you ensure that the program it outputs for $MULT_n$ has fewer than $1000 \cdot n^2$ lines?

Exercise 4.7 — Efficient multiplication (challenge). Write a program using your favorite programming language that on input an integer $n$, outputs a NAND program that computes $MULT_n$ and has at most $10000n^{1.9}$ lines. What is the smallest number of lines you can use to multiply two 2048 bit numbers?

Exercise 4.8 — Multibit function. Prove that

a. If there is an $s$-line NAND program to compute $F : \{0,1\}^n \rightarrow \{0,1\}$ and an $s'$-line NAND program to compute $F' : \{0,1\}^n \rightarrow \{0,1\}$, then there is an $s + s'$-line program to compute the function $G : \{0,1\}^n \rightarrow \{0,1\}^2$ such that $G(x) = (F(x), F'(x))$.

b. For every function $F : \{0,1\}^n \rightarrow \{0,1\}^m$, there is a NAND program of at most $10m \cdot 2^n$ lines that computes $F$.

4.6 BIBLIOGRAPHICAL NOTES

4.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

(to be completed)
5

Code as data, data as code

“A mathematician would hardly call a correspondence between the set of 64 triples of four units and a set of twenty other units, “universal”, while such correspondence is, probably, the most fundamental general feature of life on Earth”, Misha Gromov, 2013

A NAND program can be thought of as simply a sequence of symbols, each of which can be encoded with zeros and ones using (for example) the ASCII standard. Thus we can represent every NAND program as a binary string. This statement seems obvious but it is actually quite profound. It means that we can treat a NAND program both as instructions to carrying computation and also as data that could potentially be input to other computations.

This correspondence between code and data is one of the most fundamental aspects of computing. It underlies the notion of general purpose computers, that are not pre-wired to compute only one task, and it is also the basis of our hope for obtaining general artificial intelligence. This concept finds immense use in all areas of computing, from scripting languages to machine learning, but it is fair to say that we haven’t yet fully mastered it. Indeed many security exploits involve cases such as “buffer overflows” when attackers manage to inject code where the system expected only “passive” data (see Fig. 5.1). The idea of code as data reaches beyond the realm of electronic computers. For example, DNA can be thought of as both a program and data (in the...
words of Schrödinger, who wrote before DNA’s discovery a book that inspired Watson and Crick, it is both “architect’s plan and builder’s craft”).

Figure 5.1: As illustrated in this xkcd cartoon, many exploits, including buffer overflow, SQL injections, and more, utilize the blurry line between “active programs” and “static strings”.

5.1 A NAND INTERPRETER IN NAND

For every NAND program \( P \), we can represent \( P \) as a binary string. In particular, this means that for any choice of such representation, the following is a well defined mathematical function \( EVAL : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}^* \)

\[
EVAL(P, x) = \begin{cases} 
P(x) & |x| = \text{no. of } P\text{'s inputs} \\
0 & \text{otherwise}
\end{cases}
\]  

(5.1)

where we denote by \( P(x) \) the output of the program represented by the string \( P \) on the input \( x \).

The above is one of those observations that are simultaneously both simple and profound. Please make sure that you understand (1) how for every fixed choice of representing programs as strings, the function \( EVAL \) above is well defined, and (2) what this function actually does.

\( EVAL \) takes strings arbitrarily of length, and hence cannot be computed by a NAND program, that has a fixed length of inputs. However, one of the most interesting consequences of the fact that we can represent programs as strings is the following theorem:

**Theorem 5.1 — Bounded Universality of NAND programs.** For every \( s, n, m \in \mathbb{N} \) there is a NAND program that computes the function

\[
EVAL_{s,n,m} : \{0, 1\}^{S+n} \rightarrow \{0, 1\}^{m}
\]

(5.2)

defined as follows. We let \( S \) be the number of bits that are needed
to represents programs of \( s \) lines. For every string \((P, x)\) where 
\( P \in \{0,1\}^S \) and \( x \in \{0,1\}^n \), if \( P \) describes an \( s \) line NAND program 
with \( n \) input bits and \( m \) outputs bits, then \( EV AL_{s,n,m}(P, x) \) is the 
output of this program on input \( x \). \(^1\)

Of course to fully specify \( EV AL_{s,n,m} \), we need to fix a precise rep-
resentation scheme for NAND programs as binary strings. We can 
simply use the ASCII representation, though below we will choose a 
more convenient representation. But regardless of the choice of rep-
resentation, Theorem 5.1 is an immediate corollary of Theorem 4.6, 
which states that every finite function, and so in particular the function 
\( EV AL_{S,n,m} \) above, can be computed by some NAND program.

Once again, Theorem 5.1 is subtle but important. Make sure you understand what this theorem means, and why it is a corollary of Theorem 4.6.

Theorem 5.1 can be thought of as providing a “NAND interpreter in NAND”. That is, for a particular size bound, we give a single NAND program that can evaluate all NAND programs of that size. We call this NAND program \( U \) that computes \( EV AL_{s,n,m} \) a bounded universal program. “Universal” stands for the fact that this is a single program that can evaluate arbitrary code, where “bounded” stands for the fact that \( U \) only evaluates programs of bounded size. Of course this limitation is inherent for the NAND programming language where an \( N \)-line program can never compute a function with more than \( N \) inputs. (We will later on introduce the concept of loops, that allows to escape this limitation.)

It turns out that we don’t even need to pay that much of an overhead for universality.

Theorem 5.2 — Efficient bounded universality of NAND programs. For 
every \( s, n, m \in \mathbb{N} \) there is a NAND program of at most \( O(s^2 \log s) \) 
lines that computes the function \( EV AL_{S,n,m} : \{0,1\}^{S+n} \to \{0,1\}^m \) 
defined above.

Unlike Theorem 5.1, Theorem 5.2 is not a trivial corollary of the fact that every function can be computed, and takes much more effort to prove. It requires us to present a concrete NAND program for the \( EV AL_{S,n,m} \) function. We will do so in several stages.

1. First, we will spell out precisely how to represent NAND programs as strings. We can prove Theorem 5.2 using the ASCII representation, but a “cleaner” representation will be more convenient for us.

\(^1\) If \( P \) does not describe a program then we don’t care what \( EV AL_{s,n,m}(P, x) \) is. For concreteness you can think of the value as \( 0^m \).
2. Then, we will show how we can write a program to compute $EVL_{s,n,m}$ in Python.\(^2\)

3. Finally, we will show how we can transform this Python program into a NAND program.

### 5.1.1 Concrete representation for NAND programs

![Figure 5.2](image)

In the Harvard Mark I computer, a program was represented as a list of triples of numbers, which were then encoded by perforating holes in a control card.

A NAND program is simply a sequence of lines of the form

```
blah = NAND(baz,boo)
```

There is of course nothing special about these particular identifiers. Hence to represent a NAND program mathematically, we can simply identify the variables with natural numbers, and think of each line as a triple $(i,j,k)$ which corresponds to saying that we assign to the $i$-th variable the NAND of the values of the $j$-th and $k$-th variables.

We will use the set $\{1, 2, \ldots, t\} = \{0, 1, \ldots, t - 1\}$ as our set of variables, and for concreteness we will let the input variables be the first $n$ numbers, and the output variables be the last $m$ numbers (i.e., the numbers $(t-m, \ldots, t-1)$). This motivates the following definition:

**Definition 5.3 — List of tuples representation.** Let $P$ be a NAND program of $n$ inputs, $m$ outputs, and $s$ lines, and let $t$ be the number of
distinct variables used by $P$. The list of tuples representation of $P$ is the triple $(n, m, L)$ where $L$ is a list of triples of the form $(i, j, k)$ for $i, j, k \in [t]$.

For every variable of $P$, we assign a number in $[t]$ as follows:

- For every $i \in [n]$, the variable $X[i]$ is assigned the number $i$.
- For every $j \in [m]$, the variable $Y[j]$ is assigned the number $t - m + j$.
- Every other variable is assigned a number in $\{n, n+1, \ldots, t-m-1\}$ in the order of which it appears.

The list of tuples representation will be our default choice for representing NAND programs, and since “list of tuples representation” is a bit of a mouthful, we will often call this simply the representation for a program $P$.

### Example 5.4 — Representing the XOR program.

Our favorite NAND program, the XOR program:

```python
u = NAND(X[0], X[1])
v = NAND(X[0], u)
w = NAND(X[1], u)
Y[0] = NAND(v, w)
```

Is represented as the tuple $(2, 1, L)$ where $L = ((2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4))$.

That is, the variables $X[0]$ and $X[1]$ are given the indices 0 and 1 respectively, the variables $u, v, w$ are given the indices 2, 3, 4 respectively, and the variable $Y[0]$ is given the index 5.

Transforming a NAND program from its representation as code to the representation as a list of tuples is a fairly straightforward programming exercise, and in particular can be done in a few lines of Python.\(^3\) Note that this representation loses information such as the particular names we used for the variables, but this is OK since these names do not make a difference to the functionality of the program.

\(^3\) If you’re curious what these 15 lines are, see the appendix.

### 5.1.2 Representing a program as a string

To obtain a representation that we can use as input to a NAND program, we need to take a step further and map the triple $(n, m, L)$ to a binary string. Here there are many different choices, but let us fix one of them. If the list $L$ has $s$ triples in it, we will represent it as simply the string $\text{str}(L)$ which will be the concatenation of the $3s$ numbers in the binary basis, which can be encoded as a string of length $3s\ell$ where $\ell = \lceil \log 3s \rceil$ is a number of bits that is guaranteed to be sufficient to represent numbers in $[t]$ (since $t \leq 3s$). We will represent the program $(n, m, L)$ as the string $\langle n \rangle \langle m \rangle \langle s \rangle \text{str}(L)$ where $\langle n \rangle$ and $\langle m \rangle$ are some
prefix-free representations of \(n, m\) and \(s\) (see Section 2.3.2). Hence an \(s\) line program will be represented by a string of length \(O(s \log s)\). In the context of computing \(EVAL_{s,n,m}\) the number of lines, inputs, and outputs, is fixed, and so we can drop \(n, m, s\) and simply think of it as a function that maps \(\{0, 1\}^{3s+\ell+\lambda}\) to \(\{0, 1\}^{m}\), where \(\ell = \lceil \log 3s \rceil\).

5.1.3 A NAND interpreter in “pseudocode”

To prove Theorem 5.2 it suffices to give a NAND program of \(O(s^2 \log s) \leq O((s \log s)^2)\) lines that can evaluate NAND programs of \(s\) lines. Let us start by thinking how we would evaluate such programs if we weren’t restricted to the NAND operations. That is, let us describe informally an algorithm that on input \(n, m, s\), a list of triples \(L\), and a string \(x \in \{0, 1\}^n\), evaluates the program represented by \((n, m, L)\) on the string \(x\).

It would be highly worthwhile for you to stop here and try to solve this problem yourself. For example, you can try thinking how you would write a program \(\text{NANDEVAL}(n, m, s, L, x)\) that computes this function in the programming language of your choice.

Here is a description of such an algorithm:

**Input:** Numbers \(n, m\) and a list \(L\) of \(s\) triples of numbers in \([\mathcal{S}]\) for some \(\mathcal{S} \leq 3s\), as well as a string \(x \in \{0, 1\}^n\).

**Goal:** Evaluate the program represented by \((n, m, L)\) on the input \(x \in \{0, 1\}^n\).

**Operation:**

1. We will create a dictionary data structure \(Vartable\) that for every \(i \in [\mathcal{I}]\) stores a bit. We will assume we have the operations \(\text{GET}(Vartable, i)\) which restore the bit corresponding to \(i\), and the operation \(\text{UPDATE}(Vartable, i, b)\) which update the bit corresponding to \(i\) with the value \(b\). (More concretely, we will write this as \(Vartable = \text{UPDATE}(Vartable, i, b)\) to emphasize the fact that the state of the data structure changes, and to keep our convention of using functions free of “side effects”.)

2. We will initialize the table by setting the \(i\)-th value of \(Vartable\) to \(x_i\) for every \(i \in [n]\).

3. We will go over the list \(L\) in order, and for every triple \((i, j, k)\) in \(L\), we let \(a\) be \(\text{GET}(Vartable, j)\), \(b\) be \(\text{GET}(Vartable, k)\), and then set the value corresponding to \(i\) to the NAND of \(a\) and \(b\). That is, let \(Vartable = \text{UPDATE}(Vartable, i, \text{NAND}(a, b))\).
4. Finally, we output the value $\text{GET}(\text{Vartable}, t - m + j)$ for every $j \in [m]$.

Please make sure you understand this algorithm and why it does produce the right value.

5.1.4 A NAND interpreter in Python
To make things more concrete, let us see how we implement the above algorithm in the *Python* programming language. We will construct a function $\text{NANDEVAL}$ that on input $n, m, L, x$ will output the result of evaluating the program represented by $(n, m, L)$ on $x$.\(^4\) (We will compute the value $s$ to be the size of $L$ and the value $t$ to be the maximum number appearing in $L$ plus one.)

```python
def NANDEVAL(n, m, L, X):
    # Evaluate a NAND program from its list of triple representation.
    s = len(L)  # number of lines
    t = max(max(a, b, c) for (a, b, c) in L) + 1  # maximum index in L + 1

    Vartable = [0] * t  # we'll simply use an array to store data

    def GET(V, i): return V[i]
    def UPDATE(V, i, b):
        V[i] = b
        return V

    # load input values to Vartable:
    for i in range(n): Vartable = UPDATE(Vartable, i, X[i])

    # Run the program
    for (i, j, k) in L:
        a = GET(Vartable, j)
        b = GET(Vartable, k)
        c = NAND(a, b)
        Vartable = UPDATE(Vartable, i, c)

    # Return outputs Vartable[t-m], ...
    return [GET(Vartable, t-m+j) for j in range(m)]
```

\(^4\)To keep things simple, we will not worry about the case that $L$ does not represent a valid program of $n$ inputs and $m$ outputs. Also, there is nothing special about Python. We could have easily presented a corresponding function in JavaScript, C, OCaml, or any other programming language.
# Test on XOR (2 inputs, 1 output)
L = ((2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4))
print(NANDEVAL(2, 1, L, (0, 1)))  # XOR(0,1)  
# [1]
print(NANDEVAL(2, 1, L, (1, 1)))  # XOR(1,1)  
# [0]

Accessing an element of the array Vartable at a given index takes a constant number of basic operations. Hence (since \( n, m \leq s \) and \( t \leq 3s \)), the program above will use \( O(s) \) basic operations.\(^5\)

5.1.5 Constructing the NAND interpreter in NAND
We now turn to describing the proof of Theorem 5.2. To do this, it is of course not enough to give a Python program. Rather, we need to show how we compute the function \( EVAL_{s,n,m} \) by a NAND program. In other words, our job is to transform, for every \( s, n, m \), the Python code above to a NAND program \( \text{U}_{s,n,m} \) that computes the function \( EVAL_{s,n,m} \).

Before reading further, try to think how you could give a “constructive proof” of Theorem 5.2. That is, think of how you would write, in the programming language of your choice, a function universal(s,n,m) that on input \( s, n, m \) outputs the code for the NAND program \( \text{U}_{s,n,m} \) such that \( \text{U}_{s,n,m} \) computes \( EVAL_{s,n,m} \). Note that there is a subtle but crucial difference between this function and the Python NANDEVAL program described above. Rather than actually evaluating a given program \( P \) on some input \( w \), the function universal should output the code of a NAND program that computes the map \( (P,w) \mapsto P(x) \).

Our construction will follow very closely the Python implementation of EVAL above. We will use variables Vartable[0], ..., Vartable[2\( \ell \)-1], where \( \ell = \lceil \log 3s \rceil \) to store our variables. However, NAND doesn’t have integer-valued variables, so we cannot write code such as Vartable[i] for some variable \( i \). However, we can implement the function \( \text{GET} (\text{Vartable}, i) \) that outputs the \( i \)-th bit of the array \( \text{Vartable} \). Indeed, this is nothing by the function \( \text{LOOKUP} \) that we have seen in Theorem 4.4!

Please make sure that you understand why \( \text{GET} \) and \( \text{LOOKUP} \) are the same function.

We saw that we can compute \( \text{LOOKUP} \) on arrays of size \( 2^\ell \) in time...
To compute the update function on input \(V, i, b\), we need to scan the array \(V\), and for \(j \in [2^\ell]\), have our \(j\)-th output be \(V[j]\) unless \(j\) is equal to \(i\), in which case the \(j\)-th output is \(b\). We can do this as follows:

1. For every \(j \in [2^\ell]\), there is an \(O(\ell)\) line NAND program to compute the function \(EQUALS_j : \{0,1\}^{\ell} \to \{0,1\}\) that on input \(i\) outputs 1 if and only if \(i\) is equal to (the binary representation of) \(j\). (We leave verifying this as Exercise 5.2 and Exercise 5.3.)

2. We have seen that we can compute the function \(IF : \{0,1\}^3 \to \{0,1\}\) such that \(IF(a, b, c) = b\) if \(a = 1\) and \(c\) if \(a = 0\).

Together, this means that we can compute \(UPDATE\) as follows:

```python
def UPDATE(V, i, b):
    # update a 2**ell length array at location i to
    # the value b
    for j in range(2**ell):  # j = 0, 1, 2, ..., 2^ell
        a = EQUALS_j(i)
        Y[j] = IF(a, b, V[j])
    return Y
```

Once we can compute \(GET\) and \(UPDATE\), the rest of the implementation amounts to “book keeping” that needs to be done carefully, but is not too insightful. Hence we omit the details from this chapter. See the appendix for the full details of how to compute the universal NAND evaluator in NAND.

Since the loop over \(j\) in \(UPDATE\) is run \(2^\ell\) times, and computing \(EQUALS_j\) takes \(O(\ell)\) lines, the total number of lines to compute \(UPDATE\) is \(O(2^\ell \cdot \ell) = O(s \log s)\). Since we run this function \(s\) times, the total number of lines for computing \(EVAL_{s,n,m}\) is \(O(s^2 \log s)\). This completes (up to the omitted details) the proof of Theorem 5.2.

---

**Improving to quasilinear overhead (advanced optional note)** The NAND program above is less efficient than its Python counterpart, since NAND does not offer arrays with efficient random access. Hence for example the \(LOOKUP\) operation on an array of \(s\) bits takes \(\Omega(s)\) lines in NAND even though it takes \(O(1)\) steps (or maybe \(O(\log s)\) steps, depending how we count) in Python.

It turns out that it is possible to improve the bound of Theorem 5.2, and evaluate \(s\) line NAND programs using a NAND program of \(O(s \log s)\) lines. The key is to consider the description of NAND programs as...
5.2 A PYTHON INTERPRETER IN NAND (DISCUSSION)

To prove Theorem 5.2 we essentially translated every line of the Python program for \texttt{EVAL} into an equivalent NAND snippet. It turns out that none of our reasoning was specific to the particular function \texttt{EVAL}. It is possible to translate every Python program into an equivalent \texttt{NAND} program of comparable efficiency.\footnote{More concretely, if the Python program takes \(T(n)\) operations on inputs of length at most \(n\) then we can find a NAND program of \(O(T(n) \log T(n))\) lines that agrees with the Python program on inputs of length \(n\).} Actually doing so requires taking care of many details and is beyond the scope of this course, but let me convince you why you should believe it is possible in principle. We can use \texttt{CPython} (the reference implementation for Python), to evaluate every Python program using a C program. We can combine this with a C compiler to transform a Python program to various flavors of “machine language”.

So, to transform a Python program into an equivalent NAND program, it is enough to show how to transform a machine language program into an equivalent NAND program. One minimalistic (and hence convenient) family of machine languages is known as the \texttt{ARM} architecture which powers a great many mobile devices including essentially all Android devices.\footnote{ARM stands for “Advanced RISC Machine” where RISC in turn stands for “Reduced instruction set computer”.} There are even simpler machine languages, such as the \texttt{LEG} architecture for which a backend for the LLVM compiler was implemented (and hence can be the target of compiling any of large and growing list of languages that this compiler supports). Other examples include the \texttt{TinyRAM} architecture (motivated by interactive proof systems that we will discuss much later in this course) and the teaching-oriented \texttt{Ridiculously Simple Computer} architecture.\footnote{The reverse direction of compiling NAND to C code, is much easier. We show code for a \texttt{NAND2C} function in the appendix.}

Going one by one over the instruction sets of such computers and translating them to NAND snippets is no fun, but it is a feasible thing to do. In fact, ultimately this is very similar to the transformation that takes place in converting our high level code to actual silicon gates that are not so different from the operations of a NAND program. Indeed, tools such as \texttt{MyHDL} that transform “Python to Silicon” can
be used to convert a Python program to a NAND program.

The NAND programming language is just a teaching tool, and by no means do I suggest that writing NAND programs, or compilers to NAND, is a practical, useful, or even enjoyable activity. What I do want is to make sure you understand why it can be done, and to have the confidence that if your life (or at least your grade in this course) depended on it, then you would be able to do this. Understanding how programs in high level languages such as Python are eventually transformed into concrete low-level representation such as NAND is fundamental to computer science.

The astute reader might notice that the above paragraphs only outlined why it should be possible to find for every particular Python-computable function $F$, a particular comparably efficient NAND program $P$ that computes $F$. But this still seems to fall short of our goal of writing a “Python interpreter in NAND” which would mean that for every parameter $n$, we come up with a single NAND program $UNIV_n$ such that given a description of a Python program $P$, a particular input $x$, and a bound $T$ on the number of operations (where the length of $P$, $x$ and the magnitude of $T$ are all at most $n$) would return the result of executing $P$ on $x$ for at most $T$ steps. After all, the transformation above would transform every Python program into a different NAND program, but would not yield “one NAND program to rule them all” that can evaluate every Python program up to some given complexity. However, it turns out that it is enough to show such a transformation for a single Python program. The reason is that we can write a Python interpreter in Python: a Python program $U$ that takes a bit string, interprets it as Python code, and then runs that code. Hence, we only need to show a NAND program $U^*$ that computes the same function as the particular Python program $U$, and this will give us a way to evaluate all Python programs.

What we are seeing time and again is the notion of universality or self reference of computation, which is the sense that all reasonably rich models of computation are expressive enough that they can “simulate themselves”. The importance of this phenomena to both the theory and practice of computing, as well as far beyond it, including the foundations of mathematics and basic questions in science, cannot be overstated.

5.3 COUNTING PROGRAMS, AND LOWER BOUNDS ON THE SIZE OF NAND PROGRAMS

One of the consequences of our representation is the following:
Theorem 5.5 — Counting programs.

\[ |\text{Size}(s)| \leq 2^{O(s \log s)}, \]  

(5.3)

That is, there are at most \(2^{O(s \log s)}\) functions computed by NAND programs of at most \(s\) lines.

Moreover, the implicit constant in the \(O(\cdot)\) notation in Theorem 5.5 is at most 10.\(^9\)

**Proof Idea:** The idea behind the proof is that we can represent every \(s\) line program by a binary string of \(O(s \log s)\) bits. Therefore the number of functions computed by \(s\)-line programs cannot be larger than the number of such strings, which is \(2^{O(s \log s)}\). In the actual proof, given below, we count the number of representations a little more carefully, talking directly about triples rather than binary strings, although the idea remains the same.star

**Proof of Theorem 5.5.** Every NAND program \(P\) with \(s\) lines has at most \(3s\) variables. Hence, using our canonical representation, \(P\) can be represented by the numbers \(n, m\) of \(P\)’s inputs and outputs, as well as by the list \(L\) of \(s\) triples of natural numbers, each of which is smaller or equal to \(3s\).

If two programs compute distinct functions then they have distinct representations. So we will simply count the number of such representations: for every \(s' \leq s\), the number of \(s'\)-long lists of triples of numbers in \([3s]\) is \((3s)^{3s'}\), which in particular is smaller than \((3s)^{3s}\). So, for every \(s' \leq s\) and \(n, m\), the total number of representations of \(s'\)-line programs with \(n\) inputs and \(m\) outputs is smaller than \((3s)^{3s}\).

Since a program of at most \(s\) lines has at most \(s\) inputs and outputs, the total number of representations of all programs of at most \(s\) lines is smaller than

\[
s \times s \times s \times (3s)^{3s} = (3s)^{3s+3} \tag{5.4}
\]

(the factor \(s \times s\) times arises from taking all of the at most \(s\) options for the number of inputs \(n\), all of the at most \(s\) options for the number of outputs \(m\), and all of the at most \(s\) options for the number of lines \(s'\)). We claim that for \(s\) large enough, the righthand side of Eq. (5.4) (and hence the total number of representations of programs of at most \(s\) lines) is smaller than \(2^{4s \log s}\). Indeed, we can write \(3s = 2^\log(3s) = 2^\log 3^\log s \leq 2^{2^\log s}\), and hence the righthand side of Eq. (5.4) is at most \((2^2^\log s)^{3s+3} = 2^{2^\log s(3s+3)} \leq 2^{4s \log s}\) for \(s\) large enough.

For every function \(F \in \text{Size}(s)\) there is a program \(P\) of at most \(s\) lines that computes it, and we can map \(F\) to its representation as a tuple \((n, m, L)\). If \(F \neq F'\) then a program \(P\) that computes \(F\) must have

\* By this we mean that for all sufficiently large \(s\), \(|\text{Size}(s)| \leq 2^{10s \log s}\).
an input on which it disagrees with any program \( P' \) that computes \( F' \), and hence in particular \( P \) and \( P' \) have distinct representations. Thus we see that the map of \( \text{Size}(s) \) to its representation is one to one, and so in particular \( |\text{Size}(s)| \) is at most the number of distinct representations which is it at most \( 2^{4s \log s} \).

\[ \blacksquare \]

**Counting by ASCII representation** We can also establish Theorem 5.5 directly from the ASCII representation of the source code. Since an \( s \)-line NAND program has at most \( 3s \) distinct variables, we can change all the non input/output variables of such a program to have the form \( \text{Temp}[i] \) for \( i \) between 0 and \( 3s - 1 \) without changing the function that it computes. This means that after removing extra whitespaces, every line of such a program (which will be of the form \( \text{var} = \text{NAND}(\text{var'},\text{var''}) \)) for variable identifiers which will be either \( X[\#\#\#];X[\#\#\#] \) or \( \text{Temp}[\#\#\#] \) where \( \#\#\# \) is some number smaller than \( 3s \) will require at most, say, \( 20 + 3 \log_{10}(3s) \leq O(\log s) \) characters. Since each one of those characters can be encoded using seven bits in the ASCII representation, we see that the number of functions computed by \( s \)-line NAND programs is at most \( 2^{O(\log s)} \).

A function mapping \( \{0,1\}^2 \) to \( \{0,1\} \) can be identified with the table of its four values on the inputs \( 00, 01, 10, 11 \); a function mapping \( \{0,1\}^3 \) to \( \{0,1\} \) can be identified with the table of its eight values on the inputs \( 000, 001, 010, 011, 100, 101, 110, 111 \). More generally, every function \( F : \{0,1\}^n \rightarrow \{0,1\} \) can be identified with the table of its \( 2^n \) values on the inputs \( \{0,1\}^n \). Hence the number of functions mapping \( \{0,1\}^n \) to \( \{0,1\} \) is equal to the number of such tables which (since we can choose either 0 or 1 for every row) is exactly \( 2^{2^n} \). Note that this is double exponential in \( n \), and hence even for small values of \( n \) (e.g., \( n = 10 \)) the number of functions from \( \{0,1\}^n \) to \( \{0,1\} \) is truly astronomical.\(^{10}\) This has the following interesting corollary:

**Theorem 5.6 — Counting argument lower bound.** There is a function \( F : \{0,1\}^n \rightarrow \{0,1\} \) such that the shortest NAND program to compute \( F \) requires \( 2^n/(100n) \) lines.

**Proof.** Suppose, towards the sake of contradiction, that every function \( F : \{0,1\}^n \rightarrow \{0,1\} \) can be computed by a NAND program of at most \( s = 2^n/(100n) \) lines. Then the by Theorem 5.5 the total number of such functions would be at most \( 2^{10s \log s} \leq 2^{10 \log s \cdot 2^n/(100n)} \). Since \( \log s = n - \log(100n) \leq n \) this means that the total number of such

\(^{10}\) “Astronomical” here is an understatement: there are much fewer than \( 2^{2^{10}} \) stars, or even particles, in the observable universe.
functions would be at most $2^{2n/10}$, contradicting the fact that there are $2^n$ of them.

We have seen before that every function mapping $\{0, 1\}^n$ to $\{0, 1\}$ can be computed by an $O(2^n / n)$ line program. We now see that this is tight in the sense that some functions do require such an astronomical number of lines to compute. In fact, as we explore in the exercises below, this is the case for most functions. Hence functions that can be computed in a small number of lines (such as addition, multiplication, finding short paths in graphs, or even the $EVAL$ function) are the exception, rather than the rule.

![Diagram](image)

**Figure 5.3.** All functions mapping $n$ bits to $m$ bits can be computed by NAND programs of $O(n 2^n / n)$ lines, but most functions cannot be computed using much smaller programs. However there are many important exceptions which are functions such as addition, multiplication, program evaluation, and many others, that can be computed in polynomial time with a small exponent.

---

**Advanced note: more efficient representation** The list of triples is not the shortest representation for NAND programs. We have seen that every NAND program of $s$ lines and $n$ inputs can be represented by a directed graph of $s + n$ vertices, of which $n$ have in-degree zero, and the $s$ others have in-degree at most two. Using the adjacency list representation, such a graph can be represented using roughly $2s \log(s + n) \leq 2s(\log s + O(1))$ bits. Using this representation we can reduce the implicit constant in Theorem 5.5 arbitrarily close to 2.
5.4 THE PHYSICAL EXTENDED CHURCH-TURING THESIS

( DISCUSSION )

We’ve seen that NAND gates can be implemented using very different systems in the physical world. What about the reverse direction? Can NAND programs simulate any physical computer?

We can take a leap of faith and stipulate that NAND programs do actually encapsulate every computation that we can think of. Such a statement (in the realm of infinite functions, which we’ll encounter in Chapter 6) is typically attributed to Alonzo Church and Alan Turing, and in that context is known as the Church Turing Thesis. As we will discuss in future lectures, the Church-Turing Thesis is not a mathematical theorem or conjecture. Rather, like theories in physics, the Church-Turing Thesis is about mathematically modelling the real world. In the context of finite functions, we can make the following informal hypothesis or prediction:

**If a function** \( F : \{0,1\}^n \rightarrow \{0,1\}^m \) **can be computed in the physical world using** \( s \) **amount of “physical resources” then it can be computed by a NAND program of roughly** \( s \) **lines.**

We call this hypothesis the “Physical Extended Church-Turing Thesis” or PECTT for short. A priori it might seem rather extreme to hypothesize that our meager NAND model captures all possible physical computation. But yet, in more than a century of computing technologies, no one has yet built any scalable computing device that challenges this hypothesis.

We now discuss the “fine print” of the PECTT in more detail, as well as the (so far unsuccessful) challenges that have been raised against it. There is no single universally-agreed-upon formalization of “roughly \( s \) physical resources”, but we can approximate this notion by considering the size of any physical computing device and the time it takes to compute the output, and ask that any such device can be simulated by a NAND program with a number of lines that is a polynomial (with not too large exponent) in the size of the system and the time it takes it to operate.

In other words, we can phrase the PECTT as stipulating that any function that can be computed by a device of volume \( V \) and time \( t \), must be computable by a NAND program that has at most \( \alpha(Vt)^\beta \) lines for some constants \( \alpha, \beta \). The exact values for \( \alpha, \beta \) are not so clear, but it is generally accepted that if \( F : \{0,1\}^n \rightarrow \{0,1\} \) is an exponentially hard function, in the sense that it has no NAND program of fewer than, say, \( 2^{n/2} \) lines, then a demonstration of a physical device that can compute \( F \) for moderate input lengths (e.g., \( n = 500 \)) would be a
violation of the PECTT.

Advanced note: making PECTT concrete We can attempt at a more exact phrasing of the PECTT as follows. Suppose that $Z$ is a physical system that accepts $n$ binary stimuli and has a binary output, and can be enclosed in a sphere of volume $V$. We say that the system $Z$ computes a function $F : \{0, 1\}^n \rightarrow \{0, 1\}$ within $t$ seconds if whenever we set the stimuli to some value $x \in \{0, 1\}^n$, if we measure the output after $t$ seconds then we obtain $F(x)$.

We can phrase the PECTT as stipulating that if there exists such a system $Z$ that computes $F$ within $t$ seconds, then there exists a NAND program that computes $F$ and has at most $\alpha(Vt)^2$ lines, where $\alpha$ is some normalization constant. In particular, suppose that $F : \{0, 1\}^n \rightarrow \{0, 1\}$ is a function that requires $2^n / (100n) > 2^{0.8n}$ lines for any NAND program (such a function exists by Theorem 5.6). Then the PECTT would imply that either the volume or the time of a system that computes $F$ will have to be at least $2^{0.8n}/\sqrt{\alpha}$. Since this quantity grows exponentially in $n$, it is not hard to set parameters so that even for moderately large values of $n$, such a system could not fit in our universe.

To fully make the PECTT concrete, we need to decide on the units for measuring time and volume, and the normalization constant $\alpha$. One conservative choice is to assume that we could squeeze computation to the absolute physical limits (which are many orders of magnitude beyond current technology). This corresponds to setting $\alpha = 1$ and using the Planck units for volume and time. The Planck length $\ell_P$ (which is, roughly speaking, the shortest distance that can theoretically be measured) is roughly $2^{-120}$ meters. The Planck time $t_P$ (which is the time it takes for light to travel one Planck length) is about $2^{-150}$ seconds. In the above setting, if a function $F$ takes, say, 1KB of input (e.g., roughly $10^4$ bits, which can encode a 100 by 100 bitmap image), and requires at least $2^{0.8n} = 2^{0.8 \times 10^4}$ NAND lines to compute, then any physical system that computes it would require either volume of $2^{0.2 \times 10^4}$ Planck length cubed, which is more than $2^{1500}$ meters cubed or take at least $2^{0.2 \times 10^4}$ Planck Time units, which is larger than $2^{1500}$ seconds. To get a sense of how big that number is, note that the universe is only about $2^{50}$ seconds old, and its observable radius is only roughly $2^{30}$ meters. The above discussion suggests that it is possible to empirically falsify the PECTT by presenting a smaller-than-universe-size system that computes such a function.

12 We can also consider variants where we use surface area instead of volume, or take $(Vt)^2$ to a different power than 2. However, none of these choices makes a qualitative difference to the discussion below.

12 There are of course several hurdles to refuting the PECTT in this way, one of which is that we can’t actually test the system on all possible inputs. However, it turns out that we can get around this issue using notions such as interactive proofs and program checking that we might encounter later in this book. Another, perhaps more salient problem, is that while we know many hard functions exist, at the moment there is no single explicit function $F : \{0, 1\}^n \rightarrow \{0, 1\}$ for which we can prove an $\omega(n)$ (let alone $\Omega(2^n/n)$) lower bound on the number of lines that a NAND program needs to compute it.
5.4.1 Attempts at refuting the PECTT

One of the admirable traits of mankind is the refusal to accept limitations. In the best case this is manifested by people achieving long-standing “impossible” challenges such as heavier-than-air flight, putting a person on the moon, circumnavigating the globe, or even resolving Fermat’s Last Theorem. In the worst case it is manifested by people continually following the footsteps of previous failures to try to do proven-impossible tasks such as build a perpetual motion machine, trisect an angle with a compass and straightedge, or refute Bell’s inequality. The Physical Extended Church Turing thesis (in its various forms) has attracted both types of people. Here are some physical devices that have been speculated to achieve computational tasks that cannot be done by not-too-large NAND programs:

- **Spaghetti sort:** One of the first lower bounds that Computer Science students encounter is that sorting $n$ numbers requires making $\Omega(n \log n)$ comparisons. The “spaghetti sort” is a description of a proposed “mechanical computer” that would do this faster. The idea is that to sort $n$ numbers $x_1, \ldots, x_n$, we could cut $n$ spaghetti noodles into lengths $x_1, \ldots, x_n$, and then if we simply hold them together in our hand and bring them down to a flat surface, they will emerge in sorted order. There are a great many reasons why this is not truly a challenge to the PECTT hypothesis, and I will not ruin the reader’s fun in finding them out by her or himself.

- **Soap bubbles:** One function $F : \{0, 1\}^n \to \{0, 1\}$ that is conjectured to require a large number of NAND lines to solve is the Euclidean Steiner Tree problem. This is the problem where one is given $m$ points in the plane $(x_1, y_1), \ldots, (x_m, y_m)$ (say with integer coordinates ranging from 1 till $m$), and hence the list can be represented as a string of $n = O(m \log m)$ size and some number $K$. The goal is to figure out whether it is possible to connect all the points by line segments of total length at most $K$. This function is conjectured to be hard because it is NP complete - a concept that we’ll encounter later in this course - and it is in fact reasonable to conjecture that as $m$ grows, the number of NAND lines required to compute this function grows exponentially in $m$, meaning that the PECTT would predict that if $m$ is sufficiently large (such as few hundreds or so) then no physical device could compute $F$. Yet, some people claimed that there is in fact a very simple physical device that could solve this problem, that can be constructed using some wooden pegs and soap. The idea is that if we take two glass plates, and put $m$ wooden pegs between them in the locations $(x_1, y_1), \ldots, (x_m, y_m)$ then bubbles will form whose edges touch those pegs in the way that will minimize the total energy which turns out to be a func-
tion of the total length of the line segments. The problem with this device of course is that nature, just like people, often gets stuck in “local optima”. That is, the resulting configuration will not be one that achieves the absolute minimum of the total energy but rather one that can’t be improved with local changes. Aaronson has carried out actual experiments (see Fig. 5.4), and saw that while this device often is successful for three or four pegs, it starts yielding suboptimal results once the number of pegs grows beyond that.

![Figure 5.4](image)

Figure 5.4: Scott Aaronson tests a candidate device for computing Steiner trees using soap bubbles.

- **DNA computing.** People have suggested using the properties of DNA to do hard computational problems. The main advantage of DNA is the ability to potentially encode a lot of information in relatively small physical space, as well as compute on this information in a highly parallel manner. At the time of this writing, it was demonstrated that one can use DNA to store about $10^{16}$ bits of information in a region of radius about milimeter, as opposed to about $10^{10}$ bits with the best known hard disk technology. This does not posit a real challenge to the PECTT but does suggest that one should be conservative about the choice of constant and not assume that current hard disk + silicon technologies are the absolute best possible.\(^{13}\)

- **Continuous/real computers.** The physical world is often described using continuous quantities such as time and space, and people have suggested that analog devices might have direct access to computing with real-valued quantities and would be inherently

\(^{13}\) We were extremely conservative in the suggested parameters for the PECTT, having assumed that as many as $\ell_j^2 10^{-6} \sim 10^{01}$ bits could potentially be stored in a milimeter radius region.
more powerful than discrete models such as NAND machines. Whether the “true” physical world is continuous or discrete is an open question. In fact, we do not even know how to precisely phrase this question, let alone answer it. Yet, regardless of the answer, it seems clear that the effort to measure a continuous quantity grows with the level of accuracy desired, and so there is no “free lunch” or way to bypass the PECTT using such machines (see also this paper). Related to that are proposals known as “hypercomputing” or “Zeno’s computers” which attempt to use the continuity of time by doing the first operation in one second, the second one in half a second, the third operation in a quarter second and so on. These fail for a similar reason to the one guaranteeing that Achilles will eventually catch the tortoise despite the original Zeno’s paradox.

- **Relativity computer and time travel.** The formulation above assumed the notion of time, but under the theory of relativity time is in the eye of the observer. One approach to solve hard problems is to leave the computer to run for a lot of time from his perspective, but to ensure that this is actually a short while from our perspective. One approach to do so is for the user to start the computer and then go for a quick jog at close to the speed of light before checking on its status. Depending on how fast one goes, few seconds from the point of view of the user might correspond to centuries in computer time (it might even finish updating its Windows operating system!). Of course the catch here is that the energy required from the user is proportional to how close one needs to get to the speed of light. A more interesting proposal is to use time travel via *closed timelike curves (CTCs)*. In this case we could run an arbitrarily long computation by doing some calculations, remembering the current state, and the travelling back in time to continue where we left off. Indeed, if CTCs exist then we’d probably have to revise the PECTT (though in this case I will simply travel back in time and edit these notes, so I can claim I never conjectured it in the first place…)

- **Humans.** Another computing system that has been proposed as a counterexample to the PECTT is a 3 pound computer of about 0.1m radius, namely the human brain. Humans can walk around, talk, feel, and do others things that are not commonly done by NAND programs, but can they compute partial functions that NAND programs cannot? There are certainly computational tasks that at the moment humans do better than computers (e.g., play some video games, at the moment), but based on our current understanding of the brain, humans (or other animals) have no inherent computational advantage over computers. The brain has about $10^{11}$ neurons, each operating in a speed of about 1000 operations
per seconds. Hence a rough first approximation is that a NAND program of about $10^{14}$ lines could simulate one second of a brain’s activity.\footnote{14} Note that the fact that such a NAND program (likely) exists does not mean it is easy to find it. After all, constructing this program took evolution billions of years. Much of the recent efforts in artificial intelligence research is focused on finding programs that replicate some of the brain’s capabilities and they take massive computational effort to discover, these programs often turn out to be much smaller than the pessimistic estimates above. For example, at the time of this writing, Google’s neural network for machine translation has about $10^4$ nodes (and can be simulated by a NAND program of comparable size). Philosophers, priests and many others have since time immemorial argued that there is something about humans that cannot be captured by mechanical devices such as computers; whether or not that is the case, the evidence is thin that humans can perform computational tasks that are inherently impossible to achieve by computers of similar complexity.\footnote{15}

- Quantum computation. The most compelling attack on the Physical Extended Church Turing Thesis comes from the notion of quantum computing. The idea was initiated by the observation that systems with strong quantum effects are very hard to simulate on a computer. Turning this observation on its head, people have proposed using such systems to perform computations that we do not know how to do otherwise. At the time of this writing, Scalable quantum computers have not yet been built, but it is a fascinating possibility, and one that does not seem to contradict any known law of nature. We will discuss quantum computing in much more detail later in this course. Modeling it will essentially involve extending the NAND programming language to the “QNAND” programming language that has one more (very special) operation. However, the main take away is that while quantum computing does suggest we need to amend the PECTT, it does not require a complete revision of our worldview. Indeed, almost all of the content of this course remains the same whether the underlying computational model is the “classical” model of NAND programs or the quantum model of QNAND programs (also known as quantum circuits).

\footnote{14} This is a very rough approximation that could be wrong to a few orders of magnitude in either direction. For one, there are other structures in the brain apart from neurons that one might need to simulate, hence requiring higher overhead. On the other hand, it is by no mean clear that we need to fully clone the brain in order to achieve the same computational tasks that it does.

\footnote{15} There are some well known scientists that have advocated that humans have inherent computational advantages over computers. See also this.
We say “conjectured” and not “proved” because, while we can phrase such a statement as a precise mathematical conjecture, at the moment we are unable to prove such a statement for any cryptosystem. This is related to the \( P \) vs \( NP \) question we will discuss in future chapters.

“Security” really means that (a) it is conjectured that there is no Boolean circuit (or, equivalently, a NAND gate) of size much smaller than \( 2^{2^{128}} \) that can break the system, and (b) we assume that no other physical mechanism can do better, and hence it would take roughly a \( 2^{2^{128}} \) amount of “resources” to break the system.

**Lecture Recap**

- We can think of programs both as describing a process, as well as simply a list of symbols that can be considered as data that can be fed as input to other programs.
- We can write a NAND program that evaluates arbitrary NAND programs. Moreover, the efficiency loss in doing so is not too large.
- We can even write a NAND program that evaluates programs in other programming languages such as Python, C, Lisp, Java, Go, etc.
- By a leap of faith, we could hypothesize that the number of lines in the smallest NAND program for a function \( F \) captures roughly the amount of physical resources required to compute \( F \). This statement is known as the Physical Extended Church-Turing Thesis (PECTT).
- NAND programs capture a surprisingly wide array of computational models. The strongest currently known challenge to the PECTT comes from the potential for using quantum mechanical effects to speed-up computation, a model known as quantum computers.

**5.5 Exercises**

**Disclaimer** Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

**Exercise 5.1** Which one of the following statements is false:

a. There is an \( O(s^3) \) line NAND program that given as input program \( P \) of \( s \) lines in the list-of-tuples representation computes the
output of \( P \) when all its input are equal to 1.

b. There is an \( O(s^3) \) line NAND program that given as input program \( P \) of \( s \) characters encoded as a string of \( 7s \) bits using the ASCII encoding, computes the output of \( P \) when all its input are equal to 1.

c. There is an \( O(\sqrt{s}) \) line NAND program that given as input program \( P \) of \( s \) lines in the list-of-tuples representation computes the output of \( P \) when all its input are equal to 1.

Exercise 5.2 — Equals function. For every \( k \in \mathbb{N} \), show that there is an \( O(k) \) line NAND program that computes the function \( EQUALS_k : \{0,1\}^{2k} \rightarrow \{0,1\} \) where \( EQUALS(x, x') = 1 \) if and only if \( x = x' \).

Exercise 5.3 — Equal to constant function. For every \( k \in \mathbb{N} \) and \( x' \in \{0,1\}^k \), show that there is an \( O(k) \) line NAND program that computes the function \( EQUALS_{x'} : \{0,1\}^k \rightarrow \{0,1\} \) that on input \( x \in \{0,1\}^k \) outputs 1 if and only if \( x = x' \).

Exercise 5.4 — Random functions are hard (challenge). Suppose \( n > 1000 \) and that we choose a function \( U : \{0,1\}^n \rightarrow \{0,1\} \) at random, choosing for every \( x \in \{0,1\}^n \) the value \( F(x) \) to be the result of tossing an independent unbiased coin. Prove that the probability that there is a \( 2^n/(1000n) \) line program that computes \( F \) is at most \( 2^{-100} \).

Exercise 5.5 — Circuit hierarchy theorem (challenge). Prove that there is a constant \( c \) such that for every \( n \), there is some function \( F : \{0,1\}^n \rightarrow \{0,1\} \) s.t. (1) \( F \) can be computed by a NAND program of at most \( cn^5 \) lines, but (2) \( F \) cannot be computed by a NAND program of at most \( n^4/c \) lines.

5.6 BIBLIOGRAPHICAL NOTES

Scott Aaronson’s blog post on how information is physical is a good discussion on issues related to the physical extended Church-Turing Physics. Aaronson’s survey on NP complete problems and physical reality is also a great source for some of these issues, though might be easier to read after we reach Chapter 14 on NP and NP-completeness.

5.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

- Lower bounds. While we’ve seen the “most” functions mapping \( n \) bits to one bit require NAND programs of exponential size \( \Omega(2^n / n) \), we actually do not know of any explicit function for which we can prove that it requires, say, at least \( n^{100} \) or even \( 100n \) size. At
the moment, strongest such lower bound we know is that there are quite simple and explicit $n$-variable functions that require at least $(5 - o(1))n$ lines to compute, see this paper of Iwama et al as well as this more recent work of Kulikov et al. Proving lower bounds for restricted models of straightline programs (more often described as circuits) is an extremely interesting research area, for which Jukna’s book provides very good introduction and overview.
II
UNIFORM COMPUTATION
6
Loops and infinity

“We thus see that when $n = 1$, nine operation-cards are used; that when $n = 2$, fourteen Operation-cards are used; and that when $n > 2$, twenty-five operation-cards are used; but that no more are needed, however great $n$ may be; and not only this, but that these same twenty-five cards suffice for the successive computation of all the numbers”, Ada Augusta, countess of Lovelace, 1843

“It is found in practice that (Turing machines) can do anything that could be described as ‘rule of thumb’ or ‘purely mechanical’… (Indeed,) it is now agreed amongst logicians that ‘calculable by means of (a Turing Machine)’ is the correct accurate rendering of such phrases.”, Alan Turing, 1948

The NAND programming language (or equivalently, the Boolean circuits model) has one very significant drawback: a finite NAND program $P$ can only compute a finite function $F$, and in particular the number of inputs of $F$ is always smaller than (twice) the number of lines of $P$.\(^2\)

This does not capture our intuitive notion of an algorithm as a single recipe to compute a potentially infinite function. For example, the standard elementary school multiplication algorithm is a single algorithm that multiplies numbers of all lengths, but yet we cannot express this algorithm as a single NAND program, but rather need a different NAND program for every input length (see Fig. 6.1).

Let us consider the case of the simple parity or XOR function $XOR : \{0, 1\}^* \rightarrow \{0, 1\}$, where $XOR(x)$ equals 1 iff the number of 1’s in $x$ is odd. As simple as it is, the $XOR$ function cannot be computed by a NAND program. Rather, for every $n$, we can compute $XOR_n$ (the

---

\(^1\) Translation of “Sketch of the Analytical Engine” by L. F. Menabrea, Note G.

\(^2\) This conceptual point holds for any straightline programming language, and is independent of the particular syntactical choices we made for NAND. The particular ratio of “twice” is true for NAND because input variables cannot be written to, and hence a NAND program of $s$ lines includes at most $2s$ input variables. Coupled with the fact that a NAND program can’t include $X[i]$ if it doesn’t include $X[j]$ for $j < i$, this implies that the length of the input is at most $2s$. Similarly, a Boolean circuit whose gates correspond to two-input functions cannot have more inputs than twice the number of gates.
Once you know how to multiply multi-digit numbers, you can do so for every number \( n \) of digits, but if you had to describe multiplication using NAND programs or Boolean circuits, you would need a different program/circuit for every length \( n \) of the input.

The restriction of XOR to \( \{0,1\}^n \) using a different NAND program. For example, here is the NAND program to compute \( \text{XOR}_5 \): (see also Fig. 6.2)

\[
\begin{align*}
\text{Temp}[0] & = \text{NAND}(X[0], X[1]) \\
\text{Temp}[1] & = \text{NAND}(X[0], \text{Temp}[0]) \\
\text{Temp}[2] & = \text{NAND}(X[1], \text{Temp}[0]) \\
\text{Temp}[3] & = \text{NAND}(\text{Temp}[1], \text{Temp}[2]) \\
\text{Temp}[4] & = \text{NAND}(X[2], \text{Temp}[3]) \\
\text{Temp}[5] & = \text{NAND}(X[2], \text{Temp}[4]) \\
\text{Temp}[6] & = \text{NAND}(\text{Temp}[3], \text{Temp}[4]) \\
\text{Temp}[7] & = \text{NAND}(\text{Temp}[5], \text{Temp}[6]) \\
\text{Temp}[8] & = \text{NAND}(\text{Temp}[7], X[3]) \\
\text{Temp}[9] & = \text{NAND}(\text{Temp}[7], \text{Temp}[8]) \\
\text{Temp}[10] & = \text{NAND}(X[3], \text{Temp}[8]) \\
\text{Temp}[11] & = \text{NAND}(\text{Temp}[9], \text{Temp}[10]) \\
\text{Temp}[12] & = \text{NAND}(\text{Temp}[11], X[4]) \\
\text{Temp}[13] & = \text{NAND}(\text{Temp}[11], \text{Temp}[12]) \\
\text{Temp}[14] & = \text{NAND}(X[4], \text{Temp}[12]) \\
Y[0] & = \text{NAND}(\text{Temp}[13], \text{Temp}[14])
\end{align*}
\]

This is rather repetitive, and more importantly, does not capture the fact that there is a single algorithm to compute the parity on all inputs. Typical programming language use the notion of loops to express such an algorithm, and so we might have wanted to use code such as:

```python
# s is the "running parity", initialized to 0
while i < len(X):
    u = NAND(s, X[i])
    v = NAND(s, u)
```

Figure 6.1: Once you know how to multiply multi-digit numbers, you can do so for every number \( n \) of digits, but if you had to describe multiplication using NAND programs or Boolean circuits, you would need a different program/circuit for every length \( n \) of the input.


Figure 6.2: The circuit for computing the XOR of 5 bits. Note how it merely repeats four times the circuit to compute the XOR of 2 bits.

\[
\begin{align*}
w &= \text{NAND}(X[i], u) \\
s &= \text{NAND}(v, w) \\
i &+= 1 \\
Y[0] &= s
\end{align*}
\]

In this chapter we will show how we can extend the NAND programming language so that it can capture these kinds of constructs. We will see two ways to do so:

- The **NAND++ Programming language** extends NAND with the notion of loops and arrays to allow a finite program that can compute a function with arbitrarily long inputs.

- **Turing machines** are the classical way to give a finite description of an algorithm for arbitrarily long inputs.

It turns out that these two models are *equivalent*, and in fact they are equivalent to a great many other computational models including programming languages you may be familiar with such as C, Java, Python, Javascript, OCaml, and so on and so forth. This notion, known as *Turing equivalence* or *Turing completeness*, will be discussed in Chapter 7. We start off by presenting NAND++ and then show Turing machines, though it is also possible to present them in the opposite orders.
6.1 THE NAND++ PROGRAMMING LANGUAGE

The NAND++ programming language aims to capture the notion of a single uniform algorithm that can compute a function that takes inputs of arbitrary lengths. To do so, we need to extend the NAND programming language with two constructs:

- **Loops**: NAND is a straightline programming language—a NAND program of $s$ lines takes exactly $s$ steps of computation and hence in particular cannot even touch more than $3s$ variables. Loops allow us to capture in a short program the instructions for a computation that can take an arbitrary amount of time.

- **Arrays**: A NAND program of $s$ lines touches at most $3s$ variables. While we allow in NAND variables such as $\text{Foo}[17]$ or $\text{Bar}[22]$, they are not true arrays, since the number inside the brackets is a constant that is “hardwired” into the program. In particular a NAND program of $s$ lines cannot read an input $X[i]$ for $i > 2s$.

Thus a good way to remember NAND++ is using the following informal equation:

$$\text{NAND++} = \text{NAND} + \text{loops} + \text{arrays}$$  \hspace{1cm} (6.1)

It turns out that adding loops and arrays is enough to not only enable computing XOR, but in fact capture the full power of all programming languages! Hence we could replace “NAND++” with any of Python, C, Javascript, OCaml, etc… in the lefthand side of Eq. (6.1). But we’re getting ahead of ourselves: this issue will be discussed in Chapter 7.

6.1.1 Enhanced NAND++ programs

We now turn to describing the syntax of NAND++ programs. We’ll start by describing what we call the “enhanced NAND++ programming language”. Enhanced NAND++ has some extra features on top of NAND++ that make it easier to describe. However, we will see in Theorem 6.7 that these extra features can be implemented as “syntactic sugar” on top of standard or “vanilla” NAND++, and hence these two programming languages are equivalent in power.

Enhanced NAND++ programs add the following features on top of NAND:

- We add a special Boolean variable $\text{loop}$. If $\text{loop}$ is equal to 1 at the end of the execution then execution loops back to the first line of the program.
• We add a special integer valued variable \( i \). We add the commands \( i += \) \( foo \) and \( i -= \) \( bar \) that can add or subtract to \( i \) either zero or one, where \( foo \) and \( bar \) are standard (Boolean valued) variables.\(^3\)

• We add arrays to the language by allowing variable identifiers to have the form \( Foo[i] \). \( Foo \) is an array of Boolean values, and \( Foo[i] \) refers to the value of this array at location equal to the current value of the variable \( i \).

• The input and output \( X \) and \( Y \) are now considered arrays with values of zeroes and ones. Since both input and output could have arbitrary length, we also add two new arrays \( Xvalid \) and \( Yvalid \) to mark their length. We define \( Xvalid[i] = 1 \) if and only if \( i \) is smaller than the length of the input, and similarly we will set \( Yvalid[j] \) to equal 1 if and only if \( j \) is smaller than the length of the output.\(^4\)

\[ \text{Example 6.1 — XOR in Enhanced NAND++.} \]

The following is an enhanced NAND++ program to compute the XOR function on inputs of arbitrary length. That is \( \text{XOR} : \{0,1\}^* \to \{0,1\} \) such that \( \text{XOR}(x) = \sum_{i=0}^{\mid x \mid - 1} x_i \mod 2 \) for every \( x \in \{0,1\}^* \).

\[
\begin{align*}
temp_0 &= \text{NAND}(X[0], X[0]) \\
Yvalid[0] &= \text{NAND}(X[0], temp_0) \\
temp_2 &= \text{NAND}(X[i], Y[0]) \\
temp_3 &= \text{NAND}(X[i], temp_2) \\
temp_4 &= \text{NAND}(Y[0], temp_2) \\
Y[0] &= \text{NAND}(temp_3, temp_4) \\
loop &= \text{NAND}(temp_3, temp_4) \\
i &= Xvalid[i] \\
i &= Xvalid[i]
\end{align*}
\]

\[ \text{Example 6.2 — Increment in Enhanced NAND++.} \]

We now present enhanced NAND++ program to compute the increment function. That is, \( \text{INC} : \{0,1\}^* \to \{0,1\}^* \) such that for every \( x \in \{0,1\}^n \), \( \text{INC}(x) \) is the \( n+1 \) bit long string \( y \) such that if \( X = \sum_{i=0}^{n-1} x_i \cdot 2^i \) is the number represented by \( x \), then \( y \) is the binary representation of the number \( X + 1 \).

We start by showing the program using the “syntactic sugar” we’ve seen before of using shorthand for some NAND programs we have seen before to compute simple functions such as IF, XOR and AND (as well as the constant one function as well as the function COPY that just maps a bit to itself).
The above is not, strictly speaking, a valid enhanced NAND++ program. If we “open up” all of the syntactic sugar, we get the following valid program to compute this syntactic sugar.

\[
\begin{align*}
\text{temp}_0 &= \text{NAND}(\text{started}, \text{started}) \\
\text{temp}_1 &= \text{NAND}(\text{started}, \text{temp}_0) \\
\text{temp}_2 &= \text{NAND}(\text{started}, \text{started}) \\
\text{temp}_3 &= \text{NAND}(\text{temp}_1, \text{temp}_2) \\
\text{temp}_4 &= \text{NAND}(\text{carry}, \text{started}) \\
\text{carry} &= \text{NAND}(\text{temp}_3, \text{temp}_4) \\
\text{temp}_6 &= \text{NAND}(\text{started}, \text{started}) \\
\text{started} &= \text{NAND}(\text{started}, \text{temp}_6) \\
\text{temp}_8 &= \text{NAND}(\text{X}[i], \text{carry}) \\
\text{temp}_9 &= \text{NAND}(\text{X}[i], \text{temp}_8) \\
\text{temp}_{10} &= \text{NAND}(\text{carry}, \text{temp}_8) \\
\text{Y}[i] &= \text{NAND}(\text{temp}_9, \text{temp}_{10}) \\
\text{temp}_{12} &= \text{NAND}(\text{X}[i], \text{carry}) \\
\text{carry} &= \text{NAND}(\text{temp}_{12}, \text{temp}_{12}) \\
\text{temp}_{14} &= \text{NAND}(\text{started}, \text{started}) \\
\text{Yvalid}[i] &= \text{NAND}(\text{started}, \text{temp}_{14}) \\
\text{temp}_{16} &= \text{NAND}(\text{Xvalid}[i], \text{Xvalid}[i]) \\
\text{loop} &= \text{NAND}(\text{temp}_{16}, \text{temp}_{16}) \\
i &= \text{NAND}(\text{temp}_{16}, \text{temp}_{16})
\end{align*}
\]

Working out the above two example can go a long way towards understanding NAND++. See the appendix for a full specification of the language.

6.1.2 Variables as arrays and well-formed programs

In NAND we allowed variables to have names such as \texttt{foo}_{17} or even \texttt{Bar[23]} but the numerical part of the identifier played essentially the same role as alphabetical part. In particular, NAND would be just as powerful if we didn’t allow any numbers in the variable identifiers. With the introduction of the special index variable \(i\), in NAND++ things are different, and we do have actual arrays.
To make sure there is no confusion, we will use the convention that plain variables (which we will also refer to as scalar variables) are written with all lower case, and array variables begin with an upper case letter. Moreover, it turns out that we can ensure without loss of generality that arrays are always indexed by the variable i. (That is, if Foo is an array, then whenever Foo is referred to in the program, it is always in the form Foo[i] and never as Foo[17], Foo[159] or any other constant numerical index.) Hence all the variable identifiers in “well formed” NAND++ programs will either have the form foo_123 (a sequence of lower case letters, underscores, and numbers, with no brackets or upper case letters) or the form Bar[i] (an identifier starting with an upper case letter, and ending with [i]). See Lemma 6.9 for a more formal treatment of the notion of “well formed programs”.

6.1.3 “Oblivious” / “Vanilla” NAND++
Since our goal in theoretical computer science is not as much to construct programs as to analyze them, we want to use as simple as possible computational models. Hence our actual “plain vanilla” NAND++ programming language will be even more “bare bones” than enhanced NAND++. In particular, standard NAND++ does not contain the commands i += foo and i -= bar to control the integer-valued variable i. If we don’t have these commands, how would we ever be able to access arbitrary elements of our arrays? The idea is that standard NAND++ prescribes a pre-fixed schedule that i progresses in, regardless of the code of the program or the particular input. Just like a bus takes always the same route, and you need to wait until it reaches your station, if you want to access, for example, location 132 in the array Foo, you can wait until the iteration in which i will equal 132, at which point Foo[i] will refer to the 132-th bit of the array Foo.

So what is this schedule that i progresses in? There are many choices for such a schedule that would have worked, but we fix a particular choice for simplicity. Initially when we run a NAND++ program, the variable i equals 0. When we finish executing all the lines of code for the first time, if loop equals 0 we halt. Otherwise we continue to the second iteration, but this time the variable i will equal 1. At the end of this iteration once again we halt if loop equals 0, and otherwise we proceed to the third iteration where i gets the value of 0 again. We continue in this way with the fourth iteration having i = 1 and in the fifth iteration i is equal to 2, after which it decreases step by step to 0 again and so on and so forth. Generally, in the k-th iteration the value of i equals I(k) where I = (I(0), I(1), I(2), …) is the...
following sequence (see Fig. 6.3):

\[ 0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, ... \]  

(6.2)

The above is a perfectly fine description of the sequence \( I(0), I(1), I(2), ... \) but it is also possible to find an explicit mathematical formula for \( I(k) \). Specifically, it is an annoying but not hard exercise to show that \( I(k) \) is equal to the minimum of \( |k - r(r + 1)| \) where this minimum is taken over all integers \( r \) in \( \{0, ..., k\} \). It can also be shown that the value of \( r \) that achieves this minimum is between \( \sqrt{k} - 1 \) and \( \sqrt{k} \).

![Figure 6.3](image)

Figure 6.3: The value of \( i \) is a function of the current iteration. The variable \( i \) progresses according to the sequence 0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, .... Via some cumbersome but routine calculation, it can be shown that at the \( f \)-th iteration the value of \( i \) equals \( f - r(r + 1) \) if \( f \leq (r + 1)^2 \) and \( (r + 1)(r + 2) - f \) if \( f < (r + 1)^2 \) where \( r = \lfloor \sqrt{k + 1/4} - 1/2 \rfloor \).

**Example 6.3 — XOR in vanilla NAND++.** Here is the XOR function in NAND++ (using our standard syntactic sugar to make it more readable):

```plaintext
Yvalid[0] = one(X[0])
Y[0] = IF(Visited[i], Y[0], XOR(X[i], Y[0]))
Visited[i] = one(X[0])
loop = Xvalid[i]
```

Note that we use the array \( \text{Visited} \) to “mark” the positions of the input that we have already visited. The line \( \text{IF(Visited[i], Y[0], XOR(X[i], Y[0]))} \) ensures that the output value \( Y[0] \) is XOR’ed with the \( i \)-th bit of the input only at the first time we see it.

It would be very instructive for you to compare the enhanced NAND++ program for XOR of Example 6.1 with the standard NAND++ program of Example 6.3.
Solved Exercise 6.1 — Computing index location. Prove that at the $k$-th iteration of the loop, the value of the variable $i$ is equal to $\text{index}(k)$ where $\text{index} : \mathbb{N} \to \mathbb{N}$ is defined as follows:

$$
\text{index}(k) = \begin{cases} 
    k - r(r + 1) & k \leq (r + 1)^2 \\
    (r + 1)(r + 2) - k & \text{otherwise}
  \end{cases}
$$

where $r = \lfloor \sqrt{k + 1/4} - 1/2 \rfloor$.

Solution: We say that a NAND program completed its $r$-th round when the index variable $i$ reaches the 0 point for $r + 1$ times and hence completes the sequence:

$$
0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, \ldots, 0, 1, \ldots, r, r - 1, \ldots, 0
$$

This happens when the program completed

$$
1 + 2 + 4 + 6 + \cdots + 2r = r^2 + r + 1
$$

iterations of its main loop. (The last equality is obtained by applying the formula for the sum of an arithmetic progression.) This means that if we keep a “loop counter” $k$ that is initially set to 0 and increases by one at the end of any iteration, then the “round” $r$ is the largest integer such that $r(r + 1) \leq k$. One can verify that this means that $r = \lfloor \sqrt{k + 1/4} - 1/2 \rfloor$. When $k$ is between $r(r + 1)$ and $(r + 1)^2$ then the index $i$ is ascending, and hence the value of $\text{index}(k)$ will be $k - r(r + 1)$. When $k$ is between $(r + 1)^2$ and $(r + 1)(r + 2)$ then the index $i$ is descending, and hence the value of $\text{index}(k)$ will be $r - (k - (r + 1)^2) = (r + 1)(r + 2) - k$.

6.2 COMPUTABLE FUNCTIONS

We now turn to making one of the most important definitions in this book, that of computable functions. This definition is deceptively simple, but will be the starting point of many deep results and questions. We start by formalizing the notion of a NAND++ computation:

Definition 6.4 — NAND++ computation. Let $P$ be a NAND++ program. For every input $x \in \{0, 1\}^*$, we define the output of $P$ on input $x$ (denotes as $P(x)$) to be the result of the following process:

- Initialize the variables $X[i] = x_i$ and $X\text{valid}[i] = 1$ for all $i \in [n]$ (where $n = |x|$). All other variables (including $i$ and $\text{loop}$) default to 0.
• Run the program line by line. At the end of the program, if \( \text{loop}=1 \) then increment/decrement \( i \) according to the schedule 0, 1, 0, 1, 2, 1, 0, 1, ... and go back to the first line.

• If \( \text{loop}=0 \) at the end of the program, then we halt and output \( Y[0], ..., Y[m-1] \) where \( m \) is the smallest integer such that \( \text{Yvalid}(m) = 0 \).

If the program does not halt on input \( x \), then we say it has no output, and we denote this as \( P(x) = \bot \).

Enhanced NAND++ computation

Definition 6.4 can be easily adapted for enhanced NAND++ programs. The only modification is the natural one: instead of \( i \) travelling according to the sequence 0, 1, 0, 1, 2, 1, 0, 1, ..., \( i \) is increased/decreased based on the \( i += \text{foo} \) and \( i -= \text{bar} \) operations.

We can now define what it means for a function to be computable:

**Definition 6.5 — Computable functions.** Let \( F : \{0,1\}^* \to \{0,1\}^* \) be a (total) function and let \( P \) be a NAND++ program. We say that \( P \) computes \( F \) if for every \( x \in \{0,1\}^* \), \( P(x) = F(x) \).

We say that a function \( F \) is NAND++ computable if there is a NAND++ program that computes it.

We will often drop the “NAND++” qualifier and simply call a function computable if it is NAND++ computable. This may seem “reckless” but, as we’ll see in Chapter 7, it turns out that being NAND++-computable is equivalent to being computable in essentially any reasonable model of computation.

Definition 6.5 is, as we mentioned above, one of the most important definitions in this book. Please re-read it (and Definition 6.4) and make sure you understand it. Try to think how you would define the notion of a NAND++ program \( P \) computing a function, and make sure that you arrive at the same definition.

This is a good point to remind the reader of the distinction between functions and programs:

\[
\text{Functions} \neq \text{Programs} \quad (6.6)
\]

A program \( P \) can compute some function \( F \), but it is not the same
as $F$. In particular there can be more than one program to compute the same function. Being “NAND++ computable” is a property of functions, not of programs.

### Decidable languages

Many other texts use the term \textit{decidable languages} (also known as \textit{recursive languages}) instead of \textit{computable functions}. This terminology has its roots in formal language theory as was pursued by linguists such as Noam Chomsky. A \textit{formal language} is simply a subset $L \subseteq \{0,1\}^*$ (or more generally $L \subseteq \Sigma^*$ for some finite alphabet $\Sigma$). The \textit{membership} or \textit{decision} problem for a language $L$, is the task of determining, given $x \in \{0,1\}^*$, whether or not $x \in L$. One can see that this task is equivalent to computing the Boolean function $F : \{0,1\}^* \rightarrow \{0,1\}$ which is defined as $F(x) = 1$ iff $x \in L$. Thus saying that the function $F$ is computable is equivalent to saying that the corresponding language $L$ is decidable. The corresponding concept to a \textit{partial function} is known as a \textit{promise problem}.

### 6.2.1 Infinite loops and partial functions

One crucial difference between NAND and NAND++ programs is the following. Looking at a NAND program $P$, we can always tell how many inputs and how many outputs it has (by simply looking at the $X$ and $Y$ variables). Furthermore, we are guaranteed that if we invoke $P$ on any input then some output will be produced.

In contrast, given any particular NAND++ program $P'$, we cannot determine a priori the length of the output. In fact, we don’t even know if an output would be produced at all! For example, the following NAND++ program would go into an infinite loop if the first bit of the input is zero:

\[
\text{loop} = \text{NAND}(X[0], X[0])
\]

If a program $P$ fails to stop and produce an output on some an input $x$, then it cannot compute any total function $F$, since clearly on input $x$, $P$ will fail to output $F(x)$. However, $P$ can still compute a \textit{partial} function.\footnote{A \textit{partial function} $F$ from a set $A$ to a set $B$ is a function that is only defined on a \textit{subset} of $A$, (see Section 1.4.4). We can also think of such a function as mapping $A$ to $B \cup \{\bot\}$ where $\bot$ is a special “failure” symbol such that $F(a) = \bot$ indicates the function $F$ is not defined on $a$.}

For example, consider the partial function $DIV$ that on input a pair $(a, b)$ of natural numbers, outputs $\lfloor a/b \rfloor$ if $b > 0$, and is undefined otherwise. We can define a program $P$ that computes $DIV$ on input $a, b$ by outputting the first $c = 0, 1, 2, ...$ such that $cb \geq a$. If $a > 0$ and $b = 0$ then the program $P$ will never halt, but this is OK, since $DIV$ is undefined on such inputs. If $a = 0$ and $b = 0$, the program $P$ will output 0, which is also OK, since we don’t care about what the
program outputs on inputs on which \(\text{DIV}\) is undefined. Formally, we define computability of partial functions as follows:

**Definition 6.6 — Computable (partial or total) functions.** Let \(F\) be either a total or partial function mapping \(\{0,1\}^*\) to \(\{0,1\}^*\) and let \(P\) be a NAND++ program. We say that \(P\) computes \(F\) if for every \(x \in \{0,1\}^*\) on which \(F\) is defined, \(P(x) = F(x)\).\(^7\)

We say that a (partial or total) function \(F\) is NAND++ computable if there is a NAND++ program that computes it.

### 6.3 EQUIVALENCE OF “VANILLA” AND “ENHANCED” NAND++

We have defined so far not one but two programming languages to handle functions with unbounded input lengths: “enhanced” NAND++ which contains the \(i \leftarrow\) bar and \(i \rightarrow\) foo operations, and the standard or “vanilla” NAND++, which does not contain these operations, but rather where the index \(i\) travels obliviously according to the schedule 0, 1, 0, 1, 2, 1, 0, 1, ...

We now show these two versions are equivalent in power:

**Theorem 6.7 — Equivalence of enhanced and standard NAND++.** Let \(F : \{0,1\}^* \rightarrow \{0,1\}^*\). Then \(F\) is computable by a NAND++ program if and only if \(F\) is computable by an enhanced NAND++ program.

**Proof Idea:** To prove the theorem we need to show (1) that for every NAND++ program \(P\) there is an enhanced NAND++ program \(Q\) that computes the same function as \(P\), and (2) that for every enhanced NAND++ program \(Q\), there is a NAND++ program \(P\) that computes the same function as \(Q\).

Showing (1) is quite straightforward: all we need to do is to show that we can ensure that \(i\) follows the sequence 0, 1, 0, 1, 2, 1, 0, 1, ... using the \(i \leftarrow\) foo and \(i \rightarrow\) foo operations. The idea is that we use a Visited array to keep track at which places we visited, as well as a special Atstart array for which we ensure that \(\text{Atstart}[0] = 1\) but \(\text{Atstart}[i] = 0\) for every \(i > 0\). We can use these arrays to check in each iteration whether \(i\) is equal to 0 (in which case we want to execute \(i \leftarrow 1\) at the end of the iteration), whether \(i\) is at a point which we haven’t seen before (in which case we want to execute \(i \rightarrow 1\) at the end of the iteration), or whether it’s at neither of those extremes (in which case we should add or subtract to \(i\) the same value as the last iteration).

Showing (2) is a little more involved. Our main observation is that we can simulate a conditional GOTO command in NAND++. That is,
we can come up with some “syntactic sugar” that will have the effect of jumping to a different line in the program if a certain variable is equal to 1. Once we have this, we can implement looping commands such as \texttt{while}. This allows us to simulate a command such as \texttt{i += foo} when \texttt{i} is currently in the “decreasing phase” of its cycle by simply waiting until \texttt{i} reaches the same point in the “increasing phase”. The intuition is that the difference between standard and enhanced NAND++ is like the difference between a bus and a taxi. Enhanced NAND++ is like a taxi - you tell \texttt{i} where to do. Standard NAND++ is like a bus - you wait until \texttt{i} arrives at the point you want it to be in. A bus might be a little slower, but will eventually get you to the same place. ∗

We split the full proof of \textbf{Theorem 6.7} into two parts. In \textbf{Section 6.3.1} we show the easier direction of simulating standard NAND++ programs by enhanced ones. In \textbf{Section 6.3.2} we show the harder direction of simulating enhanced NAND++ programs by standard ones. Along the way we will show how we can simulate the \texttt{GOTO} operation in NAND++ programs.

\textbf{6.3.1 Simulating NAND++ programs by enhanced NAND++ programs.}

Let \texttt{P} be a standard NAND++ program. To create an enhanced NAND++ program that computes the same function, we will add a variable \texttt{indexincreasing} and code to ensure that at the end of the iteration, if \texttt{indexincreasing} equals 1 then \texttt{i} needs to increase by 1 and otherwise \texttt{i} needs to decrease by 1. Once we ensure that, we can emulate \texttt{P} by simply adding the following lines to the end of the program

\texttt{i += indexincreasing}
\texttt{i -= NOT(indexincreasing)}

where \texttt{one} and \texttt{zero} are variables which are always set to be zero or one, and \texttt{IF} is shorthand for NAND implementation of our usual IF function (i.e., \texttt{IF(a,b,c)} equals \texttt{b} if \texttt{a} = 1 and \texttt{c} otherwise).

To compute \texttt{indexincreasing} we use the fact that the sequence 0, 1, 0, 1, 2, 1, 0, 1, ... of \texttt{i}’s travels in a standard NAND++ program is obtained from the following rules:

1. At the beginning \texttt{i} is increasing.
2. If \texttt{i} reaches a point which it hasn’t seen before, then it starts decreasing.
3. If \texttt{i} reaches the initial point 0, then it starts increasing.
To know which points we have seen before, we can borrow Hansel and Gretel’s technique of leaving “breadcrumbs”. That is, we will create an array Visited and add code Visited[i] = one at the end of every iteration. This means that if Visited[i] = 0 then we know we have not visited this point before. Similarly, we create an array Atstart array and add code Atstart[0] = one (while all other location remain at the default value of zero). Now we can use Visited and Atstart to compute the value of indexincreasing. Specifically, we will add the following pieces of code:

\begin{verbatim}
Atstart[0] = COPY(one)
indexincreasing =
  IF(Visited[i],indexincreasing,zero)
indexincreasing = IF(Atstart[i],one,indexincreasing)
Visited[i] = COPY(one)
\end{verbatim}

at the very end of the program.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure}
\caption{We can know if the index variable i should increase or decrease by keeping an array atstart letting us know when i reaches 0, and hence i starts increasing, and breadcrumb letting us know when we reach a point we haven’t seen before, and hence i starts decreasing. TODO: update figure to Atstart and Visited notation.}
\end{figure}

Given any standard NAND++ program P, we can add the above lines of code to it to obtain an enhanced NAND++ program Q that will behave in exactly the same way as P and hence will compute the same function. This completes the proof of the first part of Theorem 6.7.

### 6.3.2 Simulating enhanced NAND++ programs by NAND++ programs.

To simulate enhanced NAND++ programs by vanilla ones, we will do as follows. We introduce an array Markposition which normally would be all zeroes. We then replace the line i += foo with code that achieves the following:

1. We first check if foo=0. If so, then we do nothing.
2. Otherwise we set \( \text{Markposition}[i] = \text{one} \).

3. We then want to add code that will do nothing until we get to the position \( i + 1 \). We can check this condition by verifying that both \( \text{Markposition}[i] = 1 \) and \( \text{indexincreasing} = 1 \) at the end of the iteration.

We will start by describing how we can achieve this under the assumption that we have access to \texttt{GOTO} and \texttt{LABEL} operations. \texttt{LABEL} \((l)\) simply marks a line of code with the string \( l \). \texttt{GOTO} \((l, \text{cond})\) jumps in execution to the position labeled \( l \) if \( \text{cond} \) is equal to \( 1 \).\(^8\)

If the original program had the form:

\[
\text{pre-code... } \# \text{pre-increment-code} \\
i += \text{foo} \\
\text{post-code... } \# \text{post-increment-code}
\]

Then the new program will have the following form:

\[
\text{pre-code... } \# \text{pre-increment code} \\
\# \text{replacement for } i += \text{foo} \\
w\text{aiting} = \text{foo} \ # \text{if } \text{foo}=1 \text{ then we need to wait} \\
\text{Markposition}[i] = \text{foo} \ # \text{we mark the position we were at} \\
\text{GO}\text{T}(\"end\", w\text{aiting}) \ # \text{If waiting then jump till end.} \\
\text{LABEL}(\"postcode\") \\
w\text{aiting} = \text{zero} \\
t\text{imeforpostcode} = \text{zero} \\
\text{post-code...} \\
\text{LABEL}(\"end\") \\
\text{maintainance-code... } \# \text{maintain value of} \\
\# \text{indexincreasing variable as before} \\
c\text{ondition} \quad \text{AND} \quad (\text{Markposition}[i], \text{indexincreasing}) \ # \\
\# \text{when to stop waiting.} \\
\text{Markposition}[i] = \text{IF}(\text{condition, zero, Markposition}[i]) \ # \\
\# \text{zero out Markposition if we are done waiting} \\
\text{GO}\text{T}(\"postcode\", \text{AND}(\text{condition, waiting})) \ # \text{If} \\
\# \text{condition is one and we were waiting then go to} \\
\# \text{instruction after increment}
\]

\(^8\)Since this is a NAND++ program, we assume that if the label \( l \) is before the \texttt{GOTO} then jumping in execution means that another iteration of the program is finished, and the index variable \( i \) is increased or decreased as usual.
GOTO("end",waiting) # Otherwise, if we are still in waiting then go back to "end" skipping all the rest of the code
# (since this is another iteration of the program i keeps travelling as usual.)

Please make sure you understand the above construct. Also note that the above only works when there is a single line of the form \( i += \text{foo} \) or \( i -= \text{bar} \) in the program. When there are multiple lines then we need to add more labels and variables to take care of each one of them separately. Stopping here and working out how to handle more labels is an excellent way to get a better understanding of this construction.

**Implementing GOTO: the importance of doing nothing.** The above reduced the task of completing the proof of **Theorem 6.7** to implementing the GOTO function, but we have not yet shown how to do so. We now describe how we can implement GOTO in NAND++. The idea is simple: to simulate GOTO\((l,\text{cond})\), we modify all the lines between the GOTO and LABEL commands to do nothing if the condition is true. That is, we modify code of the form:

\[
\text{pre-code...} \\
\text{GOTO}(l,\text{cond}) \\
\text{between-code...} \\
\text{LABEL}(l) \\
\text{post-code...} \\
\text{to the form} \\
\text{pre-code ... donothing}_l = \text{cond} \\
\text{GUARDED(between-code,donothing}_l) \\
\text{donothing}_l = \text{zero} \\
\text{postcode...} \\
\text{where GUARDED(between-code,donothing}_l) \text{ refers to transforming every line in between-code from} \\
\text{...}
\]
the form $\text{foo} = \text{NAND}(\text{bar}, \text{blah})$ to the form $\text{foo} = \text{IF}(\text{donothing}_l, \text{foo}, \text{NAND}(\text{bar}, \text{blah}))$. That is, the “guarded” version of the code keeps the value of every variable the same if $\text{donothing}_l$ equals 1. We leave to you to verify that the above approach extends to multiple GOTO statements. This completes the proof of the second and final part of Theorem 6.7.

It is important to go over this proof and verify you understand it. One good way to do so is to understand how you the proof handles multiple GOTO statements. You can do so by eliminating one GOTO statement at a time. For every distinct label $l$, we will have a different variable $\text{donothing}_l$.

GOTO’s in programming languages  The GOTO statement was a staple of most early programming languages, but has largely fallen out of favor and is not included in many modern languages such as Python, Java, Javascript. In 1968, Edsger Dijkstra wrote a famous letter titled “Go to statement considered harmful.” (see also Fig. 6.5). The main trouble with GOTO is that it makes analysis of programs more difficult by making it harder to argue about invariants of the program.

When a program contains a loop of the form:

```python
for j in range(100):
    do something
    do blah
```

you know that the line of code `do blah` can only be reached if the loop ended, in which case you know that $j$ is equal to 100, and might also be able to argue other properties of the state of the program. In contrast, if the program might jump to `do blah` from any other point in the code, then it’s very hard for you as the programmer to know what you can rely upon in this code. As Dijkstra said, such invariants are important because “our intellectual powers are rather geared to master static relations and … our powers to visualize processes evolving in time are relatively poorly developed” and so “we should … do …our utmost best to shorten the conceptual gap between the static program and the dynamic process.”

That said, GOTO is still a major part of lower level languages where it is used to implement higher level looping constructs such as `while` and `for` loops. For example, even though Java doesn’t have a GOTO statement, the Java Bytecode (which is a lower level representation of Java) does have such a statement.
Similarly, Python bytecode has instructions such as `POP_JUMP_IF_TRUE` that implement the `GOTO` functionality, and similar instructions are included in many assembly languages. The way we use `GOTO` to implement a higher level functionality in NAND++ is reminiscent of the way these various jump instructions are used to implement higher level looping constructs.

![Figure 6.5](image.png)

**Figure 6.5:** XKCD’s take on the `GOTO` statement.

### 6.3.3 Well formed programs: The NAND++ style manual

The notion of passing between different variants of programs can be extremely useful, as often, given a program $P$ that we want to analyze, it would be simpler for us to first modify it to an equivalent program $P'$ that has some convenient properties. You can think of this as the NAND++ equivalent of enforcing “coding conventions” that are often used for programming languages. For example, while this is not part of the Python language, Google’s Python style guide stipulates that variables that are initialized to a value and never changed (i.e., constants) are typed with all capital letters. (Similar requirements are used in other style guides.) Of course this does not really restrict the power of Google-conforming Python programs, since every Python program can be transformed to an equivalent one that satisfies this requirement. In fact, many programming languages have automatic programs known as linters that can detect and sometimes modify the program to fit certain standards.

The following solved exercise is an example of that. We will define the notion of a well-formed program and show that every NAND++ program can be transformed into an equivalent one that is well formed.

**Definition 6.8 — Well-formed programs.** We say that an (enhanced or vanilla) NAND++ program $P$ is well formed if it satisfies the following properties:

- Every reference to a variable in $P$ either has the form `foo` or `foo_123` (a scalar variable: alphanumerical string starting with a lowercase letter and no brackets) or the form `Bar[i]`
or Bar_12[i] (an array variable alphanumerical string starting with a capital letter and ending with [i]).

• $P$ contains the scalar variables zero, one and indexincreasing such that zero and one are always the constants 0 and 1 respectively, and the program contains code that ensures that at the end of each iteration, indexincreasing is equal to 1 if in the next iteration i will increase by one above its current value, and is equal to 0 if in the next iteration i will decrease by one.

• $P$ contains the array variables Visited and Atstart and code to ensure that Atstart[i] equals 1 if and only if $i = 0$, and Visited[i] equals 1 for all the positions i such that the program finished an iteration with the index variable i equalling $i$.

• $P$ contains code to set loop to 1 at the beginning of the first iteration, and to ensure that if loop is ever set to 0 then it stays at 0, and moreover that if loop equals 0 then the values of Y and Yvalid cannot change.

The following exercise shows that we can transform every NAND++ program $P$ into a well-formed program $P'$ that is equivalent to it. Hence if we are given a NAND++ program $P$, we can (and will) often assume without loss of generality that it is well-formed.

Lemma 6.9 For every (enhanced or vanilla) NAND++ program $P$, there exists an (enhanced or vanilla, respectively) NAND++ program $P'$ equivalent to $P$ that is well formed as per Definition 6.8. That is, for every input $x \in \{0,1\}^*$, either both $P$ and $P'$ do not halt on $x$, or both $P$ and $P'$ halt on $x$ and produce the same output $y \in \{0,1\}^*$.

Solved Exercise 6.2 — Making an NAND++ program well formed. Prove Lemma 6.9

Solution: Since variable identifiers on their own have no meaning in (enhanced) NAND++ (other than the special ones $X$, $Xvalid$, $Y$, $Yvalid$ and loop, that already have the desired properties), we can easily achieve the property that scalars variables start with lowercase and arrays with uppercase using “search and replace”. We just have to take care that we don’t make two distinct identifiers become the same. For example, we can do so by changing all
scalar variable identifiers to lower case, and adding to them the prefix `scalar_` and adding the prefix `Array_` to all array variable identifiers.

The property that an array variable is never references with a numerical index is more challenging. We need to remove all references to an array variable with an actual numerical index rather than `i`. One thought might be to simply convert a a reference of the form `Arr[i]` to the scalar variable `arr_i`. However, this will not necessarily preserve the functionality of the program. The reason is that we want to ensure that when `i = 17` then `Arr[i]` would give us the same value as `arr_17`.

Nevertheless, we can use the approach above with a slight twist. We will demonstrate the solution in a concrete case. (Needless to say, if you needed to solve this question in a problem set or an exam, such a demonstration of a special case would not be sufficient; but this example should be good enough for you to extrapolate a full solution.) Suppose that there are only three references to array variables with numerical indices in the program: `Foo[5]`, `Bar[12]` and `Blah[22]`. We will include three scalar variables `foo_5`, `bar_12` and `blah_22` which will serve as a cache for the values of these arrays. We will change all references to `Foo[5]` to `foo_5`, `Bar[12]` to `bar_12` and so on and so forth. But in addition to that, whenever in the code we refer to `Foo[i]` we will check if `i = 5` and if so use the value `foo_5` instead, and similarly with `Bar[i]` or `Blah[i]`.

Specifically, we will change our program as follows. We will create an array `Is_5` such that `Is_5[i] = 1` if and only if `i = 5`, and similarly create arrays `Is_12`, `Is_22`.

We can then change code of the following form

```plaintext
Foo[i] = something
```

to

```plaintext
temp = something
foo_5 = IF(Is_5[i],temp,foo_5)
Foo[i] = temp
```

and similarly code of the form

```plaintext
blah = NAND(Bar[i],baz)
```

to
temp = If(Is_22[i], bar_22, Bar[i])
blah = NAND(temp, baz)

To create the arrays we can add code of the following form in the beginning of the program (here we’re using enhanced NAND++ syntax, GOTO, and the constant one but this syntactic sugar can of course be avoided):

```
# initialization of arrays
GOTO("program body", init_done)
i += one
i += one
i += one
i += one
i += one
Is_5[i] = one
i += one
... # repeat i += one 6 more times
Is_12[i] = one
i += one
... # repeat i += one 9 more times
Is_22[i] = one
i -= one
... # repeat i -= one 21 more times
init_done = one
```

Using IF statements (which can easily be replaced with syntactic sugar) we can handle the conditions that loop, Y, and Yvalid are not written to once loop is set to 0. We leave completing all the details as an exercise to the reader (see Exercise 6.1).

```
Using IF statements (which can easily be replaced with syntactic sugar) we can handle the conditions that loop, Y, and Yvalid are not written to once loop is set to 0. We leave completing all the details as an exercise to the reader (see Exercise 6.1).
```

6.4 TURING MACHINES

"Computing is normally done by writing certain symbols on paper. We may suppose that this paper is divided into squares like a child’s arithmetic book. The behavior of the [human] computer at any moment is determined by the symbols which he is observing, and of the ‘state of mind’ at that moment… We may suppose that in a simple operation not more than one symbol is altered.,”

“We compare a man in the process of computing … to a machine which is only capable of a finite number of configurations… The machine is supplied with a ‘tape’
This definitional choice does not make much difference since, as we show here, NAND++ programs are equivalent to Turing machines in their computing power.

Alan Turing was one of the intellectual giants of the 20th century. He was not only the first person to define the notion of computation, but also intimately involved in the use of computational devices as part of the effort to break the Enigma cipher during World War II, saving millions of lives. Tragically, Turing committed suicide in 1954, following his conviction in 1952 for homosexual acts and a court-mandated hormonal treatment. In 2009, British prime minister Gordon Brown made an official public apology to Turing, and in 2013 Queen Elizabeth II granted Turing a posthumous pardon. Turing’s life is the subject of a great book and a mediocre movie.

The “granddaddy” of all models of computation is the Turing Machine, which is the standard model of computation in most textbooks. Turing machines were defined in 1936 by Alan Turing in an attempt to formally capture all the functions that can be computed by human “computers” (see Fig. 6.7) that follow a well-defined set of rules, such as the standard algorithms for addition or multiplication.

Turing thought of such a person as having access to as much “scratch paper” as they need. For simplicity we can think of this scratch paper as a one dimensional piece of graph paper (or tape, as it is commonly referred to), which is divided to “cells”, where each “cell” can hold a single symbol (e.g., one digit or letter, and more generally some element of a finite alphabet). At any point in time, the person can read from and write to a single cell of the paper, and based on the contents can update his/her finite mental state, and/or move to the cell immediately to the left or right of the current one.

Thus, Turing modeled such a computation by a “machine” that

9 This definitional choice does not make much difference since, as we show here, NAND++ programs are equivalent to Turing machines in their computing power.

10 Alan Turing was one of the intellectual giants of the 20th century. He was not only the first person to define the notion of computation, but also intimately involved in the use of computational devices as part of the effort to break the Enigma cipher during World War II, saving millions of lives. Tragically, Turing committed suicide in 1954, following his conviction in 1952 for homosexual acts and a court-mandated hormonal treatment. In 2009, British prime minister Gordon Brown made an official public apology to Turing, and in 2013 Queen Elizabeth II granted Turing a posthumous pardon. Turing’s life is the subject of a great book and a mediocre movie.
Figure 6.7: Until the advent of electronic computers, the word “computer” was used to describe a person that performed calculations. These human computers were absolutely essential to many achievements including mapping the stars, breaking the Enigma cipher, and the NASA space mission. Two recent books about these human computers (which were more often than not women) and their important contributions are The Glass Universe (from which this photo is taken) and Hidden Figures.

Figure 6.8: Steam-powered Turing Machine mural, painted by CSE grad students the University of Washington on the night before spring qualifying examinations, 1987. Image from https://www.cs.washington.edu/building/art/SPTM.
maintains one of $k$ states, and at each point can read and write a single symbol from some alphabet $\Sigma$ (containing $\{0, 1\}$) from its “work tape”. To perform computation using this machine, we write the input $x \in \{0, 1\}^n$ on the tape, and the goal of the machine is to ensure that at the end of the computation, the value $F(x)$ will be written on the tape. Specifically, a computation of a Turing Machine $M$ with $k$ states and alphabet $\Sigma$ on input $x \in \{0, 1\}^n$ proceeds as follows:

- Initially the machine is at state 0 (known as the “starting state”) and the tape is initialized to $\triangleright, x_0, \ldots, x_{n-1}, \emptyset, \emptyset, \ldots$.\(^\text{11}\)
- The location $i$ to which the machine points to is set to 0.
- At each step, the machine reads the symbol $\sigma = T[i]$ that is in the $i^{th}$ location of the tape, and based on this symbol and its state $s$ decides on:
  - What symbol $\sigma'$ to write on the tape
  - Whether to move Left (i.e., $i \leftarrow i - 1$) or Right (i.e., $i \leftarrow i + 1$)
  - What is going to be the new state $s \in [k]$
- When the machine reaches the state $s = k - 1$ (known as the “halting state”) then it halts. The output of the machine is obtained by reading off the tape from location 1 onwards, stopping at the first point where the symbol is not 0 or 1.

![Figure 6.9](image)

**Figure 6.9** A Turing machine has access to a tape of unbounded length. At each point in the execution, the machine can read/write a single symbol of the tape, and based on that decide whether to move left, right or halt.

\(^{12}\)TODO: update figure to $\{0, \ldots, k - 1\}$.

\[\text{Example 6.10 — A Turing machine for palindromes.}\] Let $PAL$ (for *palindromes*) be the function that on input $x \in \{0, 1\}^*$, outputs 1 if and only if $x$ is an (even length) palindrome, in the sense that $x = w_0 \cdots w_{n-1}w_{n-1}w_{n-2} \cdots w_0$ for some $n \in \mathbb{N}$ and $w \in \{0, 1\}^n$.

We now show a Turing Machine $M$ that computes $PAL$. To specify $M$ we need to specify (i) $M$’s tape alphabet $\Sigma$ which should
contain at least the symbols 0, 1, $\triangleright$, and $\varnothing$, and (iii) $M$'s transition function which determines what action $M$ takes when it reads a given symbol while it is in a particular state.

In our case, $M$ will use the alphabet $\{0, 1, \triangleright, \varnothing, \times\}$ and will have $k = 14$ states. Though the states are simply numbers between 0 and $k - 1$, for convenience we will give them the following labels:

<table>
<thead>
<tr>
<th>State</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>START</td>
</tr>
<tr>
<td>1</td>
<td>RIGHT_0</td>
</tr>
<tr>
<td>2</td>
<td>RIGHT_1</td>
</tr>
<tr>
<td>3</td>
<td>LOOK_FOR_0</td>
</tr>
<tr>
<td>4</td>
<td>LOOK_FOR_1</td>
</tr>
<tr>
<td>5</td>
<td>RETURN</td>
</tr>
<tr>
<td>6</td>
<td>REJECT</td>
</tr>
<tr>
<td>7</td>
<td>ACCEPT</td>
</tr>
<tr>
<td>8</td>
<td>OUTPUT_0</td>
</tr>
<tr>
<td>9</td>
<td>OUTPUT_1</td>
</tr>
<tr>
<td>10</td>
<td>0_AND_BLANK</td>
</tr>
<tr>
<td>11</td>
<td>1_AND_BLANK</td>
</tr>
<tr>
<td>12</td>
<td>BLANK_AND_STOP</td>
</tr>
<tr>
<td>13</td>
<td>STOP</td>
</tr>
</tbody>
</table>

We describe the operation of our Turing Machine $M$ in words:

- $M$ starts in state $\text{START}$ and will go right, looking for the first symbol that is 0 or 1. If we find $\varnothing$ before we hit such a symbol then we will move to the $\text{OUTPUT}_1$ state that we describe below.

- Once $M$ found such a symbol $b \in \{0, 1\}$, $M$ deletes $b$ from the tape by writing the $\times$ symbol, it enters either the $\text{RIGHT}_0$ or $\text{RIGHT}_1$ mode according to the value of $b$ and starts moving rightwards until it hits the first $\varnothing$ or $\times$ symbol.

- Once we found this symbol we into the state $\text{LOOK_FOR}_b$ or $\text{LOOK_FOR}_1$ depending on whether we were in the state $\text{RIGHT}_0$ or $\text{RIGHT}_1$ and make one left move.

- In the state $\text{LOOK}_b$, we check whether the value on the tape is $b$. If it is, then we delete it by changing its value to $\times$, and move to the state $\text{RETURN}$. Otherwise, we change to the $\text{OUTPUT}_0$ state.

- The $\text{RETURN}$ state means we go back to the beginning. Specifically, we move leftward until we hit the first symbol that is not 0 or 1, in which case we change our state to $\text{START}$.

- The $\text{OUTPUT}_b$ states mean that we are going to output the value $b$. In both these states we go left until we hit $\triangleright$. Once we do so, we make a right step, and change to the $\text{1_AND_BLANK}$ or $\text{0_AND_BLANK}$ states respectively. In the latter states, we write the corresponding value, and then move right and change to the $\text{BLANK_AND_STOP}$ state, in which we write $\varnothing$ to the tape and move to the final $\text{STOP}$ state.

The above description can be turned into a table describing for each one of the $14 \cdot 5$ combination of state and symbol, what the
Turing machine will do when it is in that state and it reads that symbol. This table is known as the transition function of the Turing machine.

The formal definition of Turing machines is as follows:

**Definition 6.11 — Turing Machine.** A (one tape) Turing machine with \( k \) states and alphabet \( \Sigma \supseteq \{0, 1, \triangleright, \emptyset\} \) is a function \( M : [k] \times \Sigma \rightarrow [k] \times \Sigma \times \{\mathbb{L}, \mathbb{R}\} \).

For every \( x \in \{0, 1\}^* \), the output of \( M \) on input \( x \), denoted by \( M(x) \), is the result of the following process:

- We initialize \( T \) to be the sequence \( \triangleright, x_0, x_1, \ldots, x_{n-1}, \emptyset, \emptyset, \ldots \), where \( n = |x| \). (That is, \( T[0] = \triangleright, T[i+1] = x_i \) for \( i \in [n] \), and \( T[i] = \emptyset \) for \( i > n \).)

- We also initialize \( s = 0 \) and \( s = 0 \).

- We then repeat the following process as long as \( s \neq k - 1 \):
  1. Let \( (s', \sigma', D) = M(s, T[i]) \)
  2. Set \( s \rightarrow s', T[i] \rightarrow \sigma' \).
  3. If \( D = \mathbb{R} \) then set \( i \rightarrow i+1 \), if \( D = \mathbb{L} \) then set \( i \rightarrow \max\{i-1, 0\} \),

The result of the process is the string \( T[1], \ldots, T[m] \) where \( m > 0 \) is the smallest integer such that \( T[m+1] \notin \{0, 1\} \). If the process never ends then we denote the result by \( \bot \).

We say that the Turing machine \( M \) computes a (partial) function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \) if for every \( x \in \{0, 1\}^* \) on which \( F \) is defined, \( M(x) = F(x) \).

You should make sure you see why this formal definition corresponds to our informal description of a Turing Machine. To get more intuition on Turing Machines, you can play with some of the online available simulators such as Martin Ugarte’s, Anthony Morphett’s, or Paul Rendell’s.

**Transition functions vs computed functions, comparison to Sipser** One should not confuse the transition function of a Turing machine \( M \) with the function that the machine computes. The transition function is a finite function, with \( k|\Sigma| \) inputs and
2k|Σ| outputs. (Can you see why?) The machine can compute an infinite function \( F \) that takes as input a string \( x \in \{0,1\}^* \) of arbitrary length and might also produce an arbitrary length string as output.

In our formal definition, we identified the machine with its transition function since the transition function tells us everything we need to know about the Turing machine, and hence serves as a good mathematical representation of it. This choice of representation is somewhat arbitrary, and is based on our convention that the state space is always the numbers \( \{0, \ldots, k-1\} \), where we use 0 as our starting state and \( k - 1 \) as our halting state. Other texts use different conventions and so their mathematical definition of a Turing machine might look superficially different, although ultimately it describes the same computational process and has the same computational powers.

For example, Sipser’s text allows a more general set of states \( Q \) and allow to designate arbitrary elements of \( Q \) as starting and halting states, though by simple relabeling of the states one can see that this has no effect on the computational power of the model. Sipser also restricts attention to Turing machines that output only a single bit. In such cases, it is convenient to have two halting states: one of them is designated as the “0 halting state” (often known as the rejecting state) and the other as the “1 halting state” (often known as the accepting state). Thus instead of writing 0 or 1, the machine will enter into one of these states and halt. This again makes no difference to the computational power, though we prefer to consider the more general model with multi-bit outputs. Finally, Sipser considers also functions with input in \( \Sigma^* \) for an arbitrary alphabet \( \Sigma \) (and hence distinguishes between the input alphabet which he denotes as \( \Sigma \) and the tape alphabet which he denotes as \( \Gamma \)), while we restrict attention to functions with binary strings as input. The bottom line is that Sipser defines Turing machines as a seven tuple consisting of the state space, input alphabet, tape alphabet, transition function, starting state, accpeting state, and rejecting state. Yet, this is simply a different representation of the same concept, just as a graph can be represented in either adjacency list or adjacency matrix form.

6.4.1 Turing machines as programming languages

The name “Turing machine”, with its “tape” and “head” evokes a physical object, while a program is ultimately, a piece of text. But we can think of a Turing machine as a program as well. For example, consider the Turing Machine \( M \) of Example 6.10 that computes the
function $PAL$ such that $PAL(x) = 1$ iff $x$ is a palindrome. We can also describe this machine as a program using the Python-like pseudocode of the form below:

```python
# Gets an array Tape that is initialized to [">", x_0, x_1, ..., x_(n-1), "∅", "∅", ...]
# At the end of the execution, Tape[1] is equal to 1 if x is a palindrome and is equal to 0 otherwise
def PAL(Tape):
    head = 0
    state = 0  # START
    while (state != 13):
        if (state == 0 && Tape[head]=='0'):
            state = 3  # LOOK_FOR_0
            Tape[head] = 'x'
            head += 1  # move right
        if (state==0 && Tape[head]=='1')
            state = 4  # LOOK_FOR_1
            Tape[head] = 'x'
            head += 1  # move right
        ... # more if statements here
```

The particular details of this program are not important. What is important is that we can describe Turing machines as programs. Moreover, note that when translating a Turing machine into a program, the Tape becomes a list or array that can hold values from the finite set $\Sigma$. The head position can be thought of as an integer valued variable that can hold integers of unbounded size. In contrast, the current state can only hold one of a fixed number of values. In particular, if the number of states is $k$, then we can represent the state of the Turing machine using $\lceil \log k \rceil$ bits. Equivalently, if our programming language had only Boolean (i.e., 0/1-valued) variables, then we could replace the variable state with $\lceil \log k \rceil$ such variables. Similarly, we can represent each element of the alphabet $\Sigma$ using $\lceil \log |\Sigma| \rceil$ bits. Hence if our programming language had only Boolean valued arrays, we could replace the array Tape with $\lceil \log |\Sigma| \rceil$ such arrays.

6.4.2 Turing machines and NAND++ programs
Given the above discussion, it might not be surprising that Turing machines turn out to be equivalent to NAND++ programs. Nevertheless, this is an important result, and the first of many other such equivalence results we will see in this book.
Theorem 6.12 — Turing machines and NAND++ programs. For every $F : \{0,1\}^* \to \{0,1\}^*$, $F$ is computable by a NAND++ program if and only if there is a Turing Machine $M$ that computes $F$.

Proof Idea: Once again, to prove such an equivalence theorem, we need to show two directions. We need to be able to (1) transform a Turing machine $M$ to a NAND++ program $P$ that computes the same function as $P$ and (2) transform a NAND++ program $P$ into a Turing machine $M$ that computes the same function as $P$.

The idea of the proof is illustrated in Fig. 6.10. To show (1), given a Turing machine $M$, we will create a NAND program $P$ that will have an array $\text{Tape}$ for the tape of $M$ and scalar (i.e., non array) variable(s) $\text{state}$ for the state of $M$. Specifically, since the state of a Turing machine is not in $\{0,1\}$ but rather in a larger set $[k]$, we will use $\lceil \log k \rceil$ variables $\text{state}_0, \ldots, \text{state}_{\lceil \log k \rceil} - 1$ variables to store the representation of the state. Similarly, to encode the larger alphabet $\Sigma$ of the tape, we will use $\lceil \log |\Sigma| \rceil$ arrays $\text{Tape}_0, \ldots, \text{Tape}_{\lceil \log |\Sigma| \rceil} - 1$, such that the $i^{th}$ location of these arrays encodes the $i^{th}$ symbol in the tape for every tape. Using the fact that every function can be computed by a NAND program, we will be able to compute the transition function of $M$, replacing moving left and right by decrementing and incrementing $i$ respectively.

We show (2) using very similar ideas. Given a program $P$ that uses $a$ array variables and $b$ scalar variables, we will create a Turing machine with about $2^b$ states to encode the values of scalar variables, and an alphabet of about $2^a$ so we can encode the arrays using our tape. (The reason the sizes are only “about” $2^a$ and $2^b$ is that we will need to add some symbols and steps for bookkeeping purposes.) The Turing Machine $M$ will simulate each iteration of the program $P$ by updating its state and tape accordingly. *

Proof of Theorem 6.12. We now prove the “if” direction of Theorem 6.12, namely we show that given a Turing machine $M$, we can find a NAND++ program $P_M$ such that for every input $x$, if $M$ halts on input $x$ with output $y$ then $P_M(x) = y$. Because by Theorem 6.7 enhanced and plain NAND++ are equivalent in power, it is sufficient to construct an enhanced NAND++ program that has this property. Moreover, since our goal is just to show such a program $P_M$ exists, we don’t need to write out the full code of $P_M$ line by line, and can take advantage of our various “syntactic sugar” in describing it.

The key observation is that by Theorem 4.6 we can compute every finite function using a NAND program. In particular, consider the function $M : [k] \times \Sigma \to [k] \times \Sigma \times \{\text{L, R}\}$ corresponding to our Turing
Machine. We can encode \([k]\) using \(\{0,1\}^\ell\), \(\Sigma\) using \(\{0,1\}^{\ell'}\), and \(\{\mathbb{L},\mathbb{R}\}\) using \(\{0,1\}\), where \(\ell = \lceil \log k \rceil\) and \(\ell' = \lceil \log |\Sigma| \rceil\). Hence we can identify \(M\) with a function \(\overline{M} : \{0,1\}^\ell \times \{0,1\}^{\ell'} \rightarrow \{0,1\}^\ell \times \{0,1\}^{\ell'} \times \{0,1\}\), and by Theorem 4.6 there exists a finite length NAND program \(\text{Compute}_M\) that computes this function \(\overline{M}\). The enhanced NAND++ program to simulate \(M\) will be the following:

```nand++

label("mainloop")
state, Tape[i], direction = ComputeM(state, Tape[i])
i += direction
i -= NOT(direction) # like in TM's, this does
    nothing if i=0
goto("mainloop", NOTEQUAL(state, k-1))
copy Tape to Y/Yvalid..
```

where we use `state` as shorthand for the tuple of variables `state_0, ..., state_\ell - 1` and `Tape[i]` as shorthand for `Tape_0[i]`, `..., Tape_{\ell'} - 1[i]` where \(\ell = \lceil \log k \rceil\) and \(\ell' = \lceil \log |\Sigma| \rceil\).

In the description above we also take advantage of our \text{GOTO} syntactic sugar as well as having access to the \text{NOTEQUAL} function to compare two strings of length \(\ell\). Copying \(X[0], ..., X[n-1]\) (where \(n\) is the smallest integer such that \(Xvalid[n] = 0\)) to locations `Tape[1]`, `..., Tape[n]` can be done by a simple loop, and we can use a similar loop at the end to copy the tape into the `Y` array (marking where to stop using `Yvalid`). Since every step of the main loop of the above program perfectly mimics the computation of the Turing Machine \(M\) as \(\text{Compute}_M\) computes the transition of the Turing Machine, and the
program carries out exactly the definition of computation by a Turing Machine as per Definition 6.11.

For the other direction, suppose that $P$ is a (standard) NAND++ program with $s$ lines, $\ell$ scalar variables, and $\ell'$ array variables. We will show that there exists a Turing machine $M_P$ with $2^{2\ell'} + 2C$ states and alphabet $\Sigma$ of size $C' + 2^{2\ell'}$ that computes the same functions as $P$ (where $C, C'$ are some constants to be determined later). Specifically, consider the function $\overline{P} : \{0,1\}^\ell \times \{0,1\}^\ell' \to \{0,1\}^\ell \times \{0,1\}^\ell'$ that on input the contents of $P$’s scalar variables and the contents of the array variables at location $i$ in the beginning of an iteration, outputs all the new values of these variables at the end of the iteration.

We can assume without loss of generality that $P$ contains the variables $\text{indexincreasing}, \text{Atzero}$ and $\text{Visited}$ as we’ve seen before, and so we can compute whether $i$ will increase or decrease based on the state of these variables. Also note that $\text{loop}$ is one of the scalar variables of $P$. Hence the Turing machine can simulate an execution of $P$ in one iteration using a finite function applied to its alphabet. The overall operation of the Turing machine will be as follows:

1. The machine $M_M$ encodes the contents of the array variables of $P$ in its tape, and the contents of the scalar variables in (part of) its state.

2. Initially, the machine $M_P$ will scan the input and copy the result to the parts of the tape corresponding to the $X$ and $X\text{valid}$ variables of $P$. (We use some extra states and alphabet symbols to achieve this.)

3. The machine will $M_P$ then simulates each iteration of $P$ by applying the constant function to update the state and the location of the head, as long as the $\text{loop}$ variable of $P$ equals 1.

4. When the loop variable equals 1, the machine $M_P$ will scan the output arrays and copy them to the beginning of the tape. (Again we can add some states and alphabet symbols to achieve this.)

5. At the end of this scan the machine $M_P$ will enter its halting state.

The above is not a full formal description of a Turing Machine, but our goal is just to show that such a machine exists. One can see that $M_M$ simulates every step of $P$, and hence computes the same function as $P$. 

R Turing Machines and NAND++ programs Once you understand the definitions of both NAND++ programs and Turing Machines, Theorem 6.12 is fairly straightforward. Indeed, NAND++ programs are not as much a different model from Turing Machines.
as a reformulation of the same model in programming language notation. Specifically, NAND++ programs correspond to a type of Turing Machines known as single tape oblivious Turing machines.

Running time equivalence (optional) If we examine the proof of Theorem 6.12 then we can see that the equivalence between NAND++ programs and Turing machines is up to polynomial overhead in the number of steps required to compute the function. Specifically, in the Transformation of a NAND++ program to a Turing machine we used one step of the machine to compute one iteration of the NAND++ program, and so if the NAND++ program \( P \) took \( T \) iterations to compute the function \( F \) on some input \( x \in \{0,1\}^n \) and \( |F(x)| = m \), then the number of steps that the Turing machine \( M_P \) takes is \( O(T + n + m) \) (where the extra \( O(n + m) \) is to copy the input and output). In the other direction, our program to simulate a machine \( M \) took one iteration to simulate a step of \( M \), but we used some syntactic sugar, and in particular allowed ourselves to use an enhanced NAND++ program. A careful examination of the proof of Theorem 6.7 shows that our transformation of an enhanced to a standard NAND++ (using the “breadcrumbs” and “wait for the bus” strategies) would at the worst case expand \( T \) iterations into \( O(T^2) \) iterations. This turns out the most expensive step of all the other syntactic sugar we used. Hence if the Turing machine \( M \) takes \( T \) steps to compute \( F(x) \) (where \( |x| = n \) and \( |F(x)| = m \)) then the (standard) NAND++ program \( P_M \) will take \( O(T^2 + n + m) \) steps to compute \( F(x) \). We will come back to this question of measuring number of computation steps later in this course. For now the main take away point is that NAND++ programs and Turing Machines are roughly equivalent in power even when taking running time into account.

6.5 UNIFORMITY, AND NAND VS NAND++ (DISCUSSION)

While NAND++ adds an extra operation over NAND, it is not exactly accurate to say that NAND++ programs are “more powerful” than NAND programs. NAND programs, having no loops, are simply not applicable for computing functions with more inputs than they have lines. The key difference between NAND and NAND++ is that NAND++ allows us to express the fact that the algorithm for computing parities of length-100 strings is really the same one as the algorithm for computing parities of length-5 strings (or similarly the fact that the algorithm for adding \( n \)-bit numbers is the same for every
That is, one can think of the NAND++ program for general parity as the “seed” out of which we can grow NAND programs for length 10, length 100, or length 1000 parities as needed. This notion of a single algorithm that can compute functions of all input lengths is known as uniformity of computation and hence we think of NAND++ as uniform model of computation, as opposed to NAND which is a nonuniform model, where we have to specify a different program for every input length.

Looking ahead, we will see that this uniformity leads to another crucial difference between NAND++ and NAND programs. NAND++ programs can have inputs and outputs that are longer than the description of the program and in particular we can have a NAND++ program that “self replicates” in the sense that it can print its own code. This notion of “self replication”, and the related notion of “self reference” is crucial to many aspects of computation, as well of course to life itself, whether in the form of digital or biological programs.

For now, what you ought to remember is the following differences between uniform and non uniform computational models:

- **Non uniform computational models:** Examples are NAND programs and Boolean circuits. These are models where each individual program/circuit can compute a finite function \( F : \{0,1\}^n \to \{0,1\}^m \). We have seen that every finite function can be computed by some program/circuit. To discuss computation of an infinite function \( F : \{0,1\}^* \to \{0,1\}^* \) we need to allow a sequence \( \{P_n\}_{n \in \mathbb{N}} \) of programs/circuits (one for every input length), but this does not capture the notion of a single algorithm to compute the function \( F \).

- **Uniform computational models:** Examples are (standard or enhanced) NAND++ programs and Turing Machines. These are models where a single program/machine can take inputs of arbitrary length and hence compute an infinite function \( F : \{0,1\}^* \to \{0,1\}^* \). The number of steps that a program/machine takes on some input is not a priori bounded in advance and in particular there is a chance that it will enter into an infinite loop. Unlike the nonuniform case, we have not shown that every infinite function can be computed by some NAND++ program/Turing Machine. We will come back to this point in Chapter 8.

Lecture Recap

- NAND++ programs introduce the notion of loops, and allow us to capture a single algorithm that can evaluate functions of any input length.
- Enhanced NAND++ programs, which allow control on the index variable \( i \), are equivalent in...
power to standard NAND++ programs.

• NAND++ programs are also equivalent in power to Turing machines.

• Running a NAND++ program for any finite number of steps corresponds to a NAND program. However, the key feature of NAND++ is that the number of iterations can depend on the input, rather than being a fixed upper bound in advance.

6.6 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 6.1 — Well formed NAND++ programs. Complete ?? in the vanilla NAND++ case to give a full proof that for every standard (i.e., non-enhanced) NAND++ program $P$ there exists a standard NAND++ program $P'$ such that $P'$ is well formed and $P'$ is equivalent to $P$.

Exercise 6.2 — Single vs multiple bit. Prove that for every $F : \{0, 1\}^* \to \{0, 1\}^*$, the function $F$ is computable if and only if the following function $G : \{0, 1\}^* \to \{0, 1\}$ is computable, where $G$ is defined as follows:

$$
G(x, i, \sigma) = \begin{cases} 
F(x)_i & i < |F(x)|, \sigma = 0 \\
1 & i < |F(x)|, \sigma = 1 \\
0 & i \geq |F(x)|
\end{cases}
$$

6.7 BIBLIOGRAPHICAL NOTES

Salil Vadhan proposed the following analytically easier to describe sequence for NAND++: $INDEX(\ell) = \min\{\ell - [\sqrt{\ell}]^2, [\sqrt{\ell}]^2 - \ell\}$ which has the form $0, 0, 1, 1, 0, 1, 2, 2, 1, 0, 1, 2, 3, 3, 2, 1, 0, 1, 2, 3, 4, 4, 3, 2, 1, 0, ….$

6.8 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

6.9 ACKNOWLEDGEMENTS
7

Equivalent models of computation

“All problems in computer science can be solved by another level of indirection”, attributed to David Wheeler.

“Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction. This distinction and a notation for describing it, from which we deviate trivially, is given by Church.”, John McCarthy, 1960 (in paper describing the LISP programming language)

So far we have defined the notion of computing a function based on the rather esoteric NAND++ programming language. While we have shown this is equivalent with Turing machines, the latter also don’t correspond closely to the way computation is typically done these days. In this chapter we justify this choice by showing that the definition of computable functions will remain the same under a wide variety of computational models. In fact, a widely believed claim known as the Church-Turing Thesis holds that every “reasonable” definition of computable function is equivalent to ours. We will discuss the Church-Turing Thesis and the potential definitions of “reasonable” in Section 7.6.

7.1 RAM MACHINES AND NAND++

One of the limitations of NAND++ (and Turing machines) is that we can only access one location of our arrays/tape at a time. If currently \( i = 22 \) and we want to access \( Foo[957] \) then it will take us at least 923 steps to get there. In contrast, almost every programming language has a formalism for directly accessing memory locations. Hardware implementations also provide so called Random Access Memory (RAM)
which can be thought of as a large array $\text{Mem}$, such that given an index $p$ (i.e., memory address, or a pointer), we can read from and write to the $p^{th}$ location of $\text{Mem}$.\footnote{“Random access memory” is quite a misnomer, since it has nothing to do with probability. Alas at this point the term is quite entrenched. Still, we will try to call use the term indexed access instead.}

The computational model that allows access to such a memory is known as a RAM machine (sometimes also known as the Word RAM model). In this model the memory is an array of unbounded size where each cell can store a single word, which we think of as a string in $\{0,1\}^w$ and also as a number in $[2^w]$. The parameter $w$ is known as the word size and is chosen as some function of the input length $n$. A typical choice is that $w = c \log n$ for some constant $c$. This is sometimes known as the “transdichotomous RAM model”. In addition to the memory array, the word RAM model also contains a constant number of registers $r_1, \ldots, r_k$ that also contain a single word. The operations in this model include loops, arithmetic on registers, and reading and writing from a memory location addressed by the register.

We will use an extension of NAND++ to capture the RAM model. Specifically, we define the NAND« programming language as follows:

- The variables are allowed to be (non negative) integer valued rather than only Boolean. That is, a scalar variable foo holds an non negative integer in $\mathbb{N}$ (rather than only a bit in $\{0,1\}$), and an array variable Bar holds an array of integers.

- We allow indexed access to arrays. If foo is a scalar and Bar is an array, then Bar[foo] refers to the location of Bar indexed by the value of foo.

- As is often the case in programming language, we will assume that for Boolean operations such as NAND, a zero valued integer is considered as false, and a nonzero valued integer is considered as true.

- In addition to NAND we will also allow the basic arithmetic operations of addition, subtraction, multiplication, (integer) division, as well as comparisons (equal, greater than, less than, etc..)

- We will also include as part of the language basic control flow structures such as if and while.

The full description of the NAND« programing language is in the appendix.\footnote{One restriction mentioned there is that the integer values in a variable always range between 0 and $T - 1$ where $T$ is the number of steps the program took so far. Hence all the arithmetic operations will “truncate” their results so that the output is in this range. This restriction does not make a difference for any of the discussion in this chapter, but will help us make a more accurate accounting of the running time in the future.} However, the most important fact you need to know about it is the following:

\textbf{Theorem 7.1} — NAND++ (TM’s) and NAND« (RAM) are equivalent. For
every function $F: \{0, 1\}^* \to \{0, 1\}^*$, $F$ is computable by a NAND++ program if and only if $F$ is computable by a NAND« program.

**Proof Idea:** Clearly NAND« is only more powerful than NAND++, and so if a function $F$ is computable by a NAND++ program then it can be computed by a NAND« program. The challenging direction is of course to transform a NAND« program $P$ to an equivalent NAND++ program $Q$. To describe the proof in full we will need to cover the full formal specification of the NAND« language, and show how we can implement every one of its features as syntactic sugar on top of NAND++.

This can be done but going over all the operations in detail is rather tedious. Hence we will focus on describing the main ideas behind this transformation. The transformation has two steps:

1. **Indexed access of bit arrays:** NAND« generalizes NAND++ in two main ways: (a) adding indexed access to the arrays (i.e., $\text{Foo[bar]}$ syntax) and (b) moving from Boolean valued variables to integer valued ones. We will start by showing how to handle (a). Namely, we will show how we can implement in NAND++ the operation $\text{Setindex(Bar)}$ such that if $\text{Bar}$ is an array that encodes some integer $i$, then after executing $\text{Setindex(Bar)}$ the value of $i$ will equal to $j$. This will allow us to simulate syntax of the form $\text{Foo[Bar]}$ by $\text{Setindex(Bar)}$ followed by $\text{Foo[i]}$.

2. **Two dimensional bit arrays:** We will then show how we can use “syntactic sugar” to augment NAND++ with two dimensional arrays. That is, have two indices $i$ and $j$ and two dimensional arrays, such that we can use the syntax $\text{Foo[i][j]}$ to access the $(i, j)$-th location of Foo.

3. **Arrays of integers:** Finally we will encode a one dimensional array $\text{Arr of integers}$ by a two dimensional $\text{Arrbin of bits}$. The idea is simple: if $a_{i,0}, \ldots, a_{i,\ell}$ is a binary (prefix-free) representation of $\text{Arr[i]}$, then $\text{Arrbin[i][j]}$ will be equal to $a_{i,j}$.

Once we have arrays of integers, we can use our usual syntactic sugar for functions, $\text{GOTO}$ etc. to implement the arithmetic and control flow operations of NAND«.

We do not show the full formal proof of Theorem 7.1 but focus on the most important parts: implementing indexed access, and simulating two dimensional arrays with one dimensional ones.

### 7.1.1 Indexed access in NAND++
Let us choose some prefix-free representation for the natural numbers (see Section 2.3.2). For example, if a natural number $k$ is equal
to $\sum_{i=0}^{\ell} k_i \cdot 2^i$ for $\ell = \lceil \log k \rceil$, then we can represent it as the string $(k_0, k_0, k_1, k_1, \ldots, k_\ell, k_\ell, 1, 0)$.

To implement indexed access in NAND++, we need to be able to do the following. Given an array $\text{Bar}$, implement to operation $\text{Setindex(Bar)}$ that will set $i$ to the value encoded by $\text{Bar}$. This can be achieved as follows:

1. Set $i$ to zero, by decrementing it until we reach the point where $\text{Atzero}[i] = 1$ (where $\text{Atzero}$ is an array that has 1 only in position 0).

2. Let $\text{Temp}$ be an array encoding the number 0.

3. While the number encoded by $\text{Temp}$ differs from the number encoded by $\text{Bar}$:
   
   (a) Increment $\text{Temp}$
   
   (b) Set $i += \text{one}$

   At the end of the loop, $i$ is equal to the value at $\text{Bar}$, and so we can use this to read or write to arrays at the location corresponding to this value. In code, we can implement the above operations as follows:

   ```
   # set i to 0, assume Atzero, one are initialized
   LABEL("zero_idx")
   i -= one
   GOTO("zero_idx",NOT(Atzero[i]))
   ... 

   # zero out temp
   #(code below assumes a specific prefix-free encoding
   in which 10 is the "end marker")
   Temp[0] = 1
   Temp[1] = 0
   # set i to Bar, assume we know how to increment,
   # compare
   LABEL("increment_temp")
   cond = EQUAL(Temp,Bar)
   i += cond
   INC(Temp)
   GOTO("increment_temp",cond)
   # if we reach this point, i is number encoded by
   Bar
   ... 
   ```
7.1.2 Two dimensional arrays in NAND++

To implement two dimensional arrays, we embed them in a one dimensional array. The idea is that we come up with a one to one function \( \text{embed} : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \), and so embed the location \((i, j)\) of the two dimensional array \(\text{Two}\) in the location \(\text{embed}(i, j)\) of the array \(\text{One}\).

Since the set \(\mathbb{N} \times \mathbb{N}\) seems “much bigger” than the set \(\mathbb{N}\), a priori it might not be clear that such a one to one mapping exists. However, once you think about it more, it is not that hard to construct. For example, you could ask a child to use scissors and glue to transform a 10” by 10” piece of paper into a 1” by 100” strip. If you think about it, this is essentially a one to one map from \([10] \times [10]\) to \([10]\). We can generalize this to obtain a one to one map from \([n] \times [n]\) to \([n^2]\) and more generally a one to one map from \(\mathbb{N} \times \mathbb{N}\) to \(\mathbb{N}\). Specifically, the following map \(\text{embed}\) would do (see Fig. 7.1):

\[
\text{embed}(x, y) = \frac{1}{2} (x + y)(x + y + 1) + x .
\]  

(7.1)

We ask you to prove that \(\text{embed}\) is indeed one to one, as well as computable by a NAND++ program, in Exercise 7.1.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.1: Illustration of the map \(\text{embed}(x, y) = \frac{1}{2}(x + y)(x + y + 1) + x\) for \(x, y \in [10]\); one can see that for every distinct pairs \((x, y)\) and \((x', y')\), \(\text{embed}(x, y) \neq \text{embed}(x', y')\).

So, we can replace code of the form \(\text{Two}[\text{Foo}][\text{Bar}] = \text{something}\) (i.e., access the two dimensional array \(\text{Two}\) at the integers encoded by the one dimensional arrays \(\text{Foo}\) and \(\text{Bar}\)) by code of the form:

\[
\text{Blah} = \text{embed}(\text{Foo}, \text{Bar})
\]

\[
\text{Setindex}(\text{Blah})
\]

\[
\text{Two}[i] = \text{something}
\]

Computing \(\text{embed}\) is left for you the reader as Exercise 7.1, but let us hint that this can be done by simply following the grade-school algorithms for multiplication, addition, and division.
7.1.3 All the rest

Once we have two dimensional arrays and indexed access, simulating NAND« with NAND++ is just a matter of implementing the standard algorithms for arithmetic operations and comparators in NAND++. While this is cumbersome, it is not difficult, and the end result is to show that every NAND« program \( P \) can be simulated by an equivalent NAND++ program \( Q \), thus completing the proof of Theorem 7.1.

7.1.4 Turing equivalence (discussion)

Any of the standard programming language such as C, Java, Python, Pascal, Fortran have very similar operations to NAND«. (Indeed, ultimately they can all be executed by machines which have a fixed number of registers and a large memory array.) Hence using Theorem 7.1, we can simulate any program in such a programming language by a NAND++ program. In the other direction, it is a fairly easy programming exercise to write an interpreter for NAND++ in any of the above programming languages. Hence we can also simulate NAND++ programs (and so by Theorem 6.12, Turing machines) using these programming languages. This property of being equivalent in power to Turing Machines / NAND++ is called Turing Equivalent (or sometimes Turing Complete). Thus all programming languages we are familiar with are Turing equivalent.\(^3\)

Recursion in NAND« (advanced) One concept that appears in some of these languages but we did not include in NAND« programs is recursion. However, recursion (and function calls in general) can be implemented in NAND« using the stack data structure. A stack is a data structure containing a sequence of elements, where we can “push” elements into it and “pop” them from it in “first in last out” order. We can implement a stack by an array of integers \( \text{Stack} \) and a scalar variable \( \text{stackpointer} \) that will be

---

\(^3\) Some programming language have hardwired fixed (even if extremely large) bounds on the amount of memory they can access, which formally prevent them from being applicable to computing infinite functions and hence simulating Turing machines. We ignore such issues in this discussion and assume access to some storage device without a fixed upper bound on its capacity.
the number of items in the stack. We implement push(foo) by

\[ \text{Stack}[\text{stackpointer}] = \text{foo} \]
\[ \text{stackpointer} += 1 \]

and implement bar = pop() by

\[ \text{bar} = \text{Stack}[\text{stackpointer}] \]
\[ \text{stackpointer} -= 1 \]

We implement a function call to \( F \) by pushing the arguments for \( F \) into the stack. The code of \( F \) will “pop” the arguments from the stack, perform the computation (which might involve making recursive or non recursive calls) and then “push” its return value into the stack. Because of the “first in last out” nature of a stack, we do not return control to the calling procedure until all the recursive calls are done.

The fact that we can implement recursion using a non recursive language is not surprising. Indeed, machine languages typically do not have recursion (or function calls in general), and a compiler implements function calls using a stack and GOTO. You can find online tutorials on how recursion is implemented via stack in your favorite programming language, whether it’s Python, JavaScript, or Lisp/Scheme.

7.2 THE “BEST OF BOTH WORLDS” PARADIGM (DISCUSSION)

The equivalence between NAND++ and NAND« allows us to choose the most convenient language for the task at hand:

- When we want to give a theorem about all programs, we can use NAND++ because it is simpler and easier to analyze. In particular, if we want to show that a certain function \textit{can not} be computed, then we will use NAND++.

- When we want to show the existence of a program computing a certain function, we can use NAND«, because it is higher level and easier to program in. In particular, if we want to show that a function \textit{can} be computed then we can use NAND«. In fact, because NAND« has much of the features of high level programming languages, we will often describe NAND« programs in an informal manner, trusting that the reader can fill in the details and translate the high level description to the precise program. (This is just like the way people typically use informal or “pseudocode” descriptions of algorithms, trusting that their audience will know to translate these descriptions to code if needed.)
Our usage of NAND++ and NAND« is very similar to the way people use in practice high and low level programming languages. When one wants to produce a device that executes programs, it is convenient to do so for very simple and “low level” programming language. When one wants to describe an algorithm, it is convenient to use as high level a formalism as possible.

Figure 7.3: By having the two equivalent languages NAND++ and NAND«, we can “have our cake and eat it too”, using NAND++ when we want to prove that programs can’t do something, and using NAND« or other high level languages when we want to prove that programs can do something.

7.2.1 Let’s talk about abstractions.

“The programmer is in the unique position that … he has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before.”, Edsger Dijkstra, “On the cruelty of really teaching computing science”, 1988.

At some point in any theory of computation course, the instructor and students need to have the talk. That is, we need to discuss the level of abstraction in describing algorithms. In algorithms courses, one typically describes algorithms in English, assuming readers can “fill in the details” and would be able to convert such an algorithm into an implementation if needed. For example, we might describe the
**breadth first search** algorithm to find if two vertices $u, v$ are connected as follows:

1. Put $u$ in queue $Q$.
2. While $Q$ is not empty:
   - Remove the top vertex $w$ from $Q$
   - If $w = v$ then declare “connected” and exit.
   - Mark $w$ and add all unmarked neighbors of $w$ to $Q$.

Declare “unconnected”.

We call such a description a **high level description**.

If we wanted to give more details on how to implement breadth first search in a programming language such as Python or C (or NAND« / NAND++ for that matter), we would describe how we implement the queue data structure using an array, and similarly how we would use arrays to implement the marking. We call such an “intermediate level” description an implementation level or pseudocode description. Finally, if we want to describe the implementation precisely, we would give the full code of the program (or another fully precise representation, such as in the form of a list of tuples). We call this a **formal or low level** description.

While initially we might have described NAND, NAND++, and NAND« programs at the full formal level (and the NAND website contains more such examples), as the course continues we will move to implementation and high level description. After all, our focus is typically not to use these models for actual computation, but rather to analyze the general phenomenon of computation. That said, if you don’t understand how the high level description translates to an actual implementation, you should always feel welcome to ask for more details of your teachers and teaching fellows.

A similar distinction applies to the notion of representation of objects as strings. Sometimes, to be precise, we give a low level specification of exactly how an object maps into a binary string. For example, we might describe an encoding of $n$ vertex graphs as length $n^2$ binary strings, by saying that we map a graph $G$ over the vertices $[n]$ to a string $x \in \{0, 1\}^{n^2}$ such that the $n \cdot i + j$-th coordinate of $x$ is 1 if and only if the edge $i \rightarrow j$ is present in $G$. We can also use an intermediate or implementation level description, by simply saying that we represent a graph using the adjacency matrix representation.

Finally, because we are translating between the various representations of graphs (and objects in general) can be done via a NAND«
Figure 7.4: We can describe an algorithm at different levels of granularity/detail and precision. At the highest level we just write the idea in words, omitting all details on representation and implementation. In the intermediate level (also known as implementation or pseudocode) we give enough details of the implementation that would allow someone to derive it, though we still fall short of providing the full code. The lowest level is where the actual code or mathematical description is fully spelled out. These different levels of detail all have their uses, and moving between them is one of the most important skills for a computer scientist.

(and hence a NAND++) program, when talking in a high level we also suppress discussion of representation altogether. For example, the fact that graph connectivity is a computable function is true regardless of whether we represent graphs as adjacency lists, adjacency matrices, list of edge-pairs, and so on and so forth. Hence, in cases where the precise representation doesn’t make a difference, we would often talk about our algorithms as taking as input an object \( O \) (that can be a graph, a vector, a program, etc.) without specifying how \( O \) is encoded as a string.

7.3 \textsc{Lambda Calculus and Functional Programming Languages}

The \( \lambda \) calculus is another way to define computable functions. It was proposed by Alonzo Church in the 1930’s around the same time as Alan Turing’s proposal of the Turing Machine. Interestingly, while Turing Machines are not used for practical computation, the \( \lambda \) calculus has inspired functional programming languages such as LISP, ML and Haskell, and indirectly the development of many other programming languages as well. In this section we will present the \( \lambda \) calculus and show that its power is equivalent to NAND++ programs (and hence also to Turing machines). An online appendix contains a Jupyter notebook with a Python implementation of the \( \lambda \) calculus that you can...
The $\lambda$ operator. At the core of the $\lambda$ calculus is a way to define “anonymous” functions. For example, instead of defining the squaring function as

$$\text{square}(x) = x \times x$$  \hspace{1cm} (7.2)

we write it as

$$\lambda x. x \times x$$  \hspace{1cm} (7.3)

and so $(\lambda x. x \times x)(7) = 49$.

To reduce notational clutter, when writing $\lambda$ calculus expression we often drop the parenthesis for function evaluation. Hence instead of writing $f(x)$ for the result of applying the function $f$ to the input $x$, we can also write this as simply $f(x)$. Therefore we can write $(\lambda x. x \times x)(7) = 49$. In this chapter, we will use both the $f(x)$ and $f \ x$ notations for function application.

Clearly, the name of the argument to a function doesn’t matter, and so $\lambda y. y \times y$ is the same as $\lambda x. x \times x$, as both are ways to write the squaring function.

We can also apply functions on functions. For example, can you guess what number is the following expression equal to?

$$(((\lambda f. (\lambda y. (f \ y)))(\lambda x. x \times x)) \ 3)$$  \hspace{1cm} (7.4)

The expression Eq. (7.4) might seem daunting, but before you look at the solution below, try to break it apart to its components, and evaluate each component at a time. Working out this example would go a long way toward understanding the $\lambda$ calculus.

Example 7.2 — Working out a $\lambda$ expression. To understand better the $\lambda$ calculus. Let’s evaluate Eq. (7.4), one step at a time. As nice as it is for the $\lambda$ calculus to allow us anonymous functions, for complicated expressions adding names can be very helpful for understanding. So, let us write $F = \lambda f. (\lambda y. (f \ y))$ and $g = \lambda x. x \times x$. 
Therefore Eq. (7.4) becomes

\[(F g) 3.\] (7.5)

On input a function \(f, F\) outputs the function \(\lambda y.(f(f(y)))\), which in more standard notation is the mapping \(y \mapsto f(f(y))\). Our function \(g\) is simply \(g(x) = x^2\) and so \((F g)\) is the function that maps \(y\) to \((y^2)^2\) or in other words to \(y^4\). Hence \(((F g)3) = 3^4 = 81\).

\[\text{Obtaining multi-argument functions via Currying.}\]
The expression \(e\) can itself involve \(\lambda\), and so for example the function

\[\lambda x.(\lambda y.x + y)\] (7.6)

maps \(x\) to the function \(y \mapsto x + y\).

In particular, if we invoke this function on \(a\) and then invoke the result on \(b\) then we get \(a + b\). We can use this approach to achieve the effect of functions with more than one input and so we will use the shorthand \(\lambda x, y.e\) for \(\lambda x.(\lambda y.e)\). Similarly, we will use \(f(x, y, z)\) as shorthand for \(((f x) y) z\) or equivalently (since function application will associate left to right) \(fxyz\). 4

\[\text{Example 7.3 — Simplifying a } \lambda \text{ expression.}\]
Here is another example of a \(\lambda\) expression:

\[((\lambda x.(\lambda y.x)) 2) 9\]. (7.7)

\[\text{Figure 7.5: In the “currying” transformation, we can create the effect of a two parameter function } f(x, y) \text{ with the } \lambda \text{ expression } \lambda x. (\lambda y.f(x, y)) \text{ which on input } x \text{ outputs a one-parameter function } f_x \text{ that has } x \text{ “hardwired” into it and such that } f_x(y) = f(x, y). \text{ This can be illustrated by a circuit diagram; see Chelsea Voss’s site.}\]
Let us denote \((\lambda y.x)\) by \(F\). Then Eq. (7.7) has the form

\[
((\lambda x. F) \, 2) \, 9
\]

(7.8)

Now \((\lambda x. F)\) is equal to \(F[x \rightarrow 2]\). Since \(F\) is \(\lambda y. x\) this means that \((\lambda x. F)\) is the function \(\lambda y. 2\) that ignores its input and outputs 2 no matter what it is equal to. Hence Eq. (7.7) is equivalent to \((\lambda y. 2) \, 9\) which is the result of applying the function \(y \rightarrow 2\) on the input 9, which is simply the number 2.

### 7.3.1 Formal description of the \(\lambda\) calculus.

In the \(\lambda\) calculus we start with “basic expressions” that contain a single variable such as \(x\) or \(y\) and build more complex expressions using the following two rules:

- **Application**: If \(\text{exp}\) and \(\text{exp}'\) are \(\lambda\) expressions, then the \(\lambda\) expression \((\text{exp} \, \text{exp}')\) corresponds to applying the function described by \(\text{exp}\) to the input \(\text{exp}'\).

- **Abstraction**: If \(\text{exp}\) is a \(\lambda\) expression and \(x\) is a variable, then the \(\lambda\) expression \(\lambda x. (\text{exp})\) corresponds to the function that on any input \(z\) returns the expression \(\text{exp}[x \rightarrow z]\) replacing all (free) occurrences of \(x\) in \(\text{exp}\).

We can now formally define \(\lambda\) expressions:

**Definition 7.4 — \(\lambda\) expression.** A \(\lambda\) expression is either a single variable identifier or an expression that is built from other expressions using the application and abstraction operations.

Definition 7.4 is a recursive definition. That is, we define the concept of \(\lambda\) expression in terms of itself. This might seem confusing at first, but in fact you have known recursive definitions since you were an elementary school student. Consider how we define an arithmetic expression: it is an expression that is either a number, or is built from other expressions \(\text{exp}, \text{exp}'\) using \((\text{exp} + \text{exp}')\), \((\text{exp} - \text{exp}')\), \((\text{exp} \times \text{exp}')\), or \((\text{exp} \div \text{exp}')\).

### Precedence and parenthesis.

We will use the following rules to allow us to drop some parenthesis. Function application associates from left to right, and so \(fgh\) is the same as \((fg)h\). Function application has a higher precedence than the \(\lambda\) operator, and so \(\lambda x. fgx\) is the same as \(\lambda x. ((fg)x)\). This is similar to how we use the precedence rules in arithmetic operations to allow us to use fewer parenthesis and so write the expression \((7 \times 3) + 2\) as \(7 \times 3 + 2\).
These two rules are commonly known as "\(\beta\) reduction" and "\(\alpha\) conversion" in the literature on the \(\lambda\) calculus. As mentioned in Remark 7.3, we also use the shorthand \(\lambda x. y.e\) for \(\lambda x.(\lambda y.e)\) and the shorthand \(f(x, y)\) for \((f \ x) \ y\). This plays nicely with the “Currying” transformation of simulating multi-input functions using \(\lambda\) expressions.

As we have seen in Eq. (7.7), the rule that \((\lambda x.exp)exp'\) is equivalent to \(exp[x \rightarrow exp']\) enables us to modify \(\lambda\) expressions and obtain simpler equivalent form for them. Another rule that we can use is that the parameter does not matter and hence for example \(\lambda y.y\) is the same as \(\lambda z.z\). Together these rules define the notion of equivalence of \(\lambda\) expressions:

**Definition 7.5 — Equivalence of \(\lambda\) expressions.** Two \(\lambda\) expressions are equivalent if they can be made into the same expression by repeated applications of the following rules: 6

1. Evaluation (aka \(\beta\) reduction): The expression \((\lambda x.exp)exp'\) is equivalent to \(exp[x \rightarrow exp']\).

2. Variable renaming (aka \(\alpha\) conversion): The expression \(\lambda x.exp\) is equivalent to \(\lambda y.exp[x \rightarrow y]\).

If \(exp\) is a \(\lambda\) expression of the form \(\lambda x.exp'\) then it naturally corresponds to the function that maps any input \(z\) to \(exp'[x \rightarrow z]\). Hence the \(\lambda\) calculus naturally implies a computational model. Since in the \(\lambda\) calculus the inputs can themselves be functions, we need to fix how we evaluate an expression such as

\[(\lambda x.f)(\lambda y.gz) . \quad (7.9)\]

There are two natural conventions for this:

- **Call by name:** We evaluate Eq. (7.9) by first plugging in the right-hand expression \((\lambda y.gz)\) as input to the lefthand side function, obtaining \(f[x \rightarrow (\lambda y.gz)]\) and then continue from there.

- **Call by value:** We evaluate Eq. (7.9) by first evaluating the righthand side and obtaining \(h = g[y \rightarrow z]\), and then plugging this into the lefthandside to obtain \(f[x \rightarrow h]\).

Because the \(\lambda\) calculus has only pure functions, that do not have “side effects”, in many cases the order does not matter. In fact, it can be shown that if we obtain an definite irreducible expression (for example, a number) in both strategies, then it will be the same one. However, there could be situations where “call by value” goes into an infinite loop while “call by name” does not. Hence we will use “call by name” henceforth. 7

---

6 These two rules are commonly known as “\(\beta\) reduction” and “\(\alpha\) conversion” in the literature on the \(\lambda\) calculus.

7
7.3.2 Functions as first class objects

The key property of the $\lambda$ calculus (and functional languages in general) is that functions are “first-class citizens” in the sense that they can be used as parameters and return values of other functions. Thus, we can invoke one $\lambda$ expression on another. For example if $\text{DOUBLE}$ is the $\lambda$ expression $\lambda f.(\lambda x.f(fx))$, then for every function $f$, $\text{DOUBLE } f$ corresponds to the function that invokes $f$ twice on $x$ (i.e., first computes $fx$ and then invokes $f$ on the result). In particular, if $f = \lambda y.(y + 1)$ then $\text{DOUBLE } f = \lambda x.(x + 2)$.

7.3.3 “Enhanced” lambda calculus

We now discuss the $\lambda$ calculus as a computational model. As we did with NAND++, we will start by describing an “enhanced” version of the $\lambda$ calculus that contains some “superfluous objects” but is easier to wrap your head around. We will later show how we can do without many of those concepts, and that the “enhanced $\lambda$ calculus” is equivalent to the “pure $\lambda$ calculus”.

The enhanced $\lambda$ calculus includes the following set of “basic” objects and operations:

- **Boolean constants**: 0 and 1. We also have the $\text{IF}$ function such that $\text{IF \text{cond} } a \ b$ outputs $a$ if $\text{cond} = 1$ and $b$ otherwise. (We use currying to implement multi-input functions, so $\text{IF \text{cond} } a \ b$ is the function $\lambda a.\lambda b.a$ if $\text{cond} = 1$ and is the function $\lambda a.\lambda b.b$ if $\text{cond} = 0$.) Using $\text{IF}$ and the constants 0, 1 we can also compute logical operations such as $\text{AND, OR, NOT, NAND}$ etc.: can you see why?

- **Strings/lists**: The function $\text{PAIR}$ where $\text{PAIR } x \ y$ that creates a pair from $x$ and $y$. We will also have the function $\text{HEAD}$ and $\text{TAIL}$ to extract the first and second member of the pair. We denote by $\text{NIL}$ the empty list, and so can create the list $x, y, z$ by $\text{PAIR } x \ (\text{PAIR } y \ (\text{PAIR } z \ \text{NIL}))$, see Fig. 7.6. The function $\text{ISEMPTY}$ will return 0 on any input that was generated by $\text{PAIR}$, but will return 1 on $\text{NIL}$. A string is of course simply a list of bits.\(^8\)

\(^8\)Note that if $L$ is a list, then $\text{HEADL}$ is its first element, but $\text{TAILL}$ is not the last element but rather all the elements except the first. Since $\text{NIL}$ denotes the empty list, $\text{PAIRsNIL}$ denotes the list with the single element $x$. 

\(\text{R} (\text{Lack of) types} \) Unlike most programming languages, the pure $\lambda$-calculus doesn’t have the notion of types. Every object in the $\lambda$ calculus can also be thought of as a $\lambda$ expression and hence as a function that takes one input and returns one output. All functions take one input and return one output, and if you feed a function an input of a form it didn’t expect, it still evaluates the $\lambda$ expression via “search and replace”, replacing all instances of its parameter with copies of the input expression you fed it.
• **List operations:** The functions MAP, REDUCE, FILTER. Given a list \( L = (x_0, \ldots, x_{n-1}) \) and a function \( f \), MAP \( f \) applies \( f \) on every member of the list to obtain \( L = (f(x_0), \ldots, f(x_{n-1})) \). The function \( FILTER \ L \ f \) returns the list of \( x_i \)'s such that \( f(x_i) = 1 \), and \( REDUCE \ L \ f \) “combines” the list by outputting
\[
f(x_0, f(x_1), \ldots, f(x_{n-3}, f(x_{n-2}, f(x_{n-1}, NIL)))) \quad (7.10)
\]
For example \( REDUCE \ L + \) would output the sum of all the elements of the list \( L \). See Fig. 7.7 for an illustration of these three operations.

• **Recursion:** Finally, we want to be able to execute **recursive functions**. Since in \( \lambda \) calculus functions are **anonymous**, we can’t write a definition of the form \( f(x) = \text{blah} \) where \( \text{blah} \) includes calls to \( f \). Instead we will construct functions that take an additional function \( \text{me} \) as a parameter. The operator \( \text{RECURSE} \) will take a function of the form \( \lambda \text{me}, \text{x.exp} \) as input and will return some function \( f \) that has the property that \( f = \lambda \text{x.exp}[\text{me} \rightarrow f] \).

**Example 7.6 — Computing XOR.** Let us see how we can compute the XOR of a list in the enhanced \( \lambda \) calculus. First, we note that we can compute XOR of two bits as follows:
\[
\text{NOT} = \lambda \text{a.IF}(a, 0, 1) \quad (7.11)
\]
and
\[
\text{XOR}_2 = \lambda a, b.IF(b, \text{NOT}(a), a) \quad (7.12)
\]
(We are using here a bit of syntactic sugar to describe the functions. To obtain the \( \lambda \) expression for XOR we will simply replace the expression Eq. (7.11) in Eq. (7.12).)

Now recursively we can define the XOR of a list as follows:
\[
\text{XOR}(L) = \begin{cases} 
0 & \text{L is empty} \\
\text{XOR}_2(\text{HEAD}(L), \text{XOR}(\text{TAIL}(L))) & \text{otherwise}
\end{cases} \quad (7.13)
\]
This means that \( \text{XOR} \) is equal to
\[
\text{RECURSE} (\lambda \text{me}, \text{L.IF}(\text{ISEMPTY}(L), 0, \text{XOR}_2(\text{HEAD} L, \text{me}(\text{TAIL} L)))) . \quad (7.14)
\]
That is, XOR is obtained by applying the RECURSE operator to the function that on inputs \( me, L \), returns 0 if \( ISEMP\)TY\( (L) \) and otherwise returns \( XOR_2 \) applied to \( HEAD(L) \) and to \( me(TAIL(L)) \).

**Solved Exercise 7.1 — Compute NAND using \( \lambda \) calculus.** Give a \( \lambda \) expression \( N \) such that \( N \ x \ y = NAND(x, y) \) for every \( x, y \in \{0, 1\} \).

**Solution:** This can be done in a similar way to how we computed \( XOR_2 \). The NAND of \( x, y \) is equal to 1 unless \( x = y = 1 \). Hence we can write

\[
N = \lambda x, y. IF(x, IF(y, 0, 1), 1)
\]  
(7.15)

![Figure 7.6](image1)

**Figure 7.6:** A list \((x_0, x_1, x_2)\) in the \( \lambda \) calculus is constructed from the tail up, building the pair \((x_2, NIL)\), then the pair \((x_1, (x_2, NIL))\) and finally the pair \((x_0, (x_1, (x_2, NIL)))\). That is, a list is a pair where the first element of the pair is the first element of the list and the second element is the rest of the list. The figure on the left renders this “pairs inside pairs” construction, though it is often easier to think of a list as a “chain”, as in the figure on the right, where the second element of each pair is thought of as a link, pointer or reference to the remainder of the list.

![Figure 7.7](image2)

**Figure 7.7:** Illustration of the \( \text{MAP}, \text{FILTER} \) and \( \text{REDUCE} \) operations.

An enhanced \( \lambda \) expression is obtained by composing the objects above with the application and abstraction rules. We can now define the notion of computing a function using the \( \lambda \) calculus. We will define the simplification of a \( \lambda \) expression as the following recursive process:
1. *(Evaluation / $\beta$ reduction.)* If the expression has the form $(exp_L \, exp_R)$ then replace the expression with $exp'_L[x \rightarrow exp_R]$.

2. *(Renaming / $\alpha$ conversion.)* When we cannot simplify any further, rename the variables so that the first bound variable in the expression is $v_0$, the second one is $v_1$, and so on and so forth.

Please make sure you understand why this recursive procedure simply corresponds to the “call by name” evaluation strategy.

The result of simplifying a $\lambda$ expression is an equivalent expression, and hence if two expressions have the same simplification then they are equivalent.

---

**Definition 7.7 — Computing a function via $\lambda$ calculus.** Let $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$ be a function and $exp$ a $\lambda$ expression. For every $x \in \{0, 1\}^*$, we denote by $LIST(x)$ the $\lambda$ list $PAIR(x_0, PAIR(x_1, PAIR(\ldots PAIR(x_{n-1}, NIL)))$ that corresponds to $x$.

We say that $exp$ computes $F$ if for every $x \in \{0, 1\}^*$, the expressions $(expLIST(x))$ and $LIST(F(x))$ are equivalent, and moreover they have the same simplification.

---

The basic operations of of the enhanced $\lambda$ calculus more or less amount to the Lisp/Scheme programming language. Given that, it is perhaps not surprising that the enhanced $\lambda$-calculus is equivalent to NAND++:

**Theorem 7.8 — Lambda calculus and NAND++.** For every function $F : \{0, 1\}^* \rightarrow \{0, 1\}^*$, $F$ is computable in the enhanced $\lambda$ calculus if and only if it is computable by a NAND++ program.

**Proof Idea:** To prove the theorem, we need to show that (1) if $F$ is computable by a $\lambda$ calculus expression then it is computable by a NAND++ program, and (2) if $F$ is computable by a NAND++ program, then it is computable by an enhanced $\lambda$ calculus expression.

Showing (1) is fairly straightforward. Applying the simplification rules to a $\lambda$ expression basically amounts to “search and replace” which we can implement easily in, say, NAND«, or for that matter Python (both of which are equivalent to NAND++ in power). Showing (2) essentially amounts to writing a NAND++ interpreter in a functional programming language such as LISP or Scheme. Showing how this can be done is a good exercise in mastering some functional programming techniques that are useful in their own right.

---

*In Lisp, the $PAIR$, $HEAD$ and $TAIL$ functions are traditionally called $\text{cons}$, $\text{car}$ and $\text{cdr}$.
Proof of Theorem 7.8. We only sketch the proof. The “if” direction is simple. As mentioned above, evaluating \( \lambda \) expressions basically amounts to “search and replace”. It is also a fairly straightforward programming exercise to implement all the above basic operations in an imperative language such as Python or C, and using the same ideas we can do so in NAND\( ^n \) as well, which we can then transform to a NAND++ program.

For the “only if” direction, we need to simulate a NAND++ program using a \( \lambda \) expression. First, by Solved Exercise 7.1 we can compute the NAND function, and hence every finite function, using the \( \lambda \) calculus. Thus the main task boils down to simulating the \textit{arrays} of NAND++ using the \textit{lists} of the enhanced \( \lambda \) calculus.

We will encode each array \( A \) of NAND++ program by a list \( L \) of the NAND program. For the index variable \( i \), we will have a special list \( I \) that has 1 only in the location corresponding to the value of \( i \). To simulate moving \( i \) to the left, we need to remove the first item from the list, while to simulate moving \( i \) to the right, we add a zero to the head of list.

To extract the \( i \)-th bit of the array corresponding to \( L \), we need to compute the following function \( \text{get} \) that on input a pair of lists \( I \) and \( L \) of bits of the same length \( n \), \( \text{get}(I, L) \) outputs 1 if and only if there is some \( j \in [n] \) such that the \( j \)-th element of both \( I \) and \( L \) is equal to 1. This turns out to be not so hard. The key is to implement the function \( \text{zip} \) that on input a pair of lists \( I \) and \( L \) of the same length \( n \), outputs a list of \( n \) pairs \( M \) such that the \( j \)-th element of \( M \) (which we denote by \( M_j \)) is the pair \((I_j, L_j)\). Thus \( \text{zip} \) “zips together” these two lists of elements into a single list of pairs. It is a good exercise to give a recursive implementation of \( \text{zip} \), and so can implement it using the \textit{RECURSE} operator. Once we have \( \text{zip} \), we can implement \( \text{get} \) by applying an appropriate \textit{REDUCE} on the list \( \text{zip}(I, L) \).

Setting the list \( L \) at the \( i \)-th location to a certain value requires computing the function \( \text{set}(I, L, v) \) that outputs a list \( L' \) such that \( L'_j = L_j \) if \( I_j = 0 \) and \( L'_j = v \) otherwise. The function \( \text{set} \) can be implemented by applying \textit{MAP} with an appropriate operator to the list \( \text{zip}(I, L) \).

We omit the full details of implementing \( \text{set}, \text{get} \), but the bottom line is that for every NAND++ program \( P \), we can obtain a \( \lambda \) expression \( \text{NEXT}_P \) such that, if we let \( \sigma = (\text{loop, foo, bar, \ldots, I, X, X\text{valid}, Y, Y\text{valid}, Baz, Blah, \ldots}) \) be the set of Boolean values and lists that encode the current state of \( P \) (with a list for each array and for the index variable \( i \)), then \( \text{NEXT}_P\sigma \) will encode the state after performing one iteration of \( P \).

Now we can use the following “pseudocode” to simulate the program \( P \). The function \( \text{SIM}_P \) will obtain an encoding \( \sigma_0 \) of the initial
state of \( P \), and output the encoding \( \sigma^+ \) of the state of \( P \) after it halts. It will be computed as follows:

**Algorithm** \( SIM_P(\sigma) \):

1. Let \( \sigma' = NEXT_P\sigma \).
2. If \( loop(\sigma') = 0 \) then return \( \sigma' \).
3. Otherwise return \( SIM_P(\sigma') \).

where \( loop(\sigma') \) simply denotes extracting the contents of the variable \( loop \) from the tuple \( \sigma \). We can write it as the \( \lambda \) expression

\[
R E C U R S E \ (\lambda m, \sigma . I F (\text{loop}(N E X T_P\sigma) , m(N E X T_P\sigma), N E X T_P\sigma))
\]

(7.16)

Given \( SIM_P \), we can compute the function computed by \( P \) by writing expressions for encoding the input as the initial state, and decoding the output from the final state. We omit the details, though this is fairly straightforward.\(^{11}\)

**7.3.4 How basic is “basic”?**

While the collection of “basic” functions we allowed for \( \lambda \) calculus is smaller than what’s provided by most Lisp dialects, coming from NAND++ it still seems a little “bloated”. Can we make do with less? In other words, can we find a subset of these basic operations that can implement the rest?

This is a good point to pause and think how you would implement these operations yourself. For example, start by thinking how you could implement \( M A P \) using \( R E D U C E \), and then \( R E D U C E \) using \( R E C U R S E \) combined with \( 0, 1, I F, \text{PAIR, HEAD, TAIL, NIL, ISEMPTY} \) together with the \( \lambda \) operations. Now you can think how you could implement \( \text{PAIR, HEAD} \) and \( \text{T A I L} \) based on \( 0, 1, I F \). The idea is that we can represent a pair as function.

It turns out that there is in fact a proper subset of these basic operations that can be used to implement the rest. That subset is the empty set. That is, we can implement all the operations above using the \( \lambda \) formalism only, even without using 0’s and 1’s. It’s \( \lambda \)'s all the way down! The idea is that we encode 0 and 1 themselves as \( \lambda \) expressions, and build things up from there. This notion is known as Church encoding, as it was originated by Church in his effort to show that the \( \lambda \) calculus

\(^{11}\) For example, if \( X \) is a list representing the input, then we can obtain a list \( X v a l i d \) of 1’s of the same length by simply writing \( X v a l i d = M A P(X, \lambda x . 1) \).
Theorem 7.9 — Enhanced λ calculus equivalent to pure λ calculus.

There are λ expressions that implement the functions 0, 1, IF, PAIR, HEAD, TAIL, NIL, ISEMPTY, MAP, REDUCE, and RECURSE.

We will not write the full formal proof of Theorem 7.9 but outline the ideas involved in it:

- We define 0 to be the function that on two inputs \( x, y \) outputs \( y \), and 1 to be the function that on two inputs \( x, y \) outputs \( x \). Of course we use Currying to achieve the effect of two inputs and hence \( 0 = \lambda x.\lambda y.y \) and \( 1 = \lambda x.\lambda y.x \).\(^{12}\)

- The above implementation makes the IF function trivial: IF(\( \text{cond}, a, b \)) is simply \( \text{cond} \ a \ b \) since \( 0ab = b \) and \( 1ab = a \). We can write IF = \( \lambda x.x \) to achieve IF(\( \text{cond}, a, b \)) = ((IF\( \text{cond} \))a)b = \text{cond} \ a \ b.

- To encode a pair \( (x, y) \) we will produce a function \( f_{x,y} \) that has \( x \) and \( y \) “in its belly” and such that \( f_{x,y}g = gxy \) for every function \( g \). That is, we write PAIR = \( \lambda x, y.\lambda g.gxy \). Note that now we can extract the first element of a pair \( p \) by writing \( p1 \) and the second element by writing \( p0 \), and so \( \text{HEAD} = \lambda p.p1 \) and \( \text{TAIL} = \lambda p.p0 \).

- We define NIL to be the function that ignores its input and always outputs 1. That is, NIL = \( \lambda x.1 \). The ISEMPTY function checks, given an input \( p \), whether we get 1 if we apply \( p \) to the function \( z = \lambda x, y.0 \) that ignores both its inputs and always outputs 0. For every valid pair of the form \( p = \text{PAIR}xy, p_z = pxy = 0 \) while NILz = 1. Formally, ISEMPTY = \( \lambda p.p(\lambda x, y.0) \).

\( \text{R} \) Church numerals (optional) There is nothing special about Boolean values. You can use similar tricks to implement natural numbers using λ terms. The standard way to do so is to represent the number \( n \) by the function \( \text{ITER}_n \) that on input a function \( f \) outputs the function \( x \mapsto f(f(...f(x))) \) (\( n \) times). That is, we represent the natural number 1 as \( \lambda f.f \), the number 2 as \( \lambda f. (\lambda x.f(fx)) \), the number 3 as \( \lambda f. (\lambda x.f(f(fx))) \), and so on and so forth. (Note that this is not the same representation we used for 1 in the Boolean context: this is fine; we already know that the same object can be represented in more than one way.) The number 0 is represented by the function that maps any function \( f \) to the identity function \( \lambda x.x \). (That is, \( 0 = \lambda f. (\lambda x.x) \).

In this representation, we can compute \( \text{PLUS}(n, m) \)
as \( \lambda f. \lambda x. (mf)((mf)x) \) and \( TIMES(n, m) \) as \( \lambda f. n(mf) \). Subtraction and division are trickier, but can be achieved using recursion. (Working this out is a great exercise.)

### 7.3.5 List processing

Now we come to the big hurdle, which is how to implement \( MAP \), \( FILTER \), and \( REDUCE \) in the \( \lambda \) calculus. It turns out that we can build \( MAP \) and \( FILTER \) from \( REDUCE \). For example \( MAP(L, f) \) is the same as \( REDUCE(L, g) \) where \( g \) is the operation that on input \( x \) and \( y \), outputs \( PAIR(f(x), NIL) \) if \( y \) is NIL and otherwise outputs \( PAIR(f(x), y) \). (I leave checking this as a (recommended!) exercise for you, the reader.) So, it all boils down to implementing \( REDUCE \). We can define \( REDUCE(L, g) \) recursively, by setting \( REDUCE(NIL, g) = NIL \) and stipulating that given a non-empty list \( L \), which we can think of as a pair \( (head, rest) \), \( REDUCE(L, g) = g(head, REDUCE(rest, g)) \). Thus, we might try to write a \( \lambda \) expression for \( REDUCE \) as follows

\[
REDUCE = \lambda L, g. IF(ISEMPYTY(L), NIL, gHEAD(L)REDUCE(TAIL(L), g)) .
\]

(7.17)

The only fly in this ointment is that the \( \lambda \) calculus does not have the notion of recursion, and so this is an invalid definition. But of course we can use our \( RECURSE \) operator to solve this problem. We will replace the recursive call to "\( REDUCE \)" with a call to a function \( me \) that is given as an extra argument, and then apply \( RECURSE \) to this. Thus \( REDUCE = RECURSE myREDUCE \) where

\[
myREDUCE = \lambda me, L, g. IF(ISEMPYTY(L), NIL, gHEAD(L)me(TAIL(L), g)) .
\]

(7.18)

So everything boils down to implementing the \( RECURSE \) operator, which we now deal with.

### 7.3.6 Recursion without recursion

How can we implement recursion without recursion? We will illustrate this using a simple example - the \( XOR \) function. As shown in Example 7.6, we can write the \( XOR \) function of a list recursively as follows:

\[
XOR(L) = \begin{cases} 
0 & \text{if } L \text{ is empty} \\
XOR_2(HEAD(L), XOR(TAIL(L))) & \text{otherwise} 
\end{cases}
\]

(7.19)
where \(\text{XOR}_2 : \{0,1\}^2 \to \{0,1\}\) is the XOR on two bits. In Python we would write this as

```python
def xor2(a,b): return 1-b if a else b
def head(L): return L[0]
def tail(L): return L[1:]

def xor(L): return xor2(head(L),xor(tail(L))) if L else 0
```

```python
print(xor([0,1,0,1,0,1]))
# 1
```

Now, how could we eliminate this recursive call? The main idea is that since functions can take other functions as input, it is perfectly legal in Python (and the \(\lambda\) calculus of course) to give a function itself as input. So, our idea is to try to come up with a non recursive function \(\text{tempxor}\) that takes two inputs: a function and a list, and such that \(\text{tempxor}(\text{tempxor}, L)\) will output the XOR of \(L\)!

At this point you might want to stop and try to implement this on your own in Python or any other programming language of your choice (as long as it allows functions as inputs).

Our first attempt might be to simply use the idea of replacing the recursive call by \(\text{me}\). Let’s define this function as \(\text{myxor}\)

```python
def myxor(me,L): return xor2(head(L),me(tail(L))) if L else 0
```

```python
myxor(myxor,[1,0,1])
# TypeError: myxor() missing 1 required positional argument
```

Let’s test this out:

```python
myxor(myxor,[1,0,1])
# TypeError: myxor() missing 1 required positional argument
```

The problem is that \(\text{myxor}\) expects two inputs- a function and a list- while in the call to \(\text{me}\) we only provided a list. To correct this, we modify the call to also provide the function itself:

```python
def tempxor(me,L): return xor2(head(L),me(me,tail(L))) if L else 0
```

```python
tempxor(tempxor,[1,0,1])
# 0
tempxor(tempxor,[1,0,1,1])
# 1
```
We see this now works! Note the call `me(me, ..)` in the definition of `tempxor`: given a function `me` as input, `tempxor` will actually call the function on itself! Thus we can now define `xor(L)` as simply

```python
return tempxor(tempxor,L).
```

The approach above is not specific to XOR. Given a recursive function `f` that takes an input `x`, we can obtain a non-recursive version as follows:

1. Create the function `myf` that takes a pair of inputs `me` and `x`, and replaces recursive calls to `f` with calls to `me`.

2. Create the function `tempf` that converts calls in `myf` of the form `me(x)` to calls of the form `me(me,x)`.

3. The function `f(x)` will be defined as `tempf(tempf,x)`

Here is the way we implement the `RECURSE` operator in Python. It will take a function `myf` as above, and replace it with a function `g` such that `g(x)=myf(g,x)` for every `x`.

```python
def RECURSE(myf):
    def tempf(me,x):
        return myf(lambda x: me(me,x),x)
    return lambda x: tempf(tempf,x)

xor = RECURSE(myxor)

print(xor([0,1,1,0,0,1]))  # 1

print(xor([1,1,0,0,1,1,1,1]))  # 0
```

From Python to the λ calculus. In the λ calculus, a two-input function `g` that takes a pair of inputs `me,x` is written as `λme.(λx.g)`. So the function `x → me(me,x)` is simply written as `me me`. (Can you see why?) So in the λ calculus, the function `tempf` will be `f (me me)` and the function `λ x. tempf(tempf,x)` is the same as `tempf tempf`. So the `RECURSE` operator in the λ calculus is simply the following:

\[
RECURSE = λf.(λm.(f(m m)) (λm.(f(m m))))
\]  

(7.20)

The online appendix contains an implementation of the λ calculus using Python. Here is an implementation of the recursive XOR function from that appendix.\(^{13}\)

\(^{13}\) Because of specific issues of Python syntax, in this implementation we use `f * g` for applying `f` to `g` rather than `fg`, and use `λx.exp` rather than `λx.exp` for abstraction. We also use `_0` and `_1` for the λ terms for 0 and 1 so as not to confuse with the Python constants.
# XOR of two bits
XOR2 = \lambda (a,b) (\text{IF}(a,\text{IF}(b,\_0,\_1),b))

# Recursive XOR with recursive calls replaced by m parameter
myXOR = \lambda (m,l) (\text{IF}(\text{ISEMPTY}(l),\_0,\text{XOR2}(\text{HEAD}(l),m(\text{TAIL}(l)))))

# Recurse operator (aka Y combinator)
RECURSE = \lambda f ((\lambda m(f(m\\ * m)))(\lambda m(f(m\\ * m))))

# XOR function
XOR = RECURSE(myXOR)

# TESTING:
XOR(PAIR(\_1,NIL)) # List [1] # equals 1
XOR(PAIR(\_1,PAIR(\_0,PAIR(\_1,NIL)))) # List [1,0,1] # equals 0

The Y combinator The RECURSE operator above is better known as the Y combinator. It is one of a family of fixed point operators that given a lambda expression F, find a fixed point f of F such that f = Ff. If you think about it, XOR is the fixed point of myXOR above. XOR is the function such that for every x, if plug in XOR as the first argument of myXOR then we get back XOR, or in other words XOR = myXOR XOR. Hence finding a fixed point for myXOR is the same as applying RECURSE to it.

Infinite loops in the \lambda calculus The fact that \lambda-expressions can simulate NAND++ programs means that, like them, it can also enter into an infinite loop. For example, consider the \lambda expression

(\lambda x.xxx)(\lambda x.xxx) \tag{7.21}

If we try to evaluate it then the first step is to invoke the lefthand function on the righthand one and then obtain

(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \tag{7.22}
To evaluate this, the next step would be to apply the second term on the third term,\(^{14}\) which would result in

\[(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)\]  

(7.23)

We can see that continuing in this way we get longer and longer expressions, and this process never concludes.

### 7.4 OTHER MODELS

There is a great variety of models that are computationally equivalent to Turing machines (and hence to NAND++/NAND« program). Chapter 8 of the book *The Nature of Computation* is a wonderful resource for some of those models. We briefly mention a few examples.

#### 7.4.1 Parallel algorithms and cloud computing

The models of computation we considered so far are inherently sequential, but these days much computation happens in parallel, whether using multi-core processors or in massively parallel distributed computation in data centers or over the Internet. Parallel computing is important in practice, but it does not really make much difference for the question of what can and can’t be computed. After all, if a computation can be performed using \(m\) machines in \(t\) time, then it can be computed by a single machine in time \(mt\).

#### 7.4.2 Game of life, tiling and cellular automata

Many physical systems can be described as consisting of a large number of elementary components that interact with one another. One way to model such systems is using *cellular automata*. This is a system that consists of a large number (or even infinite) cells. Each cell only has a constant number of possible states. At each time step, a cell updates to a new state by applying some simple rule to the state of itself and its neighbors.

A canonical example of a cellular automaton is *Conway’s Game of Life*. In this automata the cells are arranged in an infinite two dimensional grid. Each cell has only two states: “dead” (which we can encode as 0 and identify with ∅) or “alive” (which we can encode as 1). The next state of a cell depends on its previous state and the states of its 8 vertical, horizontal and diagonal neighbors. A dead cell becomes alive only if exactly three of its neighbors are alive. A live cell continues to live if it has two or three live neighbors. Even though the number of cells is potentially infinite, we can have a finite encoding for the state by only keeping track of the live cells. If we initialize the

---

\(^{14}\) This assumes we use the “call by value” evaluation ordering which states that to evaluate a \(\lambda\) expression \(\frac{\lambda}{g}\) we first evaluate the righthand expression \(\frac{\lambda}{x}\) and then invoke \(\frac{\lambda}{x}\) on it. The “Call by name” or “lazy evaluation” ordering would first evaluate the lefthand expression \(\frac{\lambda}{x}\) and then invoke it on \(g\). In this case both strategies would result in an infinite loop. There are examples though when “call by name” would not enter an infinite loop while “call by value” would. The SML and OCaml programming languages use “call by value” while Haskell uses (a close variant of) “call by name”.
system in a configuration with a finite number of live cells, then the number of live cells will stay finite in all future steps.

We can think of such a system as encoding a computation by starting it in some initial configuration, and then defining some halting condition (e.g., we halt if the cell at position \((0, 0)\) becomes dead) and some way to define an output (e.g., we output the state of the cell at position \((1, 1)\)). Clearly, given any starting configuration \(x\), we can simulate the game of life starting from \(x\) using a NAND\(^{-}\) (or NAND\(++\)) program, and hence every “Game-of-Life computable” function is computable by a NAND\(^{-}\) program. Surprisingly, it turns out that the other direction is true as well: as simple as its rules seem, we can simulate a NAND\(++\) program using the game of life (see Fig. 7.8). The Wikipedia page for the Game of Life contains some beautiful figures and animations of configurations that produce some very interesting evolutions. See also the book The Nature of Computation.

![Game-of-Life configuration simulating a Turing Machine](image)

**Figure 7.8.** A Game-of-Life configuration simulating a Turing Machine. Figure by Paul Rendell.

### 7.4.3 Configurations of NAND\(++\)/Turing machines and one dimensional cellular automata

It turns out that even one dimensional cellular automata can be Turing complete (see Fig. 7.11). In a one dimensional automata, the cells are laid out in one infinitely long line. The next state of each cell is only a function of its past state and the state of both its neighbors.
Definition 7.10 — One dimensional cellular automata. Let $\Sigma$ be a finite set containing the symbol $\emptyset$. A one dimensional cellular automation over alphabet $\Sigma$ is described by a transition rule $r : \Sigma^3 \to \Sigma$, which satisfies $r(\emptyset, \emptyset, \emptyset) = \emptyset$.

An configuration of the automaton is specified by a string $\alpha \in \Sigma^*$. We can also think of $\alpha$ as the infinite sequence $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1}, \emptyset, \emptyset, \emptyset, \ldots)$, where $n = |\alpha|$. If $\alpha$ is a configuration and $r$ is a transition rule, then the next step configuration, denoted by $\alpha' = \text{NEXT}_r(\alpha)$ is defined as follows:

$$alpha'_i = \text{NEXT}_r(\alpha_{i-1}, \alpha_i, \alpha_{i+1})$$

(7.24)

for $i = 0, \ldots, n$. If $j$ is smaller than 0 or larger than $n - 1$ then we set $\alpha_j = \emptyset$.

In other words, the next state of the automaton $r$ at point $i$ obtained by applying the rule $r$ to the values of $\alpha$ at $i$ and its two neighbors.

Theorem 7.11 — One dimensional automata are Turing complete. For every NAND++ program $P$, there is a one dimension cellular automaton that can simulate $P$ on every input $x$.

Specifically, for every NAND++ program $P$, there is a finite alphabet $\Sigma$ and an automaton $\mathcal{A}$ over this alphabet, as well as an efficient mapping from the inputs to $P$ to starting configurations for $\mathcal{A}$ and from configurations of $\mathcal{A}$ whose first coordinate has a special form into outputs of $P$. Namely, there is a computable map $\text{ENCODE} : \{0, 1\}^* \to \Sigma^*$ and two special symbols $\sigma_0, \sigma_1 \in \Sigma$, such that for every $x \in \{0, 1\}^*$, $P(x)$ halts with input $b \in \{0, 1\}$ if and only if the automaton $\mathcal{A}$ initialized with configuration $\text{ENCODE}(x)$ eventually reaches a configuration with $\beta_0 = \sigma_b$.

Proof Idea: A configuration of $P$ contains its full state at after a particular iteration. That is, the contents of all the array and scalar variables, as well as the value of the index variable $i$. We can encode such a configuration of $P$ as a string $\alpha$ over an alphabet $\Sigma$ of $2^a + 2^{a+b}$ symbols (where $a$ is the number of array variables in $P$ and $b$ is the number of scalar variables). The idea is that in all locations $j$ except that corresponding to the current value of $i$, we will encode at $\alpha_j$ the values of
the array variables at location \( j \). In the location corresponding to \( i \) we will also include in the encoding the values of all the scalar variables.

Given this notion of an encoding, and the fact that \( i \) moves only one step in each iteration, we can see that after one iteration of the program \( P \), the configuration largely stays the same except the locations \( i, i-1, i+1 \) corresponding to the location of the current variable \( i \) and its immediate neighbors. Once we realize this, we can phrase the progression from one configuration to the next as a one dimensional cellular automaton! From this observation, **Theorem 7.11** follows in a fairly straightforward manner.

Before proving **Theorem 7.11**, let us formally define the notion of a **configuration** of a NAND++ program (see also Fig. 7.9). We will come back to this notion in later chapters as well. We restrict attention to so called **well formed** NAND++ programs (see **Definition 6.8**), that have a clean separation of array and scalar variables. Of course, this is not really a restriction since every NAND++ program \( P \) can be transformed into an equivalent one that is well formed (see **Lemma 6.9**).

**Figure 7.9**: A configuration of a (well formed) NAND++ program \( P \) with \( a \) array variables and \( b \) scalar variables is a list \( \alpha \) of strings in \( \{0,1\}^a \cup \{0,1\}^{a+b} \). In exactly one index \( \alpha_j \in \{0,1\}^{a+b} \). This corresponds to the index variable \( i = j \), and \( \alpha_j \) encodes both the contents of the scalar variables, as well as the array variables at the location \( i \). For \( j \neq i \), \( \alpha_j \) encodes the contents of the array variables at the location \( j \). The length of the list \( \alpha \) denotes the largest index that has been reached so far in the execution of the program. If in one iteration we move from \( \alpha \) to \( \alpha' \), then for every \( j \), \( \alpha'_j \) is a function of \( \alpha_{j-1}, \alpha_j, \alpha_{j+1} \).

**Definition 7.12** has many technical details, but is not actually deep and complicated. You would probably understand it better if before starting to read it, you take a moment to stop and think how **you** would encode as a string the state of a NAND++ program at a given point in an execution.

Think what are all the components that you need to know in order to be able to continue the execution from this point onwards, and what is a simple way to encode them using a list of strings (which in turn can be encoded as a string). In particular, with an eye towards our future applications, try to think of an encoding which will make it as simple as possible to map a configuration at step \( t \) to the configuration at step \( t+1 \).
**Definition 7.12 — Configuration of NAND++ programs.** Let $P$ be a well-formed NAND++ program (as per Definition 6.8) with $a$ array variables and $b$ scalar variables. A configuration of $P$ is a list of strings $\alpha = (\alpha_0, \ldots, \alpha_{t-1})$ such that for every $j \in [t]$, $\alpha_j$ is either in $\{0, 1\}^a$ or in $\{0, 1\}^{a+b}$. Moreover, there is exactly a single coordinate $i \in [t]$, such that $\alpha_i \in \{0, 1\}^{a+b}$ and for all other coordinates $j \neq i$, $\alpha_j \in \{0, 1\}^a$.

A configuration $\alpha$ corresponds to the state of $P$ at the beginning of some iteration as follows:

- The value of the index variable $i$ is the index $i$ such that $\alpha_i \in \{0, 1\}^{a+b}$. The value of the $b$ scalar variables is encoded by the last $b$ bits of $\alpha_j$, while the value of the $a$ array variables at the location $i$ is encoded by the first $a$ bits of $\alpha_i$.

- For every $j \neq i$, the value of the $a$ array variables at the location $j$ is encoded by $\alpha_j$.

- The length of $\alpha$ corresponds to the largest position $i$ that the program have reached up until this point in the execution. (And so in particular by our convention, the value of all array vari-
Since $\alpha$ is well-formed, we assume it contains an index increasing variable that can be used to compute whether $i$ increases or decreases at the end of an iteration.

**Definition 7.12** is a little cumbersome, but ultimately a configuration is simply a string that encodes a snapshot of the state of the NAND++ program at a given point in the execution. (In operating-systems lingo, it would be a “core dump”.) Such a snapshot needs to encode the following components:

1. The current value of the index variable $i$.
2. For every scalar variable `foo`, the value of `foo`.
3. For every array variable `Bar`, the value `Bar[i]` for every $i \in \{0, \ldots, m-1\}$ where $m-1$ is the largest value that the index variable $i$ ever achieved in the computation.

The function $NEXT_p$ takes a string $\sigma \in \{0,1\}^*$ that encodes the configuration after $t$ iterations, and maps it to the string $\sigma'$ that encodes the configuration after $t - 1$. The specific details of how we represent configurations and how $NEXT_p$ are not so important as much as the following points:
• $\sigma$ and $\sigma'$ agree with each other in all but a constant number of coordinates.

• Every bit of $\sigma'$ is a function of a constant number of bits of $\sigma$.

Specifically, For every NAND++ program $P$ there is a constant $C > 0$ and a finite function $MAP_P : \{0, 1\}^{2C} \to \{0, 1, \perp\}$ such that for every $i \in \mathbb{N}$ and string $\sigma$ that encodes a valid configuration of $P$, the $i$-the bit of $NEXT_P(\sigma)$ is obtained by applying the finite function $MAP_P$ to the $2C$ bits of $\sigma$ corresponding to coordinates $i-C, i-C+1, \ldots, i+C$.\footnote{If one of those is “out of bounds”-corresponds to $\sigma_j$ for $j < 0$ or $j \geq |\sigma|$ - then we replace it with 0. If $i \geq |NEXT_P(\sigma)|$ then we think of the $i$-th bit of $NEXT_P(\sigma)$ as equaling $\perp$.}

**Proof of Theorem 7.11.** Assume without loss of generality that $P$ is a well-formed NAND++ program with $a$ array variables and $b$ scalar variables. (Otherwise we can translate it to such a program.) Let $\Sigma = \{0, 1\}^{a+b+3}$ (a space which, as we saw, is large enough to encode every coordinate of a configuration), and hence think of a configuration as a string in $\sigma \in \Sigma^*$ such that the $i$-th coordinate in $\sigma' = NEXT_P(\sigma)$ only depends on the $i-1$-th, $i$-th, and $i+1$-th coordinate of $\sigma$. Thus $NEXT_P$ (the function of Definition 7.12 that maps a configuration of $P$ into the next one) is in fact a valid rule for a one dimensional automata. The only thing we have to do is to identify the default value of $\emptyset$ with the value 0 (which corresponds to the index not being in this location and all array variables are set to 0).

For every input $x$, we can compute $\alpha(x)$ to be the configuration corresponding to the initial state of $P$ when executed on input $x$. We can modify the program $P$ so that when it decides to halt, it will first wait until the index variable $i$ reaches the 0 position and also zero out all of its scalar and array variables except for $Y$ and $Yvalid$. Hence the program eventually halts if and only the automaton eventually reaches a configuration $\beta$ in which $\beta_0$ encodes the value of $\text{loop}$ as 0, and moreover in this case, we can “read off” the output from $\beta_0$. $\blacksquare$
The automaton arising from the proof of Theorem 7.11 has a large alphabet, and furthermore one whose size that depends on the program $P$ that is being simulated. It turns out that one can obtain an automaton with an alphabet of fixed size that is independent of the program being simulated, and in fact the alphabet of the automaton can be the minimal set $\{0, 1\}$! See Fig. 7.11 for an example of such an Turing-complete automaton.

Figure 7.11: Evolution of a one dimensional automata. Each row in the figure corresponds to the configuration. The initial configuration corresponds to the top row and contains only a single “live” cell. This figure corresponds to the “Rule 110” automaton of Stefan Wolfram which is Turing Complete. Figure taken from Wolfram MathWorld.

7.5 TURING COMPLETENESS AND EQUIVALENCE, A FORMAL DEFINITION (OPTIONAL)

A computational model is some way to define what it means for a program (which is represented by a string) to compute a (partial) function. A computational model $M$ is Turing complete, if we can map every Turing machine (or equivalently NAND++ program) $Q$ into a program $P$ for $M$ that computes the same function as $Q$. It is Turing equivalent if the other direction holds as well (i.e., we can map every program in $M$ to a Turing machine/NAND++ program that computes the same function). Formally, we can define this notion as follows:
264 Introduction to Theoretical Computer Science

We could have equally well made this definition using Turing machines, \( \lambda \) calculus, and many other models.

Definition 7.13 — Turing completeness and equivalence. Let \( \mathcal{F} \) be the set of all partial functions from \( \{0, 1\}^* \) to \( \{0, 1\}^* \). A computational model is a map \( M : \{0, 1\}^* \rightarrow \mathcal{F} \). We say that a program \( P \) in the model \( M \) computes a function \( F \in \mathcal{F} \) if \( M(P) = F \).

A computational model \( M \) is Turing complete if there is a computable map \( ENCODE_M : \{0, 1\}^* \rightarrow \{0, 1\}^* \) for every NAND++ program \( P \) (represented as a string), \( M(ENCODE_M(P)) \) is equal to the partial function computed by \( P \). A computational model \( M \) is Turing equivalent if it is Turing complete and there exists a computable map \( DECODE_M : \{0, 1\}^* \rightarrow \{0, 1\}^* \) such that for every string \( Q \in \{0, 1\}^* \), \( P = ENCODE_M(Q) \) is a string representation of a NAND++ program that computes the function \( M(Q) \).

Some examples of Turing Equivalent models include:

- Turing machines
- NAND++ programs
- NAND« programs
- \( \lambda \) calculus
- Game of life (mapping programs and inputs/outputs to starting and ending configurations)
- Programming languages such as Python/C/Javascript/OCaml… (allowing for unbounded storage)

7.6 THE CHURCH-TURING THESIS (DISCUSSION)

“[In 1934], Church had been speculating, and finally definitely proposed, that the \( \lambda \)-definable functions are all the effectively calculable functions … When Church proposed this thesis, I sat down to disprove it … but, quickly realizing that [my approach failed], I became overnight a supporter of the thesis.\)” — Stephen Kleene, 1979.

“[The thesis is] not so much a definition or to an axiom but … a natural law.” — Emil Post, 1936.

We have defined functions to be computable if they can be computed by a NAND++ program, and we’ve seen that the definition would remain the same if we replaced NAND++ programs by Python programs, Turing machines, \( \lambda \) calculus, cellular automata, and many other computational models. The Church-Turing thesis is that this is
The only sensible definition of “computable” functions. Unlike the “Physical Extended Church Turing Thesis” (PECTT) which we saw before, the Church Turing thesis does not make a concrete physical prediction that can be experimentally tested, but it certainly motivates predictions such as the PECTT. One can think of the Church-Turing Thesis as either advocating a definitional choice, making some prediction about all potential computing devices, or suggesting some laws of nature that constrain the natural world. In Scott Aaronson’s words, “whatever it is, the Church-Turing thesis can only be regarded as extremely successful”. No candidate computing device (including quantum computers, and also much less reasonable models such as the hypothetical “closed time curve” computers we mentioned before) has so far mounted a serious challenge to the Church Turing thesis. These devices might potentially make some computations more efficient, but they do not change the difference between what is finitely computable and what is not.18

7.7 OUR MODELS VS OTHER TEXTS

We can summarize the models we use versus those used in other texts in the following table:

<table>
<thead>
<tr>
<th>Model</th>
<th>These notes</th>
<th>Other texts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonuniform</td>
<td>NAND programs</td>
<td>Boolean circuits, straightline programs</td>
</tr>
<tr>
<td>Uniform (random access)</td>
<td>NAND« programs</td>
<td>RAM machines</td>
</tr>
<tr>
<td>Uniform (sequential access)</td>
<td>NAND++ programs</td>
<td>Oblivious one-tape Turing machines</td>
</tr>
</tbody>
</table>

Later on in this course we may study memory bounded computation. It turns out that NAND++ programs with a constant amount of memory are equivalent to the model of finite automata (the adjectives “deterministic” or “nondeterministic” are sometimes added as well, this model is also known as finite state machines) which in turns captures the notion of regular languages (those that can be described by regular expressions).

Lecture Recap

- While we defined computable functions using NAND++ programs, we could just as well have done so using many other models, including not just NAND« but also Turing machines, RAM machines, the λ-calculus and many other models.
- Very simple models turn out to be “Turing complete” in the sense that they can simulate arbitrarily complex computation.

18 The extended Church Turing thesis, which we’ll discuss later in this course, is that NAND++ programs even capture the limit of what can be efficiently computable. Just like the PECTT, quantum computing presents the main challenge to this thesis.


7.8 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 7.1 — Pairing. Let embed : \( \mathbb{N}^2 \to \mathbb{N} \) be the function defined as
\[
\text{embed}(x_0, x_1) = \frac{1}{2}(x_0 + x_1)(x_0 + x_1 + 1) + x_1.
\]

1. Prove that for every \( x^0, x^1 \in \mathbb{N} \), embed\((x^0, x^1)\) is indeed a natural number.
2. Prove that embed is one-to-one.
3. Construct a NAND++ program \( P \) such that for every \( x^0, x^1 \in \mathbb{N} \),
\[
P(pf(x^0)pf(x^1)) = pf(embed(x^0, x^1)),
\]
where \( pf \) is the prefix-free encoding map defined above. You can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.
4. Construct NAND++ programs \( P_0, P_1 \) such that for every \( x^0, x^1 \in \mathbb{N} \) and \( i \in \mathbb{N} \),
\[
P_i(pf(embed(x^0, x^1))) = pf(x^i),
\]
you can use the syntactic sugar for inner loops, conditionals, and incrementing/decrementing the counter.

Exercise 7.2 — lambda calculus requires three variables. Prove that for every \( \lambda \)-expression \( e \) with no free variables there is an equivalent \( \lambda \)-expression \( f \) using only the variables \( x, y, z \).\(^{20}\)

7.9 BIBLIOGRAPHICAL NOTES

7.10 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include:

- Tao has proposed showing the Turing completeness of fluid dynamics (a “water computer”) as a way of settling the question of the behavior of the Navier-Stokes equations, see this popular article.

\(^{19}\) TODO: Add an exercise showing that NAND++ programs where the integers are represented using the unary basis are equivalent up to polylog terms with multi-tape Turing machines.

\(^{20}\) Hint: You can reduce the number of variables a function takes by “pairing them up”. That is, define a \( \lambda \) expression \( \text{PAIR} \) such that for every \( x, y \)
\[
\text{PAIR}\( x, y \)
\]
is some function \( f \) such that \( f_0 = x \) and \( f_1 = y \). Then use \( \text{PAIR} \) to iteratively reduce the number of variables used.

\(^{21}\) TODO: Recommend Chapter 7 in the nature of computation.
7.11 ACKNOWLEDGEMENTS
8
Universality and uncomputability

“A function of a variable quantity is an analytic expression composed in any way whatsoever of the variable quantity and numbers or constant quantities.”, Leonhard Euler, 1748.

“The importance of the universal machine is clear. We do not need to have an infinity of different machines doing different jobs. … The engineering problem of producing various machines for various jobs is replaced by the office work of ‘programming’ the universal machine”, Alan Turing, 1948

One of the most significant results we showed for NAND programs is the notion of universality: that a NAND program can evaluate other NAND programs. However, there was a significant caveat in this notion. To evaluate a NAND program of \( s \) lines, we needed to use a bigger number of lines than \( s \). (Equivalently, the function that evaluates a given circuit of \( s \) gates on a given input, requires more than \( s \) gates to compute.)

It turns out that uniform models such as NAND++ programs or Turing machines allow us to “break out of this cycle” and obtain a truly universal NAND++ program \( U \) that can evaluate all other programs, including programs that have more lines than \( U \) itself. The existence of such a universal program has far reaching applications. Indeed, it is no exaggeration to say that the existence of a universal program underlies the information technology revolution that began in the latter half of the 20th century (and is still ongoing). Up to that point in history, people have produced various special-purpose calculating devices, from the abacus, to the slide ruler, to machines to compute trigonometric series. But as Turing (who was perhaps the one to see most clearly the ramifications of universality) observed, a

Learning Objectives:
- The universal machine/program - “one program to rule them all”
- See a fundamental result in computer science and mathematics: the existence of uncomputable functions.
- See the canonical example for an uncomputable function: the halting problem.
- Introduction to the technique of reductions which will be used time and again in this course to show difficulty of computational tasks.
- Rice’s Theorem, which is a starting point for much of research on compilers and programming languages, and marks the difference between semantic and syntactic properties of programs.
general purpose computer is much more powerful. That is, we only need to build a device that can compute the single function $U$, and we have the ability, via software to extend it to do arbitrary computations. If we want to compute a new NAND++ program $P$, we do not need to build a new machine, but rather can represent $P$ as a string (or code) and use it as input for the universal program $U$. Beyond the practical applications, the existence of a universal algorithm also surprising theoretical ramifications, and in particular can be used to show the existence of uncomputable functions, upending the intuitions of mathematicians over the centuries from Euler to Hilbert. In this chapter we will prove the existence of the universal program, as well as show its implications for uncomputability.

8.1 UNIVERSALITY: A NAND++ INTERPRETER IN NAND++

Like a NAND program, a NAND++ program (or a Python or Javascript program, for that matter) is ultimately a sequence of symbols and hence can obviously be represented as a binary string. We will spell out the exact details of one such representation later, but as usual, the details are not so important (e.g., we can use the ASCII encoding of the source code). What is crucial is that we can use such representations to evaluate any program. That is, we prove the following theorem:

**Theorem 8.1 — Universality of NAND++.** There is a NAND++ program $U$ that computes the partial function $EVAL : \{0,1\}^* \rightarrow \{0,1\}^*$ defined as follows:

$$EVAL(P, x) = P(x)$$

(8.1)

for strings $P, x$ such that $P$ is a valid representation of a NAND++ program which halts and produces an output on $x$. Moreover, for every input $x \in \{0,1\}^*$ on which $P$ does not halt, $U(P, x)$ does not halt as well.

**Proof Idea:** Once you understand what the theorem says, it is not that hard to prove. The desired program $U$ is an interpreter for NAND++ program. That is, $U$ gets a representation of the program $P$ (think of the source code), and some input $x$, and needs to simulate the execution of $P$ on $x$.

Think of how you would do that in your favorite programming language. You would use some data structure, such as a dictionary, to store the values of all the variables and arrays of $P$. Then, you could simulate $P$ line by line, updating the data structure as you go along. The interpreter will continue the simulation until $\text{loop}$ is equal to 0.
Once you do that, translating this interpreter from your programming language to NAND++ can be done just as we have seen in Chapter 7. The end result is what’s known as a “meta-circular evaluator”: an interpreter for a programming language in the same one. This is a concept that has a long history in computer science starting from the original universal Turing machine. See also Fig. 8.1.

\[ \text{Figure 8.1: A particularly elegant example of a “meta-circular evaluator” comes from John McCarthy’s 1960 paper, where he defined the Lisp programming language and gave a Lisp function that evaluates an arbitrary Lisp program (see above). Lisp was not initially intended as a practical programming language and this example was merely meant as an illustration that the Lisp universal function is more elegant than the universal Turing machine, but McCarthy’s graduate student Steve Russell suggested that it can be implemented. As McCarthy later recalled, “I said to him, ho, ho, you’re confusing theory with practice, this eval is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the eval in my paper into IBM 704 machine code, fixing a bug, and then advertised this as a Lisp interpreter, which it certainly was”.

\[ \text{Theorem 8.1 yields a stronger notion than the universality we proved for NAND, in the sense that we show a single universal NAND++ program } U \text{ that can evaluate all NAND programs, including those that have more lines than the lines in } U. \]  

This also occurs in practice. For example the C compiler can be and is used to execute programs that are more complicated than itself.

\[ \text{Because we can transform other computational models, including NAND, } \lambda \text{ calculus, or a C program, this means that even the seemingly “weak” NAND++ programming language is powerful enough to contain an interpreter for all these models.} \]
To show the full proof of Theorem 8.1, we need to make sure \( \text{EVAL} \) is well defined by specifying a representation for NAND++ programs. As mentioned, one perfectly fine choice is the ASCII representation of the source code. But for concreteness, we can use the following representation:

**Representing NAND++ programs.** If \( P \) is a NAND++ program with \( a \) array variables and \( b \) scalar variables, then every iteration of \( P \) is obtained by computing a NAND program \( P' \) with \( a + b \) inputs and outputs that updates these variables (where the array variables are read and written to at the special location \( i \)).

So, we can use the list-of-triples representation of \( P' \) to represent \( P \). That is, we represent \( P \) by a tuple \((a, b, L)\) where \( L \) is a list of triples of numbers in \( \{0, \ldots, a + b - 1\} \). Each triple \((j, k, \ell)\) in \( L \) corresponds to a line of code in \( P \) of the form \( \text{foo} = \text{NAND} (\text{bar}, \text{blah}) \). The indices \( j, k, \ell \) correspond to array variables if they are in \( \{0, \ldots, a - 1\} \) and to scalar variables if they are in \( \{a, \ldots, a + b - 1\} \). We will identify the arrays \( X, X\text{valid}, Y, Y\text{valid} \) with the indices 0, 1, 2, 3 and the scalar \( \text{loop} \) with the index \( a \). (Once again, the precise details of the representation do not matter much; we could have used any other.)

**Proof of Theorem 8.1.** We will only sketch the proof, giving the major ideas. First, we observe that we can easily write a Python program that, on input a representation \( P = (a, b, L) \) of a NAND++ program and an input \( X \), evaluates \( P \) on \( X \). Here is the code of this program for concreteness, though you can feel free to skip it if you are not familiar (or interested) in Python:

```python
def EVAL(P, X):
    """Get NAND++ prog P represented as (a,b,L) and input X, produce output""
    a, b, L = *P
    vars = {} # scalar variables: for j in [a..a+b-1], vars[j] is value of scalar variable j
    arrs = {} # array variables: for j in {0..a-1}, arrs[(j,i)] is -ith position of array j

    # Special variable indices:
    # X:0, Xvalid:1, Y:2, Yvalid:3, loop:a
```
def setvar(j, v):  # set variable j to value v
    if j > a: vars[j] = v  # j is scalar
    else: arrs[(j, i)] = v  # j is array

def getvar(j):  # get value of var j (if j array
    # then at current index i)
    if j > a: return vars.get(j, 0)
    return arrs.get((j, i), 0)

def NAND(a, b): return 1 - a * b

# copy input
for j in range(len(X)):
    arrs[(0, j)] = X[j]  # X has index 0
    arrs[(1, j)] = 1  # Xvalid has index 1

maxseen = 0
i = 0
dir = 1  # +1: increase, -1: decrease
while True:
    for (j, k, l) in L:
        setvar(j, NAND(getvar(k), getvar(l)))
    if not getvar(a):
        break  # loop has index a
    i += dir
    if not i: dir = 1
    if i > maxseen:
        dir = -1
        maxseen = i

# copy output
i = 0
res = []
while getvar(3):  # if Yvalid[i]=1
    res += [getvar(2)]  # add Y[i] to result
    i += 1
return Y

Translating this Python code to NAND++ code line by line is a mechanical, even if somewhat laborious, process. However, to prove the theorem we don’t need to write the code fully, but can use our “eat the cake and have it too” paradigm. That is, while we can assume that our input program $P$ is written in the lowly NAND++ programming languages, in writing the program $U$ we are allowed
to use richer models such as NAND« (since they are equivalent by Theorem 7.1). Translating the above Python code to NAND« is truly straightforward. The only issue is that NAND« doesn’t have the dictionary data structure built in, but we can represent a dictionary of the form \{key_0 : val_0, ..., key_{m-1} : val_{m-1}\} by simply a string (stored in an array) which is the list of pairs (key_0, val_0), ..., (key_{m-1}, val_{m-1}) (where each pair is represented as a string in some prefix-free way). To retrieve an element with key k we can scan the list from beginning to end and compare each key_i with k. Similarly we scan the list to update the dictionary with a new value, either modifying it or appending the (key, val) pair at the end. The above is a very inefficient way to implement the dictionary data structure in practice, but it suffices for the purpose of proving the theorem.\(^3\)

8.2 IS EVERY FUNCTION COMPUTABLE?

We saw that NAND programs can compute every finite function. A natural guess is that NAND++ programs could compute every infinite function. However, this turns out to be false, even for functions with 0/1 output. That is, there exists a function \( F : \{0,1\}^* \to \{0,1\} \) that is uncomputable! This is actually quite surprising, if you think about it. Our intuitive notion of a “function” (and the notion most scholars had until the 20th century) is that a function \( f \) defines some implicit or explicit way of computing the output \( f(x) \) from the input \( x \).\(^4\) The notion of an “uncomputable function” thus seems to be a contradiction in terms, but yet the following theorem shows that such creatures do exist:

**Theorem 8.2 — Uncomputable functions.** There exists a function \( F^* : \{0,1\}^* \to \{0,1\} \) that is not computable by any NAND++ program.

**Proof.** The proof is illustrated in Fig. 8.2. We start by defining the following function \( G : \{0,1\}^* \to \{0,1\} \):

For every string \( x \in \{0,1\}^* \), if \( x \) satisfies (1) \( x \) is a valid representation of a NAND++ program \( P_x \) and (2) when the program \( P_x \) is executed on the input \( x \) it halts and produces an output, then we define \( G(x) \) as the first bit of this output. Otherwise (i.e., if \( x \) is not a valid representation of a program, or the program \( P_x \) never halts on \( x \)) we define \( G(x) = 0 \). We define \( F^*(x) := 1 - G(x) \).

We claim that there is no NAND++ program that computes \( F^* \). Indeed, suppose, towards the sake of contradiction, that there was some program \( P \) that computed \( F^* \), and let \( x \) be the binary string that represents the program \( P \). Then on input \( x \), the program \( P \) outputs \( F^*(x) \). But by definition, the program should also output \( 1 - F^*(x) \),

\(^3\) Reading and writing to a dictionary of \( m \) values in this implementation takes \( \Omega(m) \) steps, while it is in fact possible to do this in \( O(1) \) steps using a hash table. Since NAND« models a RAM machine which corresponds to modern electronic computers, we can also implement a hash table in NAND«.

\(^4\) In the 1800’s, with the invention of the Fourier series and with the systematic study of continuity and differentiability, people have starting looking at more general kinds of functions, but the modern definition of a function as an arbitrary mapping was not yet universally accepted. For example, in 1899 Poincare wrote “we have seen a mass of bizarre functions which appear to be forced to resemble as little as possible honest functions which serve some purpose. … they are invented on purpose to show that our ancestor’s reasoning was at fault, and we shall never get anything more than that out of them”.

hence yielding a contradiction.

$$\begin{array}{ccccccc}
 & 0 & 1 & 01 & 10 & \cdots & \lambda & \cdots \\
0 & -P_0(0) & -P_0(1) & -P_0(0) & -P_0(1) & \cdots & -P_0(\lambda) & \cdots \\
1 & -P_1(0) & -P_1(1) & -P_1(0) & -P_1(1) & \cdots & -P_1(\lambda) & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\lambda & -P_\lambda(0) & -P_\lambda(1) & -P_\lambda(0) & -P_\lambda(1) & \cdots & -P_\lambda(\lambda) & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\end{array}$$

**Figure 8.2:** We construct an uncomputable function by defining for every two strings $x, y$ the value $1 - P_y(x)$ which equals 0 if the program described by $y$ outputs 1 on $x$, and 1 otherwise. We then define $F^*(x)$ to be the “diagonal” of this table, namely $F^*(x) = 1 - P_y(x)$ for every $x$. The function $F^*$ is uncomputable, because if it was computable by some program whose string description is $x^*$ then we would get that $P_y(x^*) = F(x^*) = 1 - P_y(x^*)$.

The proof of Theorem 8.2 is short but subtle. I suggest that you pause here and go back to read it again and think about it - this is a proof that is worth reading at least twice if not three or four times. It is not often the case that a few lines of mathematical reasoning establish a deeply profound fact - that there are problems we simply cannot solve and the “firm conviction” that Hilbert alluded to above is simply false.

The type of argument used to prove Theorem 8.2 is known as diagonalization since it can be described as defining a function based on the diagonal entries of a table as in Fig. 8.2. The proof can be thought of as an infinite version of the counting argument we used for showing lower bound for NAND programs in Theorem 5.6. Namely, we show that it’s not possible to compute all functions from $\{0, 1\}^* \rightarrow \{0, 1\}$ by NAND++ programs simply because there are more functions like that then there are NAND++ programs.
8.3 THE HALTING PROBLEM

Theorem 8.2 shows that there is some function that cannot be computed. But is this function the equivalent of the “tree that falls in the forest with no one hearing it”? That is, perhaps it is a function that no one actually wants to compute. It turns out that there are natural uncomputable functions:

**Theorem 8.3 — Uncomputability of Halting function.** Let \( HALT : \{0,1\}^* \rightarrow \{0,1\} \) be the function such that \( HALT(P,x) = 1 \) if the NAND++ program \( P \) halts on input \( x \) and equals 0 if it does not. Then \( HALT \) is not computable.

Before turning to prove Theorem 8.3, we note that \( HALT \) is a very natural function to want to compute. For example, one can think of \( HALT \) as a special case of the task of managing an “App store”. That is, given the code of some application, the gatekeeper for the store needs to decide if this code is safe enough to allow in the store or not. At a minimum, it seems that we should verify that the code would not go into an infinite loop.

**Proof Idea:** One way to think about this proof is as follows:

Uncomputability of \( F^* + \) Universality = Uncomputability of \( HALT \) (8.2)

That is, we will use the universal program that computes \( EVAL \) to derive the uncomputability of \( HALT \) from the uncomputability of \( F^* \) shown in Theorem 8.2. Specifically, the proof will be by contradiction. That is, we will assume towards a contradiction that \( HALT \) is computable, and use that assumption, together with the universal program of Theorem 8.1, to derive that \( F^* \) is computable, which will contradict Theorem 8.2. *

**Proof of Theorem 8.3.** The proof will use the previously established Theorem 8.2, as illustrated in Fig. 8.3. That is, we will assume, towards a contradiction, that there is NAND++ program \( P^* \) that can compute the \( HALT \) function, and use that to derive that there is some NAND++ program \( Q^* \) that computes the function \( F^* \) defined above, contradicting Theorem 8.2. (This is known as a proof by reduction, since we reduce the task of computing \( F^* \) to the task of computing \( HALT \). By the contrapositive, this means the uncomputability of \( F^* \) implies the uncomputability of \( HALT \).)

Indeed, suppose that \( P^* \) was a NAND++ program that computes \( HALT \). Then we can write a NAND++ program \( Q^* \) that does the following on input \( x \in \{0,1\}^* \):\(^5\)

\(^5\) Note that we are using here a “high level” description of NAND++ programs. We know that we can implement the steps below, for example by first writing them in NAND and then transforming the NAND program to NAND++. Step 1 involves simply running the program \( P^* \) on some input.
Figure 8.3. We prove that $HALT$ is uncomputable using a reduction from computing the previously shown uncomputable function $F^*$ to computing $HALT$. We assume that we had an algorithm that computes $HALT$ and use that to obtain an algorithm that computes $F^*$.

Program $Q^*(x)$
1. Compute $z = P^*(x, x)$
2. If $z = 0$ then output 1.
3. Otherwise, if $z = 1$ then let $y$ be the first bit of $EVAL(x, x)$ (i.e., evaluate the program described by $x$ on the input $x$). If $y = 1$ then output 0. Otherwise output 1.

We make the following claim about $Q^*$:

Claim: For every $x \in \{0, 1\}^*$, if $P^*(x, x) = HALT(x, x)$ then the program $Q^*(x) = F^*(x)$ where $F^*$ is the function from the proof of Theorem 8.2.

Note that the claim immediately implies that our assumption that $P^*$ computes $HALT$ contradicts Theorem 8.2, where we proved that the function $F^*$ is uncomputable. Hence the claim is sufficient to prove the theorem.

Proof of claim: Let $x$ be any string. If the program described by $x$ halts on input $x$ and its first output bit is 1 then $F^*(x) = 0$ and the output $Q^*(x)$ will also equal 0 since $z = HALT(x, x) = 1$, and hence in
step 3 the program $Q^*$ will run in a finite number of steps (since the program described by $x$ halts on $x$), obtain the value $y = 1$ and output 0.

Otherwise, there are two cases. Either the program described by $x$ does not halt on $x$, in which case $z = 0$ and $Q^*(x) = 1 = F^*(x)$. Or the program halts but its first output bit is not 1. In this case $z = 1$ but the value $y$ computed by $Q^*(x)$ is not 1 and so $Q^*(x) = 1 = F^*(x)$.

As we discussed above, the desired contradiction is directly implied by the claim. 

Once again, this is a proof that’s worth reading more than once. The uncomputability of the halting problem is one of the fundamental theorems of computer science, and is the starting point for much of the investigations we will see later. An excellent way to get a better understanding of Theorem 8.3 is to do which asks you to prove an alternative proof of the same result.

8.3.1 Is the Halting problem really hard? (discussion)

Many people’s first instinct when they see the proof of Theorem 8.3 is to not believe it. That is, most people do believe the mathematical statement, but intuitively it doesn’t seem that the Halting problem is really that hard. After all, being uncomputable only means that $HALT$ cannot be computed by a NAND++ program. But programmers seem to solve $HALT$ all the time by informally or formally arguing that their programs halt. While it does occasionally happen that a program unexpectedly enters an infinite loop, is there really no way to solve the halting problem? Some people argue that they can, if they think hard enough, determine whether any concrete program that they are given will halt or not. Some have even argued that humans in general have the ability to do that, and hence humans have inherently superior intelligence to computers or anything else modeled by NAND++ programs (aka Turing machines).\(^6\)

The best answer we have so far is that there truly is no way to solve $HALT$, whether using Macs, PCs, quantum computers, humans, or any other combination of mechanical and biological devices. Indeed this assertion is the content of the Church-Turing Thesis. This of course does not mean that for every possible program $P$, it is hard to decide if $P$ enters an infinite loop. Some programs don’t even have loops at all (and hence trivially halt), and there are many other far

\(^6\) This argument has also been connected to the issues of consciousness and free will. I am not completely sure of its relevance but perhaps the reasoning is that humans have the ability to solve the halting problem but they exercise their free will and consciousness by choosing not to do so.
less trivial examples of programs that we can certify to never enter an infinite loop (or programs that we know for sure that will enter such a loop). However, there is no general procedure that would determine for an arbitrary program $P$ whether it halts or not. Moreover, there are some very simple programs for which it not known whether they halt or not. For example, the following Python program will halt if and only if Goldbach’s conjecture is false:

```python
def isprime(p):
    return all(p % i for i in range(2, p-1))

def Goldbach(n):
    return any(isprime(p) and isprime(n-p) for p in range(2, n-1))

n = 4
while True:
    if not Goldbach(n): break
    n+= 2
```

Given that Goldbach’s Conjecture has been open since 1742, it is unclear that humans have any magical ability to say whether this (or other similar programs) will halt or not.

**Figure 8.4**: XKCD’s take on solving the Halting problem, using the principle that “in the long run, we’ll all be dead”.

### 8.3.2 Reductions

The Halting problem turns out to be a linchpin of uncomputability, in the sense that Theorem 8.3 has been used to show the uncomputability of a great many interesting functions. We will see several examples in such results in this chapter and the exercises, but there are many more such results in the literature (see Fig. 8.5).
The idea behind such uncomputability results is conceptually simple but can at first be quite confusing. If we know that $HALT$ is uncomputable, and we want to show that some other function $BLAH$ is uncomputable, then we can do so via a contrapositive argument (i.e., proof by contradiction). That is, we show that if we had a NAND++ program that computes $BLAH$ then we could have a NAND++ program that computes $HALT$. (Indeed, this is exactly how we showed that $HALT$ itself is uncomputable, by showing this follows from the uncomputability of the function $F^*$ from Theorem 8.2.)

For example, to prove that $BLAH$ is uncomputable, we could show that there is a computable function $R : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every pair $P$ and $x$, $HALT(P, x) = BLAH(R(P, x))$. Such a function is known as a reduction, because we are reducing the task of computing $HALT$ to the task of computing $BLAH$. The confusing part about reductions is that we are assuming something we believe is false (that $BLAH$ has an algorithm) to derive something that we know is false (that $HALT$ has an algorithm). For this reason Michael Sipser describes such results as having the form “If pigs could whistle then horses could fly”.

A reduction-based proof has two components. For starters, since we need $R$ to be computable, we should describe the algorithm to compute it. This algorithm is known as a reduction since the transformation $R$ modifies an input to $HALT$ to an input to $BLAH$, and hence reduces the task of computing $HALT$ to the task of computing $BLAH$. The second component of a reduction-based proof is the analysis. For example, in the example above, we need to prove $HALT(P, x) = BLAH(R(P, x))$. The equality $HALT(P, x) = BLAH(R(P, x))$ boils down to proving two implications. We need to prove that (i) if $P$ halts on $x$ then $BLAH(R(P, x)) = 1$ and (ii) if $P$ does not halt on $x$ then $BLAH(R(P, x)) = 0$. When you’re coming up with a reduction based proof, it is useful to separate the two components of describing the reduction and analyzing it. Furthermore it is often useful to separate the analysis into two components corresponding to the implications (i) and (ii) above.

At the end of the day reduction-based proofs are just like other proofs by contradiction, but the fact that they involve hypothetical algorithms that don’t really exist tends to make such proofs quite confusing. The one silver lining is that at the end of the day the notion of reductions is mathematically quite simple, and so it’s not that bad even if you have to go back to first principles every time you need to remember what is the direction that a reduction should go in. (If this discussion itself is confusing, feel free to ignore it; it might become clearer after you see an example of a reduction such as the proof of Theorem 8.4 or ??.)
8.3.3 A direct proof of the uncomputability of $HALT$ (optional)

It turns out that we can combine the ideas of the proofs of Theorem 8.2 and Theorem 8.3 to obtain a short proof of the latter theorem, that does not appeal to the uncomputability of $F^*$. This short proof appeared in print in a 1965 letter to the editor of Christopher Strachey:³

To the Editor, The Computer Journal.

An Impossible Program

Sir,

A well-known piece of folk-lore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (in a railway carriage on the way to a Conference at the NPL in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers.

³Christopher Strachey was an English computer scientist and the inventor of the CPL programming language. He was also an early artificial intelligence visionary, programming a computer to play Checkers and even write love letters in the early 1950’s, see this New Yorker article and this website.
The version below uses CPL, but not in any essential way.

Suppose \( T[R] \) is a Boolean function taking a routine (or program) \( R \) with no formal or free variables as its arguments and that for all \( R, T[R] = \text{True} \) if \( R \) terminates if run and that \( T[R] = \text{False} \) if \( R \) does not terminate.

Consider the routine \( P \) defined as follows

```cpl
rec routine P

[\text{L}: \text{if } T[P] \text{ go to L}]

\text{Return} [\text{L}]

\text{If } T[P] = \text{True} \text{ the routine } P \text{ will loop, and it will only terminate if } T[P] = \text{False}. \text{ In each case } 'T[P]' \text{ has exactly the wrong value, and this contradiction shows that the function } T \text{ cannot exist.}
```

Yours faithfully,

C. Strachey

Churchill College, Cambridge

Try to stop and extract the argument for proving Theorem 8.3 from the letter above.

Since CPL is not as common today, let us reproduce this proof. The idea is the following: suppose for the sake of contradiction that there exists a program \( T \) such that \( T(f,x) \) equals \text{True} iff \( f \) halts on input \( x \). Then we can construct a program \( P \) and an input \( x \) such that \( T(P,x) \) gives the wrong answer. The idea is that on input \( x \), the program \( P \) will do the following: run \( T(x,x) \), and if the answer is \text{True} then go into an infinite loop, and otherwise halt. Now you can see that \( T(P,P) \) will give the wrong answer: if \( P \) halts when it gets its own code as input, then \( T(P,P) \) is supposed to be \text{True}, but then \( P(P) \) will go into an infinite loop. And if \( P \) does not halt, then \( T(P,P) \) is supposed to be \text{False} but then \( P(P) \) will halt. We can also code this up in Python:

```python
def CantSolveMe(T):
    """
    Gets function \( T \) that claims to solve \text{HALT}.
    Returns a pair \((P,x)\) of code and input on which \( T(P,x) \neq \text{HALT}(x) \)
    """

def fool(x):
    if T(x,x):
```

Strachey’s letter considers the no-input variant of \text{HALT}, but as we’ll see, this is an immaterial distinction.
```python
while True: pass
return "I halted"
```

return (fool, fool)

For example, consider the following Naive Python program `T` that guesses that a given function does not halt if its input contains `while` or `for`:

def T(f, x):
    """Crude halting tester - decides it doesn't
    halt if it contains a loop."""
    import inspect
    source = inspect.getsource(f)
    if source.find("while"): return False
    if source.find("for"): return False
    return True

If we now set `(f, x) = CantSolveMe(T)`, then `T(f, x) = False` but `f(x)` does in fact halt. This is of course not specific to this particular `T`: for every program `T`, if we run `(f, x) = CantSolveMe(T)` then we’ll get an input on which `T` gives the wrong answer to `HALT`.

### 8.4 Impossibility of General Software Verification

The uncomputability of the Halting problem turns out to be a special case of a much more general phenomenon. Namely, that we cannot certify semantic properties of general purpose programs. “Semantic properties” mean properties of the function that the program computes, as opposed to properties that depend on the particular syntax. For example, we can easily check whether or not a given C program contains no comments, or whether all function names begin with an upper case letter. As we’ve seen, we cannot check whether a given program enters into an infinite loop or not.

But we could still hope to check some other properties of the program. For example, we could hope to certify that a given program `M` correctly computes the multiplication operation, or that no matter what input the program is provided with, it will never reveal some confidential information. Alas it turns out that the task of checking that a given program conforms with such a specification is uncomputable. We start by proving a simple generalization of the Halting problem:

**Theorem 8.4 — Halting without input.** Let $HALTONZERO : \{0, 1\}^* \rightarrow \{0, 1\}$ be the function that on input $P \in \{0, 1\}^*$, maps $P$ to 1 if...
and only if the NAND++ program represented by $P$ halts when supplied the single bit 0 as input. Then $HALTONZERO$ is uncomputable.

The proof of Theorem 8.4 is below, but before reading it you might want to pause for a couple of minutes and think how you would prove it yourself. In particular, try to think of what a reduction from $HALT$ to $HALTONZERO$ would look like. Doing so is an excellent way to get some initial comfort with the notion of proofs by reduction, which is a notion that will recur time and again in this course.

**Proof of Theorem 8.4.** The proof is by reduction from $HALT$. We will assume, towards the sake of contradiction, that $HALTONZERO$ is computable by some algorithm $A$, and use this hypothetical algorithm $A$ to construct an algorithm $B$ to compute $HALT$, hence obtaining a contradiction to Theorem 8.3.

Since this is our first proof by reduction from the Halting problem, we will spell it out in more details than usual. Such a proof by reduction consists of two steps:

1. **Description of the reduction:** We will describe the operation of our algorithm $B$, and how it makes “function calls” to the hypothetical algorithm $A$.

2. **Analysis of the reduction:** We will then prove that under the hypothesis that Algorithm $A$ computes $HALTONZERO$, Algorithm $B$ will compute $HALT$.

Our Algorithm $B$ works as follows:

<table>
<thead>
<tr>
<th>Algorithm $B(P, x)$:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A program $P \in {0, 1}^<em>$ and $x \in {0, 1}^</em>$</td>
</tr>
<tr>
<td><strong>Assumption:</strong> Access to an algorithm $A$ such that $H(Q) = HALTONZERO(Q)$ for every program $Q$.</td>
</tr>
<tr>
<td><strong>Operation:</strong></td>
</tr>
<tr>
<td>1. Let $Q$ denote the program that does the following: “on input $z \in {0, 1}^*$, evaluate $P$ on the input $x$ and return the result”</td>
</tr>
<tr>
<td>2. Feed $Q$ into Algorithm $A$ and denote $y = A(Q)$ be the resulting output.</td>
</tr>
<tr>
<td>3. Output $y$.</td>
</tr>
</tbody>
</table>
That is, on input a pair \((P, x)\) the algorithm \(B\) uses this pair to construct a program \(Q\), feeds this program to \(A\), and outputs the result. The program \(Q\) is one that ignores its input and simply runs \(P\) on \(x\). Note however that our algorithm \(B\) does not actually execute the program \(Q\); it merely constructs it and feeds it to \(A\).

We now discuss exactly how does algorithm \(B\) performs step 1 of obtaining the source code of the program \(Q\) from the pair \((P, x)\). In fact, constructing the program \(Q\) is rather simple. We can do so by modifying \(P\) to ignore its input and use \(x\) instead. Specifically, if \(x\) is of length \(n\) we can do so by adding \(2n\) lines of initialization code that sets arrays \(MyX\) and \(MyXvalid\) to the values corresponding to \(x\) (i.e., \(MyX[i] = x_i\) and \(MyXvalid[i] = 1\) for every \(i \in [n]\)). The rest of the program \(Q\) is obtained by replacing all references to \(X\) and \(Xvalid\) with references to \(MyX\) and \(MyXvalid\) respectively. One can see that on every input \(z \in \{0, 1\}^*\), (and in particular for \(z = 0\)) executing \(Q\) on input \(z\) will correspond to executing \(P\) on the input \(x\).

The above completes the description of the reduction. The analysis is obtained by proving the following claim:

**CLAIM:** Define by \(Q(P, x)\) the program \(Q\) that Algorithm \(B\) constructs in step 1 when given as input \(P\) and \(x\). Then for every program \(P\) and input \(x\), \(Q(P, x)\) halts on the input \(0\) if and only if \(P\) halts on the input \(x\).

**Proof of claim:** Let \(P, x\) be some program and input and let \(Q = Q(P, x)\). Since \(Q\) ignores its input and simply evaluates \(P\) on the input \(x\), for every input \(z\) for \(Q\), and so in particular for the input \(z = 0\), \(Q\) will halt on the input \(z\) if and only if \(P\) halts on the input \(x\).

The claim implies that \(HALTONZERO(Q(P, x)) = HALT(P, x)\). Thus if the hypothetical algorithm \(A\) satisfies \(A(Q) = HALTONZERO(Q)\) for every \(Q\) then the algorithm \(B\) we construct satisfies \(B(P, x) = HALT(P, x)\) for every \(P, x\), contradicting the uncomputability of \(HALT\).

---

**The hardwiring technique** In the proof of Theorem 8.4 we used the technique of “hardwiring” an input \(x\) to a program \(P\). That is, modifying a program \(P\) that it uses “hardwired constants” for some of all of its input. This technique is quite common in reductions and elsewhere, and we will often use it again in this course.

Once we show the uncomputability of \(HALTONZERO\) we can extend to various other natural functions:
**Theorem 8.5 — Computing all zero function.** Let $\text{ZEROFUNC} : \{0,1\}^* \to \{0,1\}$ be the function that on input $P \in \{0,1\}^*$, maps $P$ to 1 if and only if the NAND++ program represented by $P$ outputs 0 on every input $x \in \{0,1\}^*$. Then $\text{ZEROFUNC}$ is uncomputable.

*Proof.* The proof is by reduction to $\text{HALTONZERO}$. Suppose, towards the sake of contradiction, that there was an algorithm $A$ such that $A(P') = \text{ZEROFUNC}(P')$ for every $P' \in \{0,1\}^*$. Then we will construct an algorithm $B$ that solves $\text{HALTONZERO}$. Given a program $P$, Algorithm $B$ will construct the following program $P'$: on input $x \in \{0,1\}^*$, $P'$ will first run $P(0)$, and then output 0.

Now if $P$ halts on 0 then $P'(x) = 0$ for every $x$, but if $P$ does not halt on 0 then $P'$ will never halt on every input and in particular will not compute $\text{ZEROFUNC}$. Hence, $\text{ZEROFUNC}(P') = 1$ if and only if $\text{HALTONZERO}(P) = 1$. Thus if we define algorithm $B$ as $B(P) = A(P')$ (where a program $P$ is mapped to $P'$ as above) then we see that if $A$ computes $\text{ZEROFUNC}$ then $B$ computes $\text{HALTONZERO}$, contradicting Theorem 8.4. ■

Another result along similar lines is the following:

**Theorem 8.6 — Uncomputability of verifying parity.** The following function is uncomputable

$$\text{COMPUTES-PARITY}(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{otherwise} \end{cases}$$

(8.3)

We leave the proof of Theorem 8.6 as an exercise (Exercise 8.1). I strongly encourage you to stop here and try to solve this exercise.

### 8.4.1 Rice’s Theorem

?? can be generalized far beyond the parity function and in fact it rules out verifying any type of semantic specification on programs. We define a *semantic specification* on programs to be some property that does not depend on the code of the program but just on the function that the program computes.

For example, consider the following two C programs

```c
int First(int k) {
    return 2*k;
}
```
While we contrast “semantic” with “syntactic” in this informal discussion, we only formally define the notion of a semantic function in this book. A famous example of a syntactically correct but semantically meaningless sentence in English is Chomsky’s “Colorless green ideas sleep furiously.”

Often the properties of programs that we are most interested in are the semantic ones, since we want to understand the programs’ functionality. Unfortunately, the following theorem shows that such properties are uncomputable in general:

**Theorem 8.7 — Rice’s Theorem (slightly restricted version).** We say that two strings \( \mathcal{P} \) and \( \mathcal{Q} \) representing NAND++ programs are functionally equivalent, denoted by \( \mathcal{P} \equiv \mathcal{Q} \), if for every \( x \in \{0,1\}^* \), either both \( \mathcal{P} \) and \( \mathcal{Q} \) don’t halt on \( x \), or \( \mathcal{P}(x) = \mathcal{Q}(x) \). We say that a function \( F : \{0,1\}^* \to \{0,1\} \) is semantic if for every functionally equivalent \( \mathcal{P} \) and \( \mathcal{Q} \), \( F(P) = F(Q) \).

Then the only semantic computable total functions \( F : \{0,1\}^* \to \{0,1\} \) are the constant zero function and the constant one function.

**Proof Idea:** The idea behind the proof is to show that every semantic non-trivial function \( F \) is at least as hard to compute as HALTONZERO. This will conclude the proof since by Theorem 8.4, HALTONZERO is uncomputable. If a function \( F \) is non trivial then there are two programs \( P_0 \) and \( P_1 \) such that \( F(P_0) = 0 \) and \( F(P_1) = 1 \). So, the goal would be to take a program \( P \) and find a way to map it into a program \( Q = R(P) \), such that (i) if \( P \) halts on zero then \( Q \) computes the same partial function as \( P_1 \) and (ii) if \( P \) does not halt on zero then \( Q \) computes the same partial function as \( P_0 \). Because \( F \) is semantic, this would mean that \( HALTONZERO(P) = F(R(P)) \), and hence would show that if \( F \) was computable, then HALTONZERO would be computable as well, contradicting Theorem 8.4. The details...
Introduction to Theoretical Computer Science

Note that the program $ղ$ has different code than $ժկէ$. It is not the same program, but it does have the same behavior (in this case) of never halting on any input.

Proof of Theorem 8.7. We will not give the proof in full formality, but rather illustrate the proof idea by considering a particular semantic function $F$. Define $MONOTONE : \{0,1\}^* \rightarrow \{0,1\}$ as follows: $MONOTONE(P) = 1$ if there does not exist $n \in \mathbb{N}$ and two inputs $x, x' \in \{0,1\}^n$ such that for every $i \in [n]$ $x_i \leq x'_i$ but $P(x)$ outputs 1 and $P(x') = 0$. That is, $MONOTONE(P) = 1$ if it’s not possible to find an input $x$ such that flipping some bits of $x$ from 0 to 1 will change $P$’s output in the other direction from 1 to 0. We will prove that $MONOTONE$ is uncomputable, but the proof will easily generalize to any semantic function. For starters we note that $MONOTONE$ is not actually the all zeroes or all one function:

- The program $INF$ that simply goes into an infinite loop satisfies $MONOTONE(INF) = 1$, since $INF$ is not defined anywhere and so in particular there are no two inputs $x, x'$ where $x_i \leq x'_i$ for every $i$ but $INF(x) = 0$ and $INF(x') = 1$.

- The program $PAR$ that we’ve seen, which computes the XOR or parity of its input, is not monotone (e.g., $PAR(1, 1, 0, 0, \ldots, 0) = 0$ but $PAR(1, 0, 0, \ldots, 0) = 0$) and hence $MONOTONE(PAR) = 0$.

(It is important to note that in the above we talk about programs $INF$ and $PAR$ and not the corresponding functions that they compute.)

We will now give a reduction from $HALTONZERO$ to $MONOTONE$. That is, we assume towards a contradiction that there exists an algorithm $A$ that computes $MONOTONE$ and we will build an algorithm $B$ that computes $HALTONZERO$. Our algorithm $B$ will work as follows:

\[\text{Algorithm } B(P):\]

1. On input a program $P \in \{0,1\}^*$, $B$ will construct the following program $Q$: “on input $z \in \{0,1\}^*$ do: a. Run $P(0)$, b. Return $PAR(z)$”.
2. $B$ will then return the value $1 - A(Q)$.

To complete the proof we need to show that $B$ outputs the correct answer, under our assumption that $A$ computes $MONOTONE$. In other words, we need to show that $HALTONZERO(P) = 1 - MONOTONE(Q)$. However, note that if $P$ does not halt on zero, then the program $Q$ enters into an infinite loop in step a. and will never reach step b. Hence in this case the program $Q$ is functionally equivalent to $INF$.$^{11}$ Thus, $MONOTONE(Q) = MONOTONE(INF) = 1$.

$^{11}$ Note that the program $Q$ has different code than $INF$. It is not the same program, but it does have the same behavior (in this case) of never halting on any input.
If \( P \) does halt on zero, then step a. in \( Q \) will eventually conclude and \( Q \)'s output will be determined by step b., where it simply outputs the parity of its input. Hence in this case, \( Q \) computes the non-monotone parity function (i.e., is functionally equivalent to \( PAR \)), and so we get that \( MONOTONE(Q) = MONOTONE(PAR) = 0 \). In both cases we see that \( MONOTONE(Q) = 1 - HALTONZERO(P) \), which is what we wanted to prove. An examination of this proof shows that we did not use anything about \( MONOTONE \) beyond the fact that it is semantic and non-trivial (in the sense that it is not the all zero, nor the all-ones function).

\[ \square \]

**Semantic is not the same as uncomputable** Rice’s Theorem is so powerful and such a popular way of proving uncomputability that people sometimes get confused and think that it is the only way to prove uncomputability. In particular, a common misconception is that if a function \( F \) is not semantic then it is computable. This is not at all the case. For example, consider the following function \( HALTNOYALE : \{0,1\}^* \rightarrow \{0,1\} \). This is a function that on input a string that represents a NAND++ program \( P \), outputs 1 if and only if both (i) \( P \) halts on the input 0, and (ii) the program \( P \) does not contain a variable with the identifier Yale. The function \( HALTNOYALE \) is clearly not semantic, as it will output two different values when given as input one of the following two functionally equivalent programs:

\[
Yale[0] = \text{NAND}(X[0],Yale[0]) \\
Y[0] = \text{NAND}(X[0],Yale[0])
\]

and

\[
Harvard[0] = \text{NAND}(X[0],Harvard[0]) \\
Y[0] = \text{NAND}(X[0],Harvard[0])
\]

However, \( HALTNOYALE \) is uncomputable since every program \( P \) can be transformed into an equivalent (and in fact improved :) ) program \( P' \) that does not contain the variable Yale. Hence if we could compute \( HALTNOYALE \) then we could compute also \( HALTONZERO \).

Moreover, as we will see in Chapter 10, there are uncomputable functions whose inputs are not programs, and hence for which the adjective “semantic” is not applicable.
8.4.2 Halting and Rice’s Theorem for other Turing-complete models

As we saw before, many natural computational models turn out to be equivalent to one another, in the sense that we can transform a “program” of one model (such as a λ expression, or a game-of-life configurations) into another model (such as a NAND++ program). This equivalence implies that we can translate the uncomputability of the Halting problem for NAND++ programs into uncomputability for Halting in other models. For example:

Theorem 8.8 — Turing Machine Halting. Let \( TMHALT : \{0, 1\}^* \rightarrow \{0, 1\} \) be the function that on input strings \( M \in \{0, 1\}^* \) and \( x \in \{0, 1\}^* \) outputs 1 if the Turing machine described by \( M \) halts on the input \( x \) and outputs 0 otherwise. Then \( TMHALT \) is uncomputable.

Proof. We have seen in Theorem 6.12 that for every NAND++ program \( P \) there is an equivalent Turing machine \( M_P \), such that for every \( x \), that computes the same function. The machine \( M_P \), exactly simulated \( P \), in the sense that \( M_P \) halts on \( x \) if and only \( P \) halts on \( x \) (and moreover if they both halt, they produce the same output). Going back to the proof of Theorem 6.12, we can see that the transformation of the program \( P \) to the Turing machine \( M(P) \) was described in a constructive way.

Specifically, we gave explicit instructions how to build the Turing machine \( M(P) \) given the description of the program \( P \). Thus, we can view the proof of Theorem 6.12 as a high level description of an algorithm to obtain \( M_P \) from the program \( P \), and using our “have your cake and eat it too” paradigm, this means that there exists also a NAND++ program \( R \) such that computes the map \( P \mapsto M_P \). We see that

\[
HALT(P, x) = TMHALT(M_P, x) = TMHALT(R(P), x) \tag{8.4}
\]

and hence if we assume (towards the sake of a contradiction) that \( TMHALT \) is computable then Eq. (8.4) implies that \( HALT \) is computable, hence contradicting Theorem 8.3.

The same proof carries over to other computational models such as the \( \lambda \) calculus, two dimensional (or even one-dimensional) automata etc.
Hence for example, there is no algorithm to decide if a λ expression
evaluates the identity function, and no algorithm to decide whether
an initial configuration of the game of life will result in eventually
coloring the cell (0, 0) black or not.

We can also generalize Rice’s Theorem to any Turing complete
model (see Definition 7.13):

**Theorem 8.9 — Rice’s Theorem for general models (optional).** Let
\( \mathcal{F} \) be the set of all partial functions from \( \{0, 1\}^* \) to \( \{0, 1\}^* \) and
\( \mathcal{M} : \{0, 1\}^* \rightarrow \mathcal{F} \) be a Turing complete model. Then for every func-
tion \( \mathcal{P} : \mathcal{F} \rightarrow \{0, 1\} \) that is not the constant zero or one function,
the function \( F_\mathcal{P} : \{0, 1\}^* \rightarrow \{0, 1\} \) defined as \( F_\mathcal{P}(Q) = \mathcal{P}(\mathcal{M}(Q)) \) is
uncomputable (by NAND++ programs).

Proof of Theorem 8.9. We only sketch the proof. This is actually a fairly
straightforward corollary of the “standard” Rice’s Theorem (Theo-
rem 8.7). Any non-trivial property of partial functions \( \mathcal{P} : \mathcal{F} \rightarrow \{0, 1\} \)
gives rise to a semantic and non-trivial function on NAND++ pro-
gressions \( G_\mathcal{P} : \{0, 1\}^* \rightarrow \{0, 1\} \). That is, \( G_\mathcal{P}(P) \) equals \( \mathcal{P}(F_\mathcal{P}) \) where
\( F_\mathcal{P} \) is the function computed by the program \( P \). By Rice’s Theorem,
\( G_\mathcal{P} \) will be uncomputable. However, if \( \mathcal{M} \) is a Turing-complete model,
and we could compute the function \( F_\mathcal{P} \) defined as \( F_\mathcal{P}(Q) = \mathcal{P}(\mathcal{M}(Q)) \)
then we could compute \( G_\mathcal{P} \) by simply using

\[
G_\mathcal{P}(P) = F_\mathcal{P}(\text{ENCODE}_\mathcal{M}(P))
\]  

(8.5)

where \( \text{ENCODE}_\mathcal{M} \) is the function that maps a NAND++ program
\( P \) into a program in \( \mathcal{M} \) that computes the same function. Such com-
putatle a function \( \text{ENCODE}_\mathcal{M} \) exists by the definition of Turing com-
pleteness (Definition 7.13).

**8.4.3 Is software verification doomed? (discussion)**

Programs are increasingly being used for mission critical purposes,
whether it’s running our banking system, flying planes, or monitoring
nuclear reactors. If we can’t even give a certification algorithm that a program correctly computes the parity function, how can we ever be assured that a program does what it is supposed to do? The key insight is that while it is impossible to certify that a general program conforms with a specification, it is possible to write a program in the first place in a way that will make it easier to certify. As a trivial example, if you write a program without loops, then you can certify that it halts. Also, while it might not be possible to certify that an arbitrary program computes the parity function, it is quite possible to write a particular program $P$ for which we can mathematically prove that $P$ computes the parity. In fact, writing programs or algorithms and providing proofs for their correctness is what we do all the time in algorithms research.

The field of software verification is concerned with verifying that given programs satisfy certain conditions. These conditions can be that the program computes a certain function, that it never writes into a dangerous memory location, that it respects certain invariants, and others. While the general tasks of verifying this may be uncomputable, researchers have managed to do so for many interesting cases, especially if the program is written in the first place in a formalism or programming language that makes verification easier. That said, verification, especially of large and complex programs, remains a highly challenging task in practice as well, and the number of programs that have been formally proven correct is still quite small. Moreover, even phrasing the right theorem to prove (i.e., the specification) is often a highly non-trivial endeavor.

---

**Lecture Recap**

- There is a universal NAND++ program $U$ such that on input a description of a NAND++ program $P$ and some input $x$, $U(P, x)$ halts and outputs $P(x)$ if (and only if) $P$ halts on input $x$. Unlike in the case of finite computation (i.e., NAND programs / circuits), the input to the program $U$ can be a program $P$ that has more lines than $U$ itself.

- Unlike the finite case, there are actually functions that are inherently uncomputable in the sense that they cannot be computed by any NAND++ program.

- These include not only some “degenerate” or “esoteric” functions but also functions that people have deeply cared about and conjectured that could be computed.

- If the Church-Turing thesis holds then a function $F$ that is uncomputable according to our definition cannot be computed by any finite means.
8.5 EXERCISES

**Disclaimer** Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

**Exercise 8.1 — Computing parity.** Prove Theorem 8.6 without using Rice’s Theorem.

**Exercise 8.2** For each of the following two functions, say whether it is decidable (computable) or not:

1. Given a NAND++ program \( P \), an input \( x \), and a number \( k \), when we run \( P \) on \( x \), does the index variable \( i \) ever reach \( k \)?

2. Given a NAND++ program \( P \), an input \( x \), and a number \( k \), when we run \( P \) on \( x \), does \( P \) ever write to an array at index \( k \)?

8.6 BIBLIOGRAPHICAL NOTES

The universal program and uncomputability of \( \text{HALT} \) was first shown by Turing in 1937, though closely related results were shown by Church a year before. These works built on Gödel’s 1931 incompleteness theorem that we will discuss in Chapter 10.

Talk about intuitionistic, logicist, and formalist approaches for the foundations of mathematics. Perhaps analogy to veganism. State the full Rice’s Theorem and say that it follows from the same proof as in the exercise.

The diagonalization argument used to prove uncomputability of \( F^* \) is of course derived from Cantor’s argument for the uncountability of the reals. In a twist of fate, using techniques originating from the works Gödel and Turing, Paul Cohen showed in 1963 that Cantor’s Continuum Hypothesis is independent of the axioms of set theory, which means that neither it nor its negation is provable from these axioms and hence in some sense can be considered as “neither true nor false.”

See here for recent progress on a related question.

8.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

---

12 The Continuum Hypothesis is the conjecture that for every subset \( S \) of \( \mathbb{R} \), either there is a one-to-one and onto map between \( S \) and \( \mathbb{N} \) or there is a one-to-one and onto map between \( S \) and \( \mathbb{R} \). It was conjectured by Cantor and listed by Hilbert in 1900 as one of the most important problems in mathematics.
8.8 ACKNOWLEDGEMENTS
9

Restricted computational models

“Happy families are all alike; every unhappy family is unhappy in its own way”, Leo Tolstoy (opening of the book “Anna Karenina”).

We have seen that a great many models of computation are Turing equivalent, including our NAND++/NAND« programs and Turing machines, standard programming languages such as C/Python/-Javascript etc., and other models such as the λ calculus and even the game of life. The flip side of this is that for all these models, Rice’s theorem (see Section 8.4.1) holds as well, which means that deciding any semantic property of programs in such a model is uncomputable.

The uncomputability of halting and other semantic specification problems for Turing equivalent models motivates coming up with restricted computational models that are (a) powerful enough to capture a set of functions useful for certain applications but (b) weak enough that we can still solve semantic specification problems on them. In this chapter we will discuss several such examples.

9.1 TURING COMPLETENESS AS A BUG

We have seen that seemingly simple computational models or systems can turn out to be Turing complete. The following webpage lists several examples of formalisms that “accidentally” turned out to Turing complete, including supposedly limited languages such as the C preprocessor, CCS, SQL, sendmail configuration, as well as games such as Minecraft, Super Mario, and the card game “Magic: The gathering”. This is not always a good thing, as it means that such formalisms can give rise to arbitrarily complex behavior. For example, the postscript format (a precursor of PDF) is a Turing-complete programming language meant to describe documents for printing. The expressive power of postscript can allow for short descriptions of very complex images, but it also gives rise to some nasty surprises, such as

Learning Objectives:
• See that Turing completeness is not always a good thing
• Two important examples of non-Turing-complete, always-halting formalisms: regular expressions and context-free grammars.
• The pumping lemmas for both these formalisms, and examples of non regular and non context-free functions.
• Examples of computable and uncomputable semantic properties of regular expressions and context-free grammars.
the attacks described in this page ranging from using infinite loops as a denial of service attack, to accessing the printer’s file system.

Example 9.1 — The DAO Hack. An interesting recent example of the pitfalls of Turing-completeness arose in the context of the cryptocurrency Ethereum. The distinguishing feature of this currency is the ability to design “smart contracts” using an expressive (and in particular Turing-complete) programming language. In our current “human operated” economy, Alice and Bob might sign a contract to agree that if condition X happens then they will jointly invest in Charlie’s company. Ethereum allows Alice and Bob to create a joint venture where Alice and Bob pool their funds together into an account that will be governed by some program \( P \) that decides under what conditions it disburses funds from it. For example, one could imagine a piece of code that interacts between Alice, Bob, and some program running on Bob’s car that allows Alice to rent out Bob’s car without any human intervention or overhead.

Specifically Ethereum uses the Turing-complete programming language solidity which has a syntax similar to Javascript. The flagship of Ethereum was an experiment known as The “Decentralized Autonomous Organization” or The DAO. The idea was to create a smart contract that would create an autonomously run decentralized venture capital fund, without human managers, where shareholders could decide on investment opportunities. The DAO was the biggest crowdfunding success in history and at its height was worth 150 million dollars, which was more than ten percent of the total Ethereum market. Investing in the DAO (or entering any other “smart contract”) amounts to providing your funds to be run by a computer program. i.e., “code is law”, or to use the words the DAO described itself: “The DAO is borne from immutable, unstoppable, and irrefutable computer code”. Unfortunately, it turns out that (as we saw in Chapter 8) understanding the behavior of Turing-complete computer programs is quite a hard thing to do. A hacker (or perhaps, some would say, a savvy investor) was able to fashion an input that would cause the DAO code to essentially enter into an infinite recursive loop in which it continuously transferred funds into their account, thereby cleaning out about 60 million dollars out of the DAO. While this transaction was “legal” in the sense that it complied with the code of the smart contract, it was obviously not what the humans who wrote this code had in mind. There was a lot of debate in the Ethereum community how to handle this, including some partially successful “Robin Hood” attempts to use the same loophole to drain the DAO funds into a
secure account. Eventually it turned out that the code is mutable, stoppable, and refutable after all, and the Ethereum community decided to do a “hard fork” (also known as a “bailout”) to revert history to before this transaction. Some elements of the community strongly opposed this decision, and so an alternative currency called Ethereum Classic was created that preserved the original history.

9.2 REGULAR EXPRESSIONS

Searching for a piece of text is a common task in computing. At its heart, the search problem is quite simple. The system has a collection $X = \{x_0, \ldots, x_k\}$ of strings (for example filenames on a hard-drive, or names of students inside a database), and the user wants to find out the subset of all the $x \in X$ that are matched by some pattern. (For example all files that end with the string `.txt`.) In full generality, we could allow the user to specify the pattern by giving out an arbitrary function $F : \{0, 1\}^* \rightarrow \{0, 1\}$, where $F(x) = 1$ corresponds to the pattern matching $x$. For example, the user could give a program $P$ in some Turing-complete programming language such as Python, and the system will return all the $x_i$’s such that $P(x_i) = 1$. However, we don’t want our system to get into an infinite loop just trying to evaluate this function!

Because the Halting problem for Turing-complete computational models is uncomputable, a system would not be able to verify that a given program $P$ does not halt. For this reason, typical systems for searching files or databases do not allow users to specify functions in full-fledged programming languages. Rather, they use restricted computational models that are rich enough to capture many of the queries needed in practice (e.g., all filenames ending with `.txt`, or all phone numbers of the form (xxx) xxx-xxxx inside a textfile), but restricted enough so that they cannot result in an finite loop. One of the most popular models for this application is regular expressions. You have probably come across regular expressions if you ever used an advanced text editor, a command line shell, or have done any kind of manipulations of text files.

A regular expression over some alphabet $\Sigma$ is obtained by combining elements of $\Sigma$ with the operation of concatenation, as well as $|$ (corresponding to or) and $\ast$ (corresponding to repetition zero or more times).¹ For example, the following regular expression over the alphabet $\{0, 1\}$ corresponds to the set of all even length strings $x \in \{0, 1\}^*$ where the digit at location $2i$ is the same as the one at location $2i + 1$

¹ Common implementations of regular expressions in programming languages and shells typically include some extra operations on top of $|$ and $\ast$, but these can all be implemented as “syntactic sugar” using the operators $|$ and $\ast$. 
We have seen a recursive definition before in the setting of $\lambda$ expressions (Definition 7.4). Just like recursive functions, we can define a concept recursively. A definition of some class $\mathcal{C}$ of objects can be thought of as defining a function that maps an object $\sigma$ to either $\text{false}$ or $\text{true}$ depending on whether $\sigma \in \mathcal{C}$. Thus we can think of Definition 9.2 as defining a recursive function that maps a string $r$ over $\Sigma \cup \{\text{'(}', \text{')}', \text{'|'}, \text{'*'}, \text{'}\} \cup \emptyset$ and $\text{''}$ depending on whether $r$ describes a valid regular expression.

These are the regular expressions corresponding to accepting no strings, and accepting only the empty string respectively.

For every $i$:

\[(00|11)^*\]  

\[(a|b|c|d)(a|b|c|d)^*(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*\]  

Formally, regular expressions are defined by the following recursive definition:

**Definition 9.2 — Regular expression.** A regular expression $\mathcal{E}$ over an alphabet $\Sigma$ is a string over $\Sigma \cup \{\text{'(}', \text{')}', \text{'|'}, \text{'*'}, \text{'}\} \cup \emptyset$ that has one of the following forms:

1. $\mathcal{E} = \sigma$ where $\sigma \in \Sigma$
2. $\mathcal{E} = (\mathcal{E'})(\mathcal{E''})$ where $\mathcal{E'}, \mathcal{E''}$ are regular expressions.
3. $\mathcal{E} = (\mathcal{E'})(\mathcal{E''})$ where $\mathcal{E'}, \mathcal{E''}$ are regular expressions. (We often drop the parenthesis when there is no danger of confusion and so write this as $\mathcal{E} \mathcal{E'}$.)
4. $\mathcal{E} = (\mathcal{E'})^*$ where $\mathcal{E'}$ is a regular expression.

Finally we also allow the following “edge cases”: $\mathcal{E} = \emptyset$ and $\mathcal{E} = \text{''}$.  

Every regular expression $\mathcal{E}$ corresponds to a function $\Phi_\mathcal{E} : \Sigma^* \to \{0, 1\}$ where $\Phi_\mathcal{E}(x) = 1$ if $x$ matches the regular expression. The definition of “matching” is recursive as well. For example, if $\mathcal{E}$ and $\mathcal{E'}$ match the strings $x$ and $x'$, then the expression $\mathcal{E} \mathcal{E'}$ matches the concatenated string $xx'$. Similarly, if $\mathcal{E} = (00|11)^*$ then $\Phi_\mathcal{E}(x) = 1$ if and only if $x$ is of even length and $x_{2i} = x_{2i+1}$ for every $i < |x|/2$. We now turn to the formal definition of $\Phi_\mathcal{E}$.

The formal definition of $\Phi_\mathcal{E}$ is one of those definitions that is more cumbersome to write than to grasp. Thus it might be easier for you to first work it out on your own and then check that your definition matches what is written below.
We use function notation in this book, but other texts often use the notion of languages, which are sets of strings. We say that a language \( \mathcal{L} \) is regular if and only if the corresponding function \( \Phi_{\mathcal{L}}(x) = 1 \) where \( \Phi_{\mathcal{L}} : \Sigma^* \rightarrow \{0,1\} \) is the function that outputs 1 on \( x \) iff \( x \in \mathcal{L} \).

Formally we should write \( \mathcal{L} = \{ \text{regex} \} \) instead of \( \text{regex} \) but for clarity we will use the convention that OR and concatenation are left-associative, and that we give precedence to \( \ast \), then concatenation, and then OR. Most of the time we will only explicitly write down the parenthesis that are not implied by these rules.

### Definition 9.3 — Matching a regular expression.

Let \( \text{exp} \) be a regular expression. Then the function \( \Phi_{\text{exp}} \) is defined as follows:

1. If \( \text{exp} = \sigma \) then \( \Phi_{\text{exp}}(x) = 1 \) iff \( x = \sigma \).

2. If \( \text{exp} = (\text{exp}'|\text{exp}'') \) then \( \Phi_{\text{exp}}(x) = \Phi_{\text{exp}'}(x) \lor \Phi_{\text{exp}''}(x) \) where \( \lor \) is the OR operator.

3. If \( \text{exp} = (\text{exp}'(\text{exp}'') \) then \( \Phi_{\text{exp}}(x) = 1 \) iff there is some \( x' \in \Sigma^* \) such that \( \text{exp} \) is the concatenation of \( x' \) and \( x'' \) and \( \Phi_{\text{exp}'}(x') = \Phi_{\text{exp}''}(x'') = 1 \).

4. If \( \text{exp} = (\text{exp}')^* \) then \( \Phi_{\text{exp}}(x) = 1 \) iff there are is \( k \in \mathbb{N} \) and some \( x_0, \ldots, x_{k-1} \in \Sigma^* \) such that \( x \) is the concatenation \( x_0 \cdots x_{k-1} \) and \( \Phi_{\text{exp}'}(x_i) = 1 \) for every \( i \in [k] \).

5. Finally, for the edge cases \( \Phi_{\emptyset} \) is the constant zero function, and \( \Phi_{\ast} \) is the function that only outputs 1 on the empty string "\( \ast \)".

We say that a regular expresion \( \text{exp} \) over \( \Sigma \) matches a string \( x \in \Sigma^* \) if \( \Phi_{\text{exp}}(x) = 1 \). We say that a function \( F : \Sigma^* \rightarrow \{0,1\} \) is regular if \( F = \Phi_{\text{exp}} \) for some regular expression \( \text{exp} \).

### Example 9.4 — A regular function.

Let \( \Sigma = \{a, b, c, d, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \) and \( F : \Sigma^* \rightarrow \{0,1\} \) be the function such that \( F(x) = 1 \) iff \( x \) consists of one or more of the letters \( a-d \) followed by a sequence of one or more digits (without a leading zero). As shown by Eq. (9.2), the function \( F \) is regular. Specifically, \( F = \Phi_{a|b|c|d(a|b|c|d)^\ast(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^\ast} \).

If we wanted to verify, for example, that \( \Phi_{a|b|c|d(a|b|c|d)^\ast(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^\ast} \) does output 1 on the string \( abc12078 \), we can do so by noticing that the expression \( a|b|c|d \) matches the string \( a \), \( a|b|c|d \)\(^\ast \) matches \( bc \), \( 0|1|2|3|4|5|6|7|8|9 \) matches the string \( 1 \), and the expression \( 0|1|2|3|4|5|6|7|8|9 \)\(^\ast \) matches the string \( 2078 \). Each one of those boils down to a simpler expression. For example, the expression \( a|b|c|d \)\(^\ast \) matches the string \( bc \) because both of the one-character strings \( b \) and \( c \) are matched by the expression \( a|b|c|d \).

The definitions above are not inherently difficult, but are a bit cumbersome. So you should pause here and go over it again until you understand why it corresponds to our intuitive notion of regular expresions. This is important not just for understanding regular expresions themselves (which are used time
We can think of regular expressions as a type of “programming language”. That is, we can think of a regular expression \( exp \) over the alphabet \( \Sigma \) as a program that computes the function \( \Phi_{exp}: \Sigma^* \rightarrow \{0,1\}. \) It turns out that this “regular expression programming language” is simple in the sense that for every regular expression \( exp \), we can compute the function \( \Phi_{exp} \) by a Turing Machine / NAND++ program that always halts:

**Theorem 9.5 — Regular expression always halt.** For every finite set \( \Sigma \) and \( exp \in (\Sigma \cup \{(,),|,\ast,\emptyset,"\})^* \), if \( exp \) is a valid regular expression over \( \Sigma \) then \( \Phi_{exp} \) is a total computable function from \( \Sigma^* \) to \( \{0,1\} \). That is, there is an always halting NAND++ program \( P_{exp} \) that computes \( \Phi_{exp}. \)

**Proof Idea:** The main idea behind the proof is to see that Definition 9.3 actually specifies a recursive algorithm for computing \( \Phi_{exp} \). Specifically, each one of our operations - concatenation, OR, and star - can be thought of as reducing the task of testing whether an expression \( exp \) matches a string \( x \) to testing whether some sub-expressions of \( exp \) match substrings of \( x \). Since these sub-expressions are always shorter than the original expression, this yields a recursive algorithm for checking if \( exp \) matches \( x \) which will eventually terminate at the base cases of the expressions that correspond to a single symbol or the empty string. The details are specified below.

**Proof of Theorem 9.5.** Definition 9.3 gives a way of recursively computing \( \Phi_{exp} \). The key observation is that in our recursive definition of regular expressions, whenever \( exp \) is made up of one or two expressions \( exp', exp'' \) then these two regular expressions are smaller than \( exp \), and eventually (when they have size 1) then they must correspond to the non-recursive case of a single alphabet symbol.

Therefore, we can prove the theorem by induction over the length \( m \) of \( exp \) (i.e., the number of symbols in the string \( exp \), also denoted as \( |exp| \)). For \( m = 1 \), \( exp \) is either a single alphabet symbol, "" or \( \emptyset \), and so computing the function \( \Phi_{exp} \) is straightforward. In the general case, for \( m = |exp| \) we assume by the induction hypothesis that we have proven the theorem for all expressions of length smaller than \( m \). Now, such an expression of length larger than one can obtained one of three cases using the OR, concatenation, or star operations. We now show that \( \Phi_{exp} \) will be computable in all these cases:

\[ \]
Case 1: \( \text{exp} = (\text{exp}' | \text{exp}'') \) where \( \text{exp}', \text{exp}'' \) are shorter regular expressions.

In this case by the inductive hypothesis we can compute \( \Phi_{\text{exp}'} \) and \( \Phi_{\text{exp}''} \) and so can compute \( \Phi_{\text{exp}}(x) = \Phi_{\text{exp}'}(x) \lor \Phi_{\text{exp}''}(x) \) (where \( \lor \) is the OR operator).

Case 2: \( \text{exp} = (\text{exp}')(\text{exp}'') \) where \( \text{exp}', \text{exp}'' \) are regular expressions.

In this case by the inductive hypothesis we can compute \( \Phi_{\text{exp}'} \) and \( \Phi_{\text{exp}''} \) and so can compute \( \Phi_{\text{exp}}(x) \) as

\[
\bigvee_{i=0}^{\left| x \right|-1} (\Phi_{\text{exp}'}(x_0 \cdots x_{i-1}) \land \Phi_{\text{exp}''}(x_i \cdots x_{\left| x \right|-1})) \quad (9.3)
\]

where \( \land \) is the AND operator and for \( i < j, x_j \cdots x_i \) refers to the empty string.

Case 3: \( \text{exp} = (\text{exp}')^* \) where \( \text{exp}' \) is a regular expression.

In this case by the inductive hypothesis we can compute \( \Phi_{\text{exp}'} \) and so we can compute \( \Phi_{\text{exp}}(x) \) by enumerating over all \( k \) from 1 to \( \left| x \right| \), and all ways to write \( x \) as the concatenation of \( k \) strings \( x_0 \cdots x_{k-1} \) (we can do so by enumerating over all possible \( k - 1 \) positions in which one string stops and the other begins). If for one of those partitions, \( \Phi_{\text{exp}'}(x_0) = \cdots = \Phi_{\text{exp}'}(x_{k-1}) = 1 \) then we output 1. Otherwise we output 0.

These three cases exhaust all the possibilities for an expression of length larger than one, and hence this completes the proof. \( \blacksquare \)

9.2.1 Efficient matching of regular expressions (advanced, optional)

The proof of Theorem 9.5 gives a recursive algorithm to evaluate whether a given string matches or not a regular expression. But it is not a very efficient algorithm.

However, it turns out that there is a much more efficient algorithm to match regular expressions. In particular, for every regular expression \( \text{exp} \) there is an algorithm that on input \( x \in \{0, 1\}^n \), computes \( \Phi_{\text{exp}}(x) \) in \( "O(n) \) running time".\(^8\) One way to obtain such an algorithm is to replace this recursive algorithm with dynamic programming, using the technique of memoization.\(^9\)

It turns out that the resulting dynamic program not only runs in \( O(n) \) time, but in fact uses only a constant amount of memory, and makes a single pass over its input. Such an algorithm is also known as a deterministic finite automaton (DFA). It is also known that every function that can be computed by a deterministic finite automaton is regular. The relation of regular expressions with finite automata is a beautiful topic, on which we only touch upon in this text. See books such as Sipser’s, Hopcroft, Motwani and Ullman, and Kozen’s.

We now prove the algorithmic result that regular expression match-
ing can be done by a linear (i.e., $O(n)$) time algorithm, and moreover one that uses a constant amount of memory, and makes a single pass over its input. Since we have not yet covered the topics of time and space complexity, we will describe the algorithm in high level terms, without making the computational model precise. In Chapter 16 we will define space complexity formally, and prove the equivalence of deterministic finite automata and regular expression (see also Theorem 9.7 below), which will also imply Theorem 9.6.

**Theorem 9.6 — DFA for regular expression matching.** Let $exp$ be a regular expression. Then there is an $O(n)$ time algorithm that computes $\Phi_{exp}$.

Moreover, this algorithm only makes a single pass over the input, and utilizes only a constant amount of working memory. That is, it is a deterministic finite automaton.

We note that this theorem is very interesting even if one ignores the part following the “moreover”. Hence, the reader is very welcome to ignore this part in the first pass over the theorem and its proof.

**Proof Idea:** The idea is to first obtain a more efficient recursive algorithm for computing $\Phi_{exp}$ and then turning this recursive algorithm into a constant-space single-pass algorithm using the technique of memoization. In this technique we record in a table the results of every call to a function, and then if we make future calls with the same input, we retrieve the result from the table instead of re-computing it. This simple optimization can sometimes result in huge savings in running time.

In this case, we can define a recursive algorithm that on input a regular expression $exp$ and a string $x \in \{0, 1\}^n$, computes $\Phi_{exp}(x)$ as follows:

- If $n = 0$ (i.e., $x$ is the empty string) then we output 1 iff $exp$ contains "$".
- If $n > 0$, we let $\sigma = x_{n-1}$ and let $exp' = exp[\sigma]$ to be the regular expression that matches a string $x$ iff $exp$ matches the string $x\sigma$. (It can be shown that such a regular expression $exp'$ exists and is in fact of equal or smaller “complexity” to $exp$ for some appropriate notion of complexity.) We use a recursive call to return $\Phi_{exp'}(x_0 \cdots x_{n-1})$.

The running time of this recursive algorithm can be computed by the formula $T(n) = T(n - 1) + O(1)$ which solves to $O(n)$ (where the constant in the running time can depend on the length of the regular expression $exp$).
If we want to get the stronger result of a constant space algorithm (i.e., DFA) then we can use memoization. Specifically, we will store a table of the (constantly many) expressions of length at most \(|exp|\) that we need to deal with in the course of this algorithm, and iteratively for \(i = 0, 1, \ldots, n - 1\), compute whether or not each one of those expressions matches \(x_0 \cdots x_{i-1}\). ∗

**Proof of Theorem 9.6.** The central definition for this proof is the notion of a *restriction* of a regular expression. For a regular expression \(exp\) over an alphabet \(Σ\) and symbol \(σ \in Σ\), we will define \(exp[σ]\) to be a regular expression such that \(exp[σ]\) matches a string \(x\) if and only if \(exp\) matches the string \(xσ\). For example, if \(exp\) is the regular expression \(01|01)∗(01)\) (i.e., one or more occurrences of \(01\)) then \(exp[1]\) will be \(0|01)∗0\) and \(exp[0]\) will be \(∅\).

Given an expression \(exp\) and \(σ \in \{0, 1\}\), we can compute \(exp[σ]\) recursively as follows:

1. If \(exp = τ\) for \(τ \in Σ\) then \(exp[σ] = ''\) if \(τ = σ\) and \(exp[σ] = ∅\) otherwise.
2. If \(exp = exp'|exp''\) then \(exp[σ] = exp'|exp''[σ]\).
3. If \(exp = exp' exp''\) then \(exp[σ] = exp' exp''[σ]\) if \(exp''\) can not match the empty string. Otherwise, \(exp[σ] = exp' exp''[σ]|exp'[σ]\)
4. If \(exp = (exp')^\ast\) then \(exp[σ] = (exp')^\ast(exp'[σ])\).
5. If \(exp = ''\) or \(exp = ∅\) then \(exp[σ] = ∅\).

We leave it as an exercise to prove the following claim: (which can be shown by induction following the recursive definition of \(Φ_{exp}\))

**Claim:** For every \(x \in \{0, 1\}^\ast\), \(Φ_{exp}(xσ) = 1\) if and only if \(Φ_{exp[σ]}(x) = 1\)

The claim above suggests the following algorithm:

**A recursive linear time algorithm for regular expression matching:** We can now define a recursive algorithm for computing \(Φ_{exp} :\)

**Algorithm** \(MATCH(exp, x)\):

**Inputs:** \(exp\) is normal form regular expression, \(x \in Σ^\ast\) for some \(n \in \mathbb{N}\).

1. If \(x = ''\) then return 1 if \(exp\) has the form \(exp = ''|exp'\) for some \(exp'\). (For a normal-form expression, this is the only way it matches the empty string.)
2. Otherwise, return \(MATCH(exp[x_{n-1}], x_0 \cdots x_{n-1})\).
Algorithm MATCH is a recursive algorithm that on input an expression $exp$ and a string $x \in \{0, 1\}^n$, does some constant time computation and then calls itself on input some expression $exp'$ and a string $x$ of length $n - 1$. It will terminate after $n$ steps when it reaches a string of length 0.

There is one subtle issue and that is that to bound the running time, we need to show that if we let $exp_0$ be the regular expression that this algorithm obtains at step $i$, then $exp$ does not become itself much larger than the original expression $exp$. If $exp$ is a regular expression, then for every $n \in \mathbb{N}$ and string $\alpha \in \{0, 1\}^n$, we will denote by $exp[\alpha]$ the expression $((exp[\alpha_0])[\alpha_1]) \cdots [\alpha_{n-1}]$. That is, $exp[\alpha]$ is the expression obtained by considering the restriction $exp_0 = exp[\alpha_0]$, and then considering the restriction $x_1 = exp_0[\alpha_1]$ and so on and so forth. We can also think of $exp[\alpha]$ as the regular expression that matches $x$ if and only if $exp$ matches $x\alpha_{n-1}\alpha_{n-2} \cdots \alpha_0$.

The expressions considered by Algorithm MATCH all have the form $exp[\alpha]$ for some string $\alpha$ where $exp$ is the original input expression. Thus the following claim will help us bound our algorithms complexity.¹¹

Claim: For every regular expression $exp$, the set $S(exp) = \{exp[\alpha]|\alpha \in \{0, 1\}^*\}$ is finite.

Proof of claim: We prove this by induction on the structure of $exp$. If $exp$ is a symbol, the empty string, or the empty set, then this is straightforward to show as the most expressions $S(exp)$ can contain are the expression itself, $\epsilon$, and $\emptyset$. Otherwise we split to the two cases (i) $exp = exp''$ and (ii) $exp = exp'exp''$, where $exp', exp''$ are smaller expressions (and hence by the induction hypothesis $S(exp')$ and $S(exp'')$ are finite). In the case (i), if $exp = (exp')^*$ then $exp[\alpha]$ is either equal to $(exp'[\alpha])^*$ or it is simply the empty set if $exp'[\alpha] = \emptyset$. Since $exp'[\alpha]$ is in the set $S(exp')$, the number of distinct expressions in $S(exp)$ is at most $|S(exp')| + 1$. In the case (ii), if $exp = exp'exp''$ then all the restrictions of $exp$ to strings $\alpha$ will either have the form $exp'exp''[\alpha]$ or the form $exp'exp''[\alpha]exp'[\alpha']$ where $\alpha'$ is some string such that $\alpha = \alpha'\alpha''$ and $exp[\alpha'']$ matches the empty string. Since $exp''[\alpha] \in S(exp'')$ and $exp'[\alpha'] \in S(exp')$, the number of the possible distinct expressions of the form $exp[\alpha]$ is at most $|S(exp'')| + |S(exp'')| \cdot |S(exp')|$. This completes the proof of the claim.

The bottom line is that while running our algorithm on a regular expression $exp$, all the expressions we will ever encounter will be in the finite set $S(exp)$, no matter how large the input $x$ is. Therefore, the running time of MATCH is $O(n)$ where the implicit constant in the $O$ notation can (and will) depend on $exp$ but crucially, not on the length of the input $x$.

Proving the “moreover” part: At this point, we have already

¹¹This claim is strongly related to the Myhill-Nerode Theorem. One direction of this theorem can be thought of as saying that if $exp$ is a regular expression then there is at most a finite number of strings $z_0, \ldots, z_k$ such that $\Phi_{exp[i]} \neq \Phi_{exp[j]}$ for every $0 \leq i \neq j < k$. 
proven a highly non-trivial statement: the existence of a linear-time algorithm for matching regular expressions. The reader may well be content with this, and stop reading the proof at this point. However, as mentioned above, we can do even more and in fact have a *constant space* algorithm for this. To do so, we will turn our recursive algorithm into an iterative *dynamic program*. Specifically, we replace our recursive algorithm $MATCH$ with the following iterative algorithm $MATCH'$:

### Algorithm $MATCH'(exp, x)$:

**Inputs:** $exp$ is normal form regular expression, $x \in \Sigma^n$ for some $n \in \mathbb{N}$.

**Operation:**

1. Let $S = S(exp)$. Note that this is a finite set, and by its definition, for every $exp' \in S$ and $\sigma \in \{0, 1\}$, $exp'[\sigma]$ is in $S$ as well.
2. Define a Boolean variable $v_{exp'}$ for every $exp' \in S$. Initially we set $v_{exp'} = 1$ if and only if $exp'$ matches the empty string.
3. For $i = 0, \ldots, n - 1$ do the following:
   (a) *Copy the variables* $\{v_{exp'}\}$ *to temporary variables*: For every $exp' \in S$, we set $\text{temp}_{exp'} = v_{exp'}$.
   (b) *Update the variables* $\{v_{exp'}\}$ *based on the* $i$*-th bit of* $x$: Let $\sigma = x_i$ and set $v_{exp'} = \text{temp}_{exp'}[\sigma]$ for every $exp' \in S$.
4. Output $v_{exp}$.

Algorithm $MATCH'$ maintains the invariant that at the end of step $i$, for every $exp' \in S$, the variable $v_{exp'}$ is equal if and only if $exp'$ matches the string $x_0 \cdots x_{i-1}$. In particular, at the very end, $v_{exp}$ is equal to 1 if and only if $exp$ matches the full string $x_0 \cdots x_{n-1}$. Note that $MATCH'$ only maintains a constant number of variables (as $S$ is finite), and that it proceeds in one linear scan over the input, and so this proves the theorem.

### 9.2.2 Equivalence of DFA’s and regular expressions (optional)

Surprisingly, regular expressions and constant-space algorithms turn out to be *equivalent* in power. That is, the following theorem is known

**Theorem 9.7** — Regular expressions are equivalent to constant-space algorithms. Let $\Sigma$ be a finite set and $F : \Sigma^n \to \{0, 1\}$. Then $F$ is regular if and only if there exists a $O(1)$-space algorithm to compute $F$. Moreover, if $F$ can be computed by a $O(1)$-space algorithm, then
it can also be computed by such an algorithm that makes a single pass over its input, i.e., a deterministic finite automaton.

One direction of Theorem 9.7 (namely that if \( F \) is regular then it is computable by a constant-space one-pass algorithm) follows from Theorem 9.6. The other direction can be shown using similar ideas. We defer the full proof of Theorem 9.7 to Chapter 16, where we will formally define space complexity. However, we do state here an important corollary:

Lemma 9.8 — Regular expressions closed under complement. If \( F : \Sigma^* \to \{0, 1\} \) is regular then so is the function \( \overline{F} \), where \( \overline{F}(x) = 1 - F(x) \) for every \( x \in \Sigma^* \).

Proof. If \( F \) is regular then by Theorem 9.6 it can be computed by a constant-space algorithm \( \mathcal{A} \). But then the algorithm \( \overline{\mathcal{A}} \) which does the same computation and outputs the negation of the output of \( \mathcal{A} \) also utilizes constant space and computes \( \overline{F} \). By Theorem 9.7 this implies that \( \overline{F} \) is regular as well. ■

9.3 LIMITATIONS OF REGULAR EXPRESSIONS

The fact that functions computed by regular expressions always halt is of course one of the reasons why they are so useful. When you make a regular expression search, you are guaranteed that you will get a result. This is why operating systems, for example, restrict you for searching a file via regular expressions and don’t allow searching by specifying an arbitrary function via a general-purpose programming language. But this always-halting property comes at a cost. Regular expressions cannot compute every function that is computable by NAND++ programs. In fact there are some very simple (and useful!) functions that they cannot compute, such as the following:

Lemma 9.9 — Matching parenthesis. Let \( \Sigma = \{\langle, \rangle\} \) and \( MATCHPAREN : \Sigma^* \to \{0, 1\} \) be the function that given a string of parenthesis, outputs 1 if and only if every opening parenthesis is matched by a corresponding closed one. Then there is no regular expression over \( \Sigma \) that computes \( MATCHPAREN \).

Lemma 9.9 is a consequence of the following result known as the pumping lemma:

Theorem 9.10 — Pumping Lemma. Let \( \text{exp} \) be a regular expression. Then there is some number \( n_0 \) such that for every \( w \in \{0, 1\}^* \) with \(|w| > n_0\) and \( \Phi_{\text{exp}}(w) = 1 \), it holds that we can write \( w = xyz \) where \(|y| \geq 1\), \(|xy| \leq n_0\) and such that \( \Phi_{\text{exp}}(xy^kz) = 1 \) for every \( k \in \mathbb{N} \).
To prove the “pumping lemma” we look at a word $w$ that is much larger than the regular expression $exp$ that matches it. In such a case, part of $w$ must be matched by some sub-expression of the form $(exp')^*$, since this is the only operator that allows matching words longer than the expression. If we look at the “leftmost” such sub-expression and define $y^k$ to be the string that is matched by it, we obtain the partition needed for the pumping lemma.
**Proof Idea:** The idea behind the proof is very simple (see Fig. 9.1). If we let $n_0$ be, say, twice the number of symbols that are used in the expression $\text{exp}$, then the only way that there is some $w$ with $|w| > n_0$ and $\Phi_{\text{exp}}(w) = 1$ is that $\text{exp}$ contains the * (i.e. star) operator and that there is a nonempty substring $y$ of $w$ that was matched by $(\text{exp}')^*$ for some sub-expression $\text{exp}'$ of $\text{exp}$. We can now repeat $y$ any number of times and still get a matching string. *

The pumping lemma is a bit cumbersome to state, but one way to remember it is that it simply says the following: “if a string matching a regular expression is long enough, one of its substrings must be matched using the * operator”.

**Proof of Theorem 9.10.** To prove the lemma formally, we use induction on the length of the expression. Like all induction proofs, this is going to be somewhat lengthy, but at the end of the day it directly follows the intuition above that somewhere we must have used the star operation. Reading this proof, and in particular understanding how the formal proof below corresponds to the intuitive idea above, is a very good way to get more comfort with inductive proofs of this form.

Our inductive hypothesis is that for an $n$ length expression, $n_0 = 2n$ satisfies the conditions of the lemma. The base case is when the expression is a single symbol or that it is $\emptyset$ or "" in which case the condition is satisfied just because there is no matching string of length more than one. Otherwise, $\text{exp}$ is of the form (a) $\text{exp}'\text{exp}''$, (b), $(\text{exp}')(\text{exp}'')$, (c) or $(\text{exp}')^*$ where in all these cases the subexpressions have fewer symbols than $\text{exp}$ and hence satisfy the induction hypothesis.

In case (a), every string $w$ matching $\text{exp}$ must match either $\text{exp}'$ or $\text{exp}''$. In the former case, since $\text{exp}'$ satisfies the induction hypothesis, if $|w| > n_0$ then we can write $w = xyz$ such that $xy^kz$ matches $\text{exp}'$ for every $k$, and hence this is matched by $\text{exp}$ as well.

In case (b), if $w$ matches $(\text{exp}')(\text{exp}'')$, then we can write $w = w'w''$ where $w'$ matches $\text{exp}'$ and $w''$ matches $\text{exp}''$. Again we split to subcases. If $|w'| > 2|\text{exp}'|$, then by the induction hypothesis we can write $w' = xyz$ of the form above such that $xy^kz$ matches $\text{exp}'$ for every $k$ and then $xy^kzw''$ matches $(\text{exp}')(\text{exp}'')$. This completes the proof since $|xy| \leq 2|\text{exp}'|$ and so in particular $|xy| \leq 2(|\text{exp}'| + |\text{exp}''|) \leq 2|\text{exp}|$, and hence $zw''$ can be play the role of $z$ in the proof. Otherwise, if $|w'| \leq 2|\text{exp}'|$ then since $|w|$ is larger than $2|\text{exp}|$ and $w = w'w''$ and $\text{exp} = \text{exp}'\text{exp}''$, we get that $|w'| + |w''| > 2(|\text{exp}'| + |\text{exp}''|)$. Thus, if $|w'| \leq 2|\text{exp}'|$ it must be that $|w''| > 2|\text{exp}'|$, and hence by the induction hypothesis we can write $w'' = xyz$ such that $xy^kz$ matches $\text{exp}''$ for every $k$ and $|xy| \leq 2|\text{exp}'|$. Therefore we get that
w'x y^k z matches \((exp')(exp'')\) for every \(k\) and since \(|w'| \leq 2|exp'|\), 
\(|w'x| \leq 2(|exp'| + |exp'|)\) and this completes the proof since \(w'x\) can play the role of \(x\) in the statement.

Now in the case (c), if \(w\) matches \((exp')^*\) then \(w = w_0 \ldots w_i\) where \(w_i\) is a nonempty string that matches \(exp'\) for every \(i\). If \(|w_0| > 2|exp'|\) then we can use the same approach as in the concatenation case above. Otherwise, we simply note that if \(x\) is the empty string, \(y = w_0\), and \(z = w_1 \ldots w_i\) then \(xy^k z\) will match \((exp')^*\) for every \(k\).

**Recursive definitions and inductive proofs** When an object is recursively defined (as in the case of regular expressions) then it is natural to prove properties of such objects by induction. That is, if we want to prove that all objects of this type have property \(P\), then it is natural to use an inductive steps that says that if \(o', o'', o'''\) etc have property \(P\) then so is an object \(o\) that is obtained by composing them.

Given the pumping lemma, we can easily prove **Lemma 9.9**:

**Proof of Lemma 9.9.** Suppose, towards the sake of contradiction, that there is an expression \(exp\) such that \(\Phi_{exp} = MATCH\ PAREN\). Let \(n_0\) be the number from Lemma 9.9 and let \(w = (n_0)^{n_0}\) (i.e., \(n_0\) left parenthesis followed by \(n_0\) right parenthesis). Then we see that if we write \(w = xyz\) as in Lemma 9.9, the condition \(|xy| \leq n_0\) implies that \(y\) consists solely of left parenthesis. Hence the string \(xy^2 z\) will contain more left parenthesis than right parenthesis. Hence \(MATCH\ PAREN (xy^2 z) = 0\) but by the pumping lemma \(\Phi_{exp}(xy^2 z) = 1\), contradicting our assumption that \(\Phi_{exp} = MATCH\ PAREN\).

The pumping lemma is a very useful tool to show that certain functions are not computable by a regular language. However, it is not an “if and only if” condition for regularity. There are non regular functions which still satisfy the conditions of the pumping lemma. To understand the pumping lemma, it is important to follow the order of quantifiers in Theorem 9.10. In particular, the number \(n_0\) in the statement of Theorem 9.10 depends on the regular expression (in particular we can choose \(n_0\) to be twice the number of symbols in the expression). So, if we want to use the pumping lemma to rule out the existence of a regular expression \(exp\) computing some function \(F\), we need to be able to choose an appropriate \(w\) that can be arbitrarily large and satisfies \(F(w) = 1\). This makes sense if you think about the intuition behind the pumping lemma: we need \(w\) to be large enough as to force the use of the star operator.
Figure 9.2: A cartoon of a proof using the pumping lemma that a function $F$ is not regular. The pumping lemma states that if $F$ is regular then there exists a number $n_0$ such that for every large enough $w$ with $F(w) = 1$, there exists a partition of $w$ to $w = xyz$ satisfying certain conditions such that for every $k \in \mathbb{N}$, $F(xy^kz) = 1$. You can imagine a pumping-lemma based proof as a game between you and the adversary. Every $\exists$ quantifier corresponds to an object you are free to choose on your own (and base your choice on previously chosen objects). Every $\forall$ quantifier corresponds to an object the adversary can choose arbitrarily (and again based on prior choices) as long as it satisfies the conditions. A valid proof corresponds to a strategy by which no matter what the adversary does, you can win the game by obtaining a contradiction which would be a choice of $k$ that would result in $F(xy^kz) = 0$, hence violating the conclusion of the pumping lemma.

Solved Exercise 9.1 — Palindromes is not regular. Prove that the following function over the alphabet $\{0, 1, \}$ is not regular: $PAL(w) = 1$ if and only if $w = u; u^R$ where $u \in \{0, 1\}^*$ and $u^R$ denotes $u$ “reversed”;

**Solution:** We use the pumping lemma. Suppose towards the sake of contradiction that there is a regular expression $exp$ computing $PAL$, and let $n_0$ be the number obtained by the pumping lemma (Theorem 9.10). Consider the string $w = 0^{n_0}; 0^{n_0}$. Since the reverse of the all zero string is the all zero string, $PAL(w) = 1$. Now, by the pumping lemma, if $PAL$ is computed by $exp$, then we can write $w = xyz$ such that $|xy| \leq n_0, |y| \geq 1$ and $PAL(xy^kz) = 1$ for every $k \in \mathbb{N}$. In particular, it must hold that $PAL(xz) = 1$, but this is a contradiction, since $xz = 0^{n_0}|y|, 0^{n_0}$ and so its two parts are not of the same length and in particular are not the reverse of one another.

For yet another example of a pumping-lemma based proof, see Fig. 9.2 which illustrates a cartoon of the proof of the non-regularity of the function $F : \{0, 1\}^* \rightarrow \{0, 1\}$ which is defined as $F(x) = 1$ iff $x = 0^n1^n$ for some $n \in \mathbb{N}$ (i.e., $x$ consists of a string of consecutive zeroes, followed by a string of consecutive ones of the same length).
9.4 OTHER SEMANTIC PROPERTIES OF REGULAR EXPRESSIONS

Regular expressions are widely used beyond just searching. First, they are typically used to define tokens in various formalisms such as programming data description languages. But they are also used beyond it. One nice example is the recent work on the NetKAT network programming language. In recent years, the world of networking moved from fixed topologies to “software defined networks”, that are run by programmable switches that can implement policies such as “if packet is SSL then forward it to A, otherwise forward it to B”. By its nature, one would want to use a formalism for such policies that is guaranteed to always halt (and quickly!) and that where it is possible to answer semantic questions such as “does C see the packets moved from A to B” etc. The NetKAT language uses a variant of regular expressions to achieve that.

Such applications use the fact that, due to their restrictions, we can solve not just the halting problem for them, but also answer several other semantic questions as well, all of whom would not be solvable for Turing complete models due to Rice’s Theorem (Theorem 8.7). For example, we can tell whether two regular expressions are equivalent, as well as whether a regular expression computes the constant zero function.

**Theorem 9.11** — Emptiness of regular languages is computable. There is an algorithm that given a regular expression \( \exp \), outputs 1 if and only if \( \Phi_{\exp} \) is the constant zero function.

**Proof Idea:** The idea is that we can directly observe this from the structure of the expression. The only way it will output the constant zero function is if it has the form \( \emptyset \) or is obtained by concatenating \( \emptyset \) with other expressions.

**Proof of Theorem 9.11.** Define a regular expression to be “empty” if it computes the constant zero function. The algorithm simply follows the following rules:

- If an expression has the form \( \sigma \) or "" then it is not empty.
- If \( \exp \) is not empty then \( \exp | \exp' \) is not empty for every \( \exp' \).
- If \( \exp \) is not empty then \( \exp^* \) is not empty.
- If \( \exp \) and \( \exp' \) are both not empty then \( \exp \exp' \) is not empty.
- \( \emptyset \) is empty.
• \( \sigma \) and "" are not empty.

Using these rules it is straightforward to come up with a recursive algorithm to determine emptiness. We leave verifying the details to the reader.

**Theorem 9.12** — Equivalence of regular expressions is computable.

There is an efficient algorithm that on input two regular expressions \( \mathsf{exp}, \mathsf{exp}' \), outputs 1 if and only if \( \Phi_{\mathsf{exp}} = \Phi_{\mathsf{exp}'} \).

*Proof.* Theorem 9.11 above is actually a special case of Theorem 9.12, since emptiness is the same as checking equivalence with the expression \( \emptyset \). However we prove Theorem 9.12 from Theorem 9.11. The idea is that given \( \mathsf{exp} \) and \( \mathsf{exp}' \), we will compute an expression \( \mathsf{exp}'' \) such that

\[
\Phi_{\mathsf{exp}''}(x) = (\Phi_{\mathsf{exp}}(x) \land \Phi_{\mathsf{exp}'}(x)) \lor (\overline{\Phi_{\mathsf{exp}}(x)} \land \overline{\Phi_{\mathsf{exp}'}(x)})
\]

(9.4)

(where \( \overline{y} \) denotes the negation of \( y \), i.e., \( \overline{y} = 1 - y \)). One can see that \( \mathsf{exp} \) is equivalent to \( \mathsf{exp}' \) if and only if \( \mathsf{exp}'' \) is empty. To construct this expression, we need to show how given expressions \( \mathsf{exp} \) and \( \mathsf{exp}' \), we can construct expressions \( \mathsf{exp} \land \mathsf{exp}' \) and \( \mathsf{exp} \lor \mathsf{exp}' \) that compute the functions \( \Phi_{\mathsf{exp}} \land \Phi_{\mathsf{exp}'} \) and \( \Phi_{\mathsf{exp}} \lor \Phi_{\mathsf{exp}'} \) respectively. (Computing the expression for \( \mathsf{exp} \lor \mathsf{exp}' \) is straightforward using the \( | \) operation of regular expressions.)

Specifically, by Lemma 9.8, regular functions are closed under negation, which means that for every regular expression \( \mathsf{exp} \) over the alphabet \( \Sigma \) there is an expression \( \overline{\mathsf{exp}} \) such that \( \Phi_{\overline{\mathsf{exp}}}(x) = 1 - \Phi_{\mathsf{exp}}(x) \) for every \( x \in \Sigma^* \). For every two expressions \( \mathsf{exp} \) and \( \mathsf{exp}' \) we can define \( \mathsf{exp} \lor \mathsf{exp}' \) to be simply the expression \( \mathsf{exp} | \mathsf{exp}' \) and \( \mathsf{exp} \land \mathsf{exp}' \) as \( \overline{\mathsf{exp} \lor \mathsf{exp}'} \). Now we can define

\[
\mathsf{exp}'' = (\mathsf{exp} \land \overline{\mathsf{exp}'}) \lor (\overline{\mathsf{exp}} \land \mathsf{exp}')
\]

and verify that \( \Phi_{\mathsf{exp}''} \) is the constant zero function if and only if \( \Phi_{\mathsf{exp}}(x) = \Phi_{\mathsf{exp}'}(x) \) for every \( x \in \Sigma^* \). Since by Theorem 9.11 we can verify emptiness of \( \mathsf{exp}'' \), we can also verify equivalence of \( \mathsf{exp} \) and \( \mathsf{exp}' \).

**9.5 CONTEXT FREE GRAMMARS**

If you have ever written a program, you’ve experienced a syntax error. You might also have had the experience of your program entering into an infinite loop. What is less likely is that the compiler or interpreter entered an infinite loop when trying to figure out if your program has a syntax error.

When a person designs a programming language, they need to come up with a function \( \mathsf{VALID} : \{0, 1\}^* \rightarrow \{0, 1\} \) that determines
For simplicity we drop the condition that the sequence does not have a leading zero, though it is not hard to encode it in a context-free grammar as well.

the strings that correspond to valid programs in this language. The compiler or interpreter computes \( \text{VALID} \) on the string corresponding to your source code to determine if there is a syntax error. To ensure that the compiler will always halt in this computation, language designers typically don’t use a general Turing-complete mechanism to express the function \( \text{VALID} \), but rather a restricted computational model. One of the most popular choices for such a model is context free grammar.

To explain context free grammars, let’s begin with a canonical example. Let us try to define a function \( \text{ARITH} : \Sigma^* \rightarrow \{0, 1\} \) that takes as input a string \( x \) over the alphabet \( \Sigma = \{(,), +, -, \times, \div, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \) and returns 1 if and only if the string \( x \) represents a valid arithmetic expression. Intuitively, we build expressions by applying an operation to smaller expressions, or enclosing them in parenthesis, where the “base case” corresponds to expressions that are simply numbers. A bit more precisely, we can make the following definitions:

- A **digit** is one of the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- A **number** is a sequence of digits.\(^\text{13}\)
- An **operation** is one of +, −, \( \times \), \( \div \)
- An **expression** has either the form “number” or the form “subexpression1 operation subexpression2” or “(subexpression)”.

A context free grammar (CFG) is a formal way of specifying such conditions. We can think of a CFG as a set of rules to generate valid expressions. In the example above, the rule \( \text{expression} \Rightarrow \text{expression} \times \text{expression} \) tells us that if we have built two valid expressions \( \text{exp1} \) and \( \text{exp2} \), then the expression \( \text{exp1} \times \text{exp2} \) is valid above.

We can divide our rules to “base rules” and “recursive rules”. The “base rules” are rules such as \( \text{number} \Rightarrow 0 \), \( \text{number} \Rightarrow 1 \), \( \text{number} \Rightarrow 2 \) and so on, that tell us that a single digit is a number. The “recursive rules” are rules such as \( \text{number} \Rightarrow \text{number} 0 \), \( \text{number} \Rightarrow \text{number} 1 \) and so on, that tell us that if we add a digit to a valid number then we still have a valid number. We now make the formal definition of context-free grammars:

**Definition 9.13 — Context Free Grammar.** Let \( \Sigma \) be some finite set. A **context free grammar (CFG)** over \( \Sigma \) is a triple \((V, R, s)\) where \( V \) is a set disjoint from \( \Sigma \) of variables, \( R \) is a set of rules, which are pairs \((v, z)\) (which we will write as \( v \Rightarrow z \)) where \( v \in V \) and \( z \in (\Sigma \cup V)^* \),
Example 9.14 — Context free grammar for arithmetic expressions.
The example above of well-formed arithmetic expressions can be captured formally by the following context free grammar:

- The alphabet $\Sigma$ is $\{(),+,−,×,÷,0,1,2,3,4,5,6,7,8,9\}$
- The variables are $V = \{expression, number, digit, operation\}$.
- The rules correspond the set $R$ containing the following pairs:
  - $operation \Rightarrow +, operation \Rightarrow −, operation \Rightarrow ×, operation \Rightarrow ÷$
  - $digit \Rightarrow 0,..., digit \Rightarrow 9$
  - $number \Rightarrow digit$
  - $number \Rightarrow digit number$
  - $expression \Rightarrow number$
  - $expression \Rightarrow expression operation expression$
  - $expression \Rightarrow (expression)$
- The starting variable is $expression$

There are various notations to write context free grammars in the literature, with one of the most common being Backus–Naur form where we write a rule of the form $v \Rightarrow a$ (where $v$ is a variable and $a$ is a string) in the form $<v> := a$. If we have several rules of the form $v \Rightarrow a, v \Rightarrow b$, and $v \Rightarrow c$ then we can combine them as $<v> := a|b|c$ (and this similarly extends for the case of more rules). For example, the Backus-Naur description for the context free grammar above is the following (using ASCII equivalents for operations):

- $operation := +|-|*|/$
- $digit := 0|1|2|3|4|5|6|7|8|9$
- $number := digit|digit number$
- $expression := number|expression operation expression$
- $expression := (expression)$

Another example of a context free grammar is the “matching parenthesis” grammar, which can be represented in Backus-Naur as follows:

- $match := ""|match match|(match)$

You can verify that a string over the alphabet $\{(,),\}$ can be generated from this grammar (where $match$ is the starting expression.
and "" corresponds to the empty string) if and only if it consists of a matching set of parenthesis.

### 9.5.1 Context-free grammars as a computational model

We can think of a CFG over the alphabet $\Sigma$ as defining a function that maps every string $x$ in $\Sigma^*$ to 1 or 0 depending on whether $x$ can be generated by the rules of the grammars. We now make this definition formally.

**Definition 9.15 — Deriving a string from a grammar.** If $G = (V, R, s)$ is a context-free grammar over $\Sigma$, then for two strings $\alpha, \beta \in (\Sigma \cup V)^*$ we say that $\beta$ can be derived in one step from $\alpha$, denoted by $\alpha \Rightarrow_G \beta$, if we can obtain $\beta$ from $\alpha$ by applying one of the rules of $G$. That is, we obtain $\beta$ by replacing in $\alpha$ one occurrence of the variable $v$ with the string $z$, where $v \Rightarrow z$ is a rule of $G$.

We say that $\beta$ can be derived from $\alpha$, denoted by $\alpha \Rightarrow^*_G \beta$, if it can be derived by some finite number $k$ of steps. That is, if there are $\alpha_1, ..., \alpha_k \in (\Sigma \cup V)^*$, so that $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \cdots \Rightarrow_G \alpha_k \Rightarrow_G \beta$.

We define the function computed by $(V, R, s)$ to be the map $\Phi_{V, R, s} : \Sigma^* \rightarrow \{0, 1\}$ such that $\Phi_{V, R, s}(x) = 1$ iff $s \Rightarrow^*_G x$.

We say that $F : \Sigma^* \rightarrow \{0, 1\}$ is context free if $F = \Phi_{V, R, s}$ for some CFG $(V, R, s)$.

A priori it might not be clear that the map $\Phi_{V, R, s}$ is computable, but it turns out that we can in fact compute it. That is, the “halting problem” for context free grammars is trivial.

**Theorem 9.16 — Context-free grammars always halt.** For every CFG $(V, R, s)$ over $\Sigma$, the function $\Phi_{V, R, s} : \Sigma^* \rightarrow \{0, 1\}$ is computable.

**Proof.** We only sketch the proof. It turns out that we can convert every CFG to an equivalent version that has the so called *Chomsky normal form*, where all rules either have the form $u \rightarrow vw$ for variables $u, v, w$ or the form $u \rightarrow \sigma$ for a variable $u$ and symbol $\sigma \in \Sigma$, plus potentially the rule $s \rightarrow "$ where $s$ is the starting variable. (The idea behind such a transformation is to simply add new variables as needed, and so for example we can translate a rule such as $v \rightarrow u\sigma w$ into the three rules $v \rightarrow ur, r \rightarrow tw$ and $t \rightarrow \sigma$.)

Using this form we get a natural recursive algorithm for computing whether $s \Rightarrow^*_G x$ for a given grammar $G$ and string $x$. We simply try all possible guesses for the first rule $s \rightarrow uv$ that is used in such a derivation, and then all possible ways to partition $x$ as a concatenation $x = x'x''$. If we guessed the rule and the partition correctly, then...
this reduces our task to checking whether $u \Rightarrow_G x'$ and $v \Rightarrow_G x''$, which (as it involves shorter strings) can be done recursively. The base cases are when $x$ is empty or a single symbol, and can be easily handled.

### Parse trees
While we present CFGs as merely deciding whether the syntax is correct or not, the algorithm to compute $\Phi_{V,R,s}$ actually gives more information than that. That is, on input a string $x$, if $\Phi_{V,R,s}(x) = 1$ then the algorithm yields the sequence of rules that one can apply from the starting vertex $s$ to obtain the final string $x$. We can think of these rules as determining a connected directed acyclic graph (i.e., a tree) with $s$ being a source (or root) vertex and the sinks (or leaves) corresponding to the substrings of $x$ that are obtained by the rules that do not have a variable in their second element. This tree is known as the parse tree of $x$, and often yields very useful information about the structure of $x$. Often the first step in a compiler or interpreter for a programming language is a parser that transforms the source into the parse tree (often known in this context as the abstract syntax tree). There are also tools that can automatically convert a description of a context-free grammars into a parser algorithm that computes the parse tree of a given string. (Indeed, the above recursive algorithm can be used to achieve this, but there are much more efficient versions, especially for grammars that have particular forms, and programming language designers often try to ensure their languages have these more efficient grammars.)

### 9.5.2 The power of context free grammars
While we can (and people do) talk about context free grammars over any alphabet $\Sigma$, in the following we will restrict ourselves to $\Sigma = \{0,1\}$. This is of course not a big restriction, as any finite alphabet $\Sigma$ can be encoded as strings of some finite size. It turns out that context free grammars can capture every regular expression:

**Theorem 9.17 — Context free grammars and regular expressions.** Let $exp$ be a regular expression over $\{0,1\}$, then there is a CFG $(V,R,s)$ over $\{0,1\}$ such that $\Phi_{V,R,s} = \Phi_{exp}$.

**Proof.** We will prove this by induction on the length of $exp$. If $exp$ is an expression of one bit length, then $exp = 0$ or $exp = 1$, in which case we leave it to the reader to verify that there is a (trivial) CFG that computes it. Otherwise, we fall into one of the following case: case
1: $exp = exp'exp''$, case 2: $exp = exp'|exp''$ or case 3: $exp = (exp')^*$ where in all cases $exp', exp''$ are shorter regular expressions. By the induction hypothesis have grammars $(V', R', s')$ and $(V'', R'', s'')$ that compute $\Phi_{exp'}$ and $\Phi_{exp''}$ respectively. By renaming of variables, we can also assume without loss of generality that $V'$ and $V''$ are disjoint.

In case 1, we can define the new grammar as follows: add a new starting variable $s \notin V \cup V'$ and the rule $s \rightarrow s's''$. In case 2, we can define the new grammar as follows: we add a new starting variable $s \notin V \cup V'$ and the rules $s \rightarrow s'$ and $s \rightarrow s''$. Case 3 will be the only one that uses recursion. As before we add a new starting variable $s \notin V \cup V'$, but now add the rules $s \rightarrow "$ (i.e., the empty string) and also add, for every rule of the form $(s', \alpha) \in R'$, the rule $s \rightarrow s\alpha$ to $R$.

We leave it to the reader as (again a very good!) exercise to verify that in all three cases the grammars we produce capture the same function as the original expression.

It turns out that CFG’s are strictly more powerful than regular expressions. In particular, as we’ve seen, the “matching parenthesis” function $MATCHPAREN$ can be computed by a context free grammar, whereas, as shown in Lemma 9.9, it cannot be computed by regular expressions. Here is another example:

**Solved Exercise 9.2 — Context free grammar for palindromes.** Let $PAL : \{0, 1, ;\}^* \rightarrow \{0, 1\}$ be the function defined in Solved Exercise 9.1 where $PAL(w) = 1$ iff $w$ has the form $u;u^R$. Then $PAL$ can be computed by a context-free grammar

**Solution:** A simple grammar computing $PAL$ can be described using Backus–Naur notation:

\[
\begin{align*}
\text{start} & : = ; \mid 0 \text{ start } 0 \mid 1 \text{ start } 1
\end{align*}
\]

One can prove by induction that this grammar generates exactly the strings $w$ such that $PAL(w) = 1$.

A more interesting example is computing the strings of the form $u;v$ that are not palindromes:

**Solved Exercise 9.3 — Non palindromes.** Prove that there is a context free grammar that computes $NPAL : \{0, 1, ;\}^* \rightarrow \{0, 1\}$ where $NPAL(w) = 1$ if $w = u;v$ but $v \neq u^R$.

**Solution:** Using Backus–Naur notation we can describe such a grammar as follows
palindrome := ; | 0 palindrome 0 | 1
\[=\] palindrome 1
different := 0 palindrome 1 | 1 palindrome
\[=\] 0
start := different | 0 start | 1 start
\[=\] start 0 | start 1

In words, this means that we can characterize a string \( w \) such that \( NPAL(w) = 1 \) as having the following form

\[ w = \alpha b u u^R b' \beta \] (9.5)

where \( \alpha, \beta, u \) are arbitrary strings and \( b \neq b' \). Hence we can generate such a string by first generating a palindrome \( u; u^R \) (palindrome variable), then adding either 0 on the right and 1 on the left to get something that is not a palindrome (different variable), and then we can add arbitrary number of 0’s and 1’s on either end (the start variable). ■

9.5.3 Limitations of context-free grammars (optional)

Even though context-free grammars are more powerful than regular expressions, there are some simple languages that are not captured by context free grammars. One tool to show this is the context-free grammar analog of the “pumping lemma” (Theorem 9.10):

**Theorem 9.18 — Context-free pumping lemma.** Let \((V, R, s)\) be a CFG over \( \Sigma \), then there is some \( n_0 \in \mathbb{N} \) such that for every \( x \in \Sigma^* \) with \( |x| > n_0 \) if \( \Phi_{V, R, s}(x) = 1 \) then \( x = abcd e \) such that \( |b| + |c| + |d| \leq n_1 \), \( |b| + |d| \geq 1 \), and \( \Phi_{V, R, s}(ab^kcd^k e) = 1 \) for every \( k \in \mathbb{N} \).

Proof of Theorem 9.18. We only sketch the proof. The idea is that if the total number of symbols in the rules \( R \) is \( k_0 \), then the only way to get \( |x| > k_0 \) with \( \Phi_{V, R, s}(x) = 1 \) is to use recursion. That is, there must be some variable \( v \in V \) such that we are able to derive from \( v \) the value \( bcd \) for some strings \( b, d \in \Sigma^* \), and then further on derive from \( v \) some string \( c \in \Sigma^* \) such that \( bcd \) is a substring of \( x \). If we try to take the minimal such \( v \), then we can ensure that \( |bcd| \) is at most some constant depending on \( k_0 \) and we can set \( n_0 \) to be that constant \( (n_0 = 10 \cdot |R| \cdot k_0) \).
will do, since we will not need more than $|R|$ applications of rules, and each such application can grow the string by at most $k_0$ symbols. Thus by the definition of the grammar, we can repeat the derivation to replace the substring $bcd$ in $x$ with $b^kcd^k$ for every $k \in \mathbb{N}$ while retaining the property that the output of $\Phi_{V,R,s}$ is still one. ■

Using Theorem 9.18 one can show that even the simple function $F(x) = 1$ iff $x = ww$ for some $w \in \{0,1\}^*$ is not context free. (In contrast, the function $F(x) = 1$ iff $x = wu^R$ for $w \in \{0,1\}^*$ where for $w \in \{0,1\}^n$, $w^R = w_{n-1}w_{n-2} \cdots w_0$ is context free, can you see why?)

**Solved Exercise 9.4 — Equality is not context-free.** Let $EQ : \{0,1,;\}^* \to \{0,1\}$ be the function such that $F(x) = 1$ if and only if $x = u;u$ for some $u \in \{0,1\}^*$. Then $EQ$ is not context free. ■

**Solution:** We use the context-free pumping lemma. Suppose towards the sake of contradiction that there is a grammar $G$ that computes $EQ$, and let $n_0$ be the constant obtained from Theorem 9.18. Consider the string $x = \text{abcede}$ as per Theorem 9.18, with $|bcd| \leq n_0$ and with $|b| + |d| \geq 1$. By Theorem 9.18, it should hold that $EQ(ace) = 1$. However, by case analysis this can be shown to be a contradiction. First of all, unless $b$ is on the left side of the ; separator and $d$ is on the right side, dropping $b$ and $d$ will definitely make the two parts different. But if it is the case that $b$ is on the left side and $d$ is on the right side, then by the condition that $|bcd| \leq n_0$ we know that $b$ is a string of only zeros and $d$ is a string of only ones. If we drop $b$ and $d$ then since one of them is non empty, we get that there are either less zeroes on the left side than on the right side, or there are less ones on the right side than on the left side. In either case, we get that $EQ(ace) = 0$, obtaining the desired contradiction. ■

### 9.6 Semantic Properties of Context Free Languages

As in the case of regular expressions, the limitations of context free grammars do provide some advantages. For example, emptiness of context free grammars is decidable:

**Theorem 9.19 — Emptiness for CFG’s is decidable.** There is an algorithm that on input a context-free grammar $G$, outputs 1 if and only if $\Phi_G$ is the constant zero function.

**Proof Idea:** The proof is easier to see if we transform the grammar to Chomsky Normal Form as in Theorem 9.16. Given a grammar $G$, we can recursively define a non-terminal variable $v$ to be non empty if
there is either a rule of the form $v \Rightarrow \sigma$, or there is a rule of the form $v \Rightarrow uw$ where both $u$ and $w$ are non-empty. Then the grammar is non-empty if and only if the starting variable $s$ is non-empty. ★

**Proof of Theorem 9.19.** We assume that the grammar $G$ in Chomsky Normal Form as in Theorem 9.16. We consider the following procedure for marking variables as “non empty”:

1. We start by marking all variables $v$ that are involved in a rule of the form $v \Rightarrow \sigma$ as non empty.

2. We then continue to mark $v$ as non empty if it is involved in a rule of the form $v \Rightarrow \Sigma\Omega$ where $\Sigma, \Omega$ have been marked before.

We continue this way until we cannot mark any more variables. We then declare that the grammar is empty if and only if $s$ has not been marked. To see why this is a valid algorithm, note that if a variable $v$ has been marked as “non empty” then there is some string $\alpha \in \Sigma^*$ that can be derived from $v$. On the other hand, if $v$ has not been marked, then every sequence of derivations from $v$ will always have a variable that has not been replaced by alphabet symbols. Hence in particular $\Phi_G$ is the all zero function if and only if the starting variable $s$ is not marked “non empty”. ■

9.6.1 Uncomputability of context-free grammar equivalence (optional)

By analogy to regular expressions, one might have hoped to get an algorithm for deciding whether two given context free grammars are equivalent. Alas, no such luck. It turns out that the equivalence problem for context free grammars is uncomputable. This is a direct corollary of the following theorem:

**Theorem 9.20 — Fullness of CFG’s is uncomputable.** For every set $\Sigma$, let $CFGFULL_{\Sigma}$ be the function that on input a context-free grammar $G$ over $\Sigma$, outputs 1 if and only if $G$ computes the constant 1 function. Then there is some finite $\Sigma$ such that $CFGFULL_{\Sigma}$ is uncomputable.

**Theorem 9.20** immediately implies that equivalence for context-free grammars is uncomputable, since computing “fullness” of a grammar $G$ over some alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{k-1}\}$ corresponds to checking whether $G$ is equivalent to the grammar $s \Rightarrow ^*|s\sigma_0| \cdots |s\sigma_{k-1}$. Note that Theorem 9.20 and Theorem 9.19 together imply that context-free grammars, unlike regular expressions, are not closed under complement. (Can you see why?) Since we can encode every element of $\Sigma$
using $\lceil \log |\Sigma| \rceil$ bits (and this finite encoding can be easily carried out within a grammar) Theorem 9.20 implies that fullness is also uncomputable for grammars over the binary alphabet.

**Proof Idea:** We prove the theorem by reducing from the Halting problem. To do that we use the notion of configurations of NAND++ programs, as defined in Definition 7.12. Recall that a configuration of a program $P$ is a binary string $s$ that encodes all the information about the program in the current iteration.

We define $\Sigma$ to be $\{0,1\}$ plus some separator characters and define $\text{INVALID}_P : \Sigma^* \rightarrow \{0,1\}$ to be the function that maps every string $L \in \Sigma^*$ to 1 if and only if $L$ does not encode a sequence of configurations that correspond to a valid halting history of the computation of $P$ on the empty input.

The heart of the proof is to show that $\text{INVALID}_P$ is context-free. Once we do that, we see that $P$ halts on the empty input if and only if $\text{INVALID}_P(L) = 1$ for every $L$. To show that, we will encode the list in a special way that makes it amenable to deciding via a context-free grammar. Specifically we will reverse all the odd-numbered strings.

**Proof of Theorem 9.20.** We only sketch the proof. We will show that if we can compute $\text{CFGFULL}$ then we can solve $\text{HALTONZERO}$, which has been proven uncomputable in Theorem 8.4. Let $P$ be an input program for $\text{HALTONZERO}$. We will use the notion of configurations of a NAND++ program, as defined in Definition 7.12. Recall that a configuration of a NAND++ program $P$ and input $x$ captures the full state of $P$ (contents of all the variables) at some iteration of the computation. The particular details of configurations are not so important, but what you need to remember is that:

- A configuration can be encoded by a binary string $\sigma \in \{0,1\}^*$.
- The initial configuration of $P$ on the empty input is some fixed string.
- A halting configuration will have the value of the variable $\text{loop}$ (which can be easily “read off” from it) set to 1.
- If $\sigma$ is a configuration at some step $i$ of the computation, we denote by $\text{NEXT}_P(\sigma)$ as the configuration at the next step. $\text{NEXT}_P(\sigma)$ is a string that agrees with $\sigma$ on all but a constant number of coordinates (those encoding the position corresponding to the variable $i$ and the two adjacent ones). On those coordinates, the value of $\text{NEXT}_P(\sigma)$ can be computed by some finite function.

We will let the alphabet $\Sigma = \{0,1\} \cup \{\|,#\}$. A computation history of $P$ on the input 0 is a string $L \in \Sigma$ that corresponds to a list...
\[ \sigma_0 \# \sigma_1 \# \sigma_3 \# \cdots \sigma_{i-2} \# \sigma_{i-1} \# \] (i.e., \# comes before an even numbered block, and \# comes before an odd numbered one) such that if \( i \) is even then \( \sigma_i \) is the string encoding the configuration of \( P \) on input 0 at the beginning of its \( i \)-th iteration, and if \( i \) is odd then it is the same except the string is reversed. (That is, for odd \( i \), \( \text{rev}(\sigma_i) \) encodes the configuration of \( P \) on input 0 at the beginning of its \( i \)-th iteration.)\(^{15}\)

We now define \( \text{INVALID}_P : \Sigma^* \to \{0, 1\} \) as follows:

\[
\text{INVALID}_P(L) = \begin{cases} 
0 & L \text{ is a valid computation history of } P \text{ on } 0 \\
1 & \text{otherwise}
\end{cases} 
\]  

We will show the following claim:

**CLAIM:** \( \text{INVALID}_P \) is context-free.

The claim implies the theorem. Since \( P \) halts on 0 if and only if there exists a valid computation history, \( \text{INVALID}_P \) is the constant one function if and only if \( P \) does not halt on 0. In particular, this allows us to reduce determining whether \( P \) halts on 0 to determining whether the grammar \( G_P \), corresponding to \( \text{INVALID}_P \), is full.

We now turn to the proof of the claim. We will not show all the details, but the main point \( \text{INVALID}_P(L) = 1 \) if one of the following three conditions hold:

1. \( L \) is not of the right format, i.e. not of the form \( (\text{binary-string}) \# (\text{binary-string}) \# \cdots \).
2. \( L \) contains a substring of the form \# \( \sigma \# \sigma' \# \) such that \( \sigma' \neq \text{rev}(\text{NEXT}_P(\sigma)) \)
3. \( L \) contains a substring of the form \# \( \sigma \# \sigma' \# \) such that \( \sigma' \neq \text{NEXT}_P(\text{rev}(\sigma)) \)

Since context-free functions are closed under the OR operation, the claim will follow if we show that we can verify conditions 1, 2 and 3 via a context-free grammar. For condition 1 this is very simple: checking that \( L \) is of this format can be done using a regular expression, and since regular expressions are closed under negation, this means that checking that \( L \) is not of this format can also be done by a regular expression and hence by a context-free grammar.

For conditions 2 and 3, this follows via very similar reasoning to that showing that the function \( F \) such that \( F(u \# v) = 1 \) if and only if \( u \neq \text{rev}(v) \) is context-free, see Solved Exercise 9.3. After all, the \( \text{NEXT}_P \) function only modifies its input in a constant number of places. We leave filling out the details as an exercise to the reader. Since \( \text{INVALID}_P(L) = 1 \) if and only if \( L \) satisfies one of the conditions 1., 2. or 3., and all three conditions can be tested for via a context-free grammar, this completes the proof of the claim and hence the theorem. □
9.7 SUMMARY OF SEMANTIC PROPERTIES FOR REGULAR EXPRESSIONS AND CONTEXT-FREE GRAMMARS

To summarize, we can often trade expressiveness of the model for amenability to analysis. If we consider computational models that are not Turing complete, then we are sometimes able to bypass Rice’s Theorem and answer certain semantic questions about programs in such models. Here is a summary of some of what is known about semantic questions for the different models we have seen.

<table>
<thead>
<tr>
<th>Model</th>
<th>Halting</th>
<th>Emptiness</th>
<th>Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Expressions</td>
<td>Decidable</td>
<td>Decidable</td>
<td>Decidable</td>
</tr>
<tr>
<td>Context Free Grammars</td>
<td>Decidable</td>
<td>Decidable</td>
<td>Undecidable</td>
</tr>
<tr>
<td>Turing complete models</td>
<td>Undecidable</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>

Unrestricted Grammars (optional) The reason we call context free grammars “context free” is because if we have a rule of the form \( v \to a \) it means that we can always replace \( v \) with the string \( a \), no matter the context in which \( v \) appears. More generally, we might want to consider cases where our replacement rules depend on the context.

This gives rise to the notion of general grammars that allow rules of the form \( a \to b \) where both \( a \) and \( b \) are strings over \( (V \cup \Sigma)^* \). The idea is that if, for example, we wanted to enforce the condition that we only apply some rule such as \( v \to 0v1 \) when \( v \) is surrounded by three zeroes on both sides, then we could do so by adding a rule of the form \( 000v000 \to 000v1000 \) (and of course we can add much more general conditions). Alas, this generality comes at a cost - these general grammars are Turing complete and hence their halting problem is undecidable.

Lecture Recap

- The uncomputability of the Halting problem for general models motivates the definition of restricted computational models.
- In some restricted models we can answer semantic questions such as: does a given program terminate, or do two programs compute the same function?
- Regular expressions are a restricted model of computation that is often useful to capture tasks of string matching. We can test efficiently whether an expression matches a string, as well as answer questions such as Halting and Equivalence.
• Context free grammars is a stronger, yet still not Turing complete, model of computation. The halting problem for context free grammars is computable, but equivalence is not computable.

9.8 EXERCISES

Disclaimer  Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

9.9 BIBLIOGRAPHICAL NOTES

9.10 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

9.11 ACKNOWLEDGEMENTS

16 TODO: Add letter of Christopher Strachey to the editor of The Computer Journal. Explain right order of historical achievements. Talk about intuitionistic, logicist, and formalist approaches for the foundations of mathematics. Perhaps analogy to veganism. State the full Rice’s Theorem and say that it follows from the same proof as in the exercise.
10
Is every theorem provable?

“Take any definite unsolved problem, such as … the existence of an infinite number of prime numbers of the form $2^n + 1$. However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes…”

“…This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus.”, David Hilbert, 1900.

“The meaning of a statement is its method of verification.”, Moritz Schlick, 1938 (aka “The verification principle” of logical positivism)

The problems shown uncomputable in Chapter 8, while natural and important, still intimately involved NAND++ programs or other computing mechanisms in their definitions. One could perhaps hope that as long as we steer clear of functions whose inputs are themselves programs, we can avoid the “curse of uncomputability”. Alas, we have no such luck.

In this chapter we will see an example of a natural and seemingly “computation free” problem that nevertheless turns out to be uncomputable: solving Diophantine equations. As a corollary, we will see one of the most striking results of 20th century mathematics: Gödel’s Incompleteness Theorem, which showed that there are some mathematical statements (in fact, in number theory) that are inherently unprovable. We will actually start with the latter result, and then show the former.
The 1700’s and 1800’s were a time of great discoveries in mathematics but also of several crises. The discovery of calculus by Newton and Leibnitz in the late 1600’s ushered a golden age of problem solving. Many longstanding challenges succumbed to the new tools that were discovered, and mathematicians got ever better at doing some truly impressive calculations. However, the rigorous foundations behind these calculations left much to be desired. Mathematicians manipulated infinitesimal quantities and infinite series cavalierly, and while most of the time they ended up with the correct results, there were a few strange examples (such as trying to calculate the value of the infinite series $1 - 1 + 1 - 1 + 1 + \ldots$) which seemed to give out different answers depending on the method of calculation. This led to a growing sense of unease in the foundations of the subject which was addressed in works of mathematicians such as Cauchy, Weierstrass, and Riemann, who eventually placed analysis on firmer foundations, giving rise to the $\epsilon$’s and $\delta$’s that students taking honors calculus grapple with to this day.

In the beginning of the 20th century, there was an effort to replicate this effort, in greater rigor, to all parts of mathematics. The hope was to show that all the true results of mathematics can be obtained by starting with a number of axioms, and deriving theorems from them using logical rules of inference. This effort was known as the Hilbert program, named after the influential mathematician David Hilbert.

Alas, it turns out the results we’ve seen dealt a devastating blow to this program, as was shown by Kurt Gödel in 1931:

**Theorem 10.1** — Gödel’s Incompleteness Theorem: informal version.

For every sound proof system for sufficiently rich mathematical statements, there is a mathematical statement that is true but is not provable.

Before proving Theorem 10.2, we need to specify what does it mean to be “provable” (and even formally define the notion of a “mathematical statement”). Thus we need to define the notion of a proof system. In geometry and other areas of mathematics, proof systems are often
defined by starting with some basic assumptions or axioms and then
deriving more statements by using inference rules such as the famous
Modus Ponens, but what axioms shall we use? What rules?

Our idea will be to use an extremely general notion of proof, not
even restricting ourselves to ones that have the form of axioms and
inference. A proof will be simply a piece of text- a finite string- that
satisfies:

1. (effectiveness) Given a statement \( x \) and a proof \( w \) (both of which can
   be encoded as strings) we can verify that \( w \) is a valid proof for \( x \).
   (For example, by going line by line and checking that each line does
   indeed follow from the preceding ones using one of the allowed
   inference rules.)

2. (soundness) If there is a valid proof \( w \) for \( x \) then \( x \) is true.

Those seem like rather minimal requirements that one would want
from every proof system. Requirement 2 (soundness) is the very defi-
nition of a proof system: you shouldn’t be able to prove things that are
not true. Requirement 1 is also essential. If it there is no set of rules
(i.e., an algorithm) to check that a proof is valid then in what sense
is it a proof system? We could replace it with the system where the
“proof” for a statement \( x \) would simply be “trust me: it’s true”.

A mathematical statement will also simply be a string. Mathe-
matical statements states a fact about some mathematical object. For
example, the following is a mathematical statement:

\[
\text{“The number 2,696,635,869,504,783,333,238,805,675,613,588,278,597,832,162,617,892,474,670,798,113
is prime”.
}\]

(This happens to be a false statement; can you see why?)

Mathematical statements don’t have to be about numbers. They
can talk about any other mathematical object including sets, strings,
functions, graphs and yes, even programs. Thus, another example of a
mathematical statement is the following:

```python
def f(n):
    if n==1:
        return 1
    return f(3*n+1) if n % 2 else f(n//2)
```

(We actually don’t know if this statement is true or false.)

We start by considering statements of the second type. Our first
formalization of Theorem 10.2 will be the following
Theorem 10.2 — Gödel’s Incompleteness Theorem: computational variant. Let \( V : \{0,1\}^* \rightarrow \{0,1\} \) a computable purported verification procedure for mathematical statements of the form “Program \( P \) halts on the zero input” and “Program \( P \) does not halt on the zero input”. Then either:

- \( V \) is not sound: There exists a false statement \( x \) and a string \( w \in \{0,1\}^* \) such that \( V(x, w) = 1 \).

- \( V \) is not complete: There exists a true statement \( x \) such that for every \( w \in \{0,1\}^* \), \( V(x, w) = 0 \).

Proof Idea: If we had such a complete and sound proof system then we could solve the HALTONZERO problem. On input a program \( P \), we would search all purported proofs \( w \) and halt as soon as we find a proof of either “\( P \) halts on zero” or “\( P \) does not halt on zero”. If the system is sound and complete then we will eventually find such a proof, and it will provide us with the correct output.

Proof of Theorem 10.2. Assume for the sake of contradiction that there was such a proof system \( V \). We will use \( V \) to build an algorithm \( A \) that computes HALTONZERO, hence contradicting Theorem 8.4. Our algorithm \( A \) will will work as follows:

**Algorithm A:**

- **Input:** NAND++ program \( P \)
- **Goal:** Determine if \( P \) halts on the input 0.
- **Assumption:** We have access to a proof system \( V \) such that for every statement \( x \) of the form “Program \( Q \) halts on 0” or “Program \( Q \) does not halt on 0”, there exists some string \( w \in \{0,1\}^* \) such that \( V(x, w) = 1 \) if and only if \( x \) is true.

**Operation:**

- For \( n = 0, 1, 2, \ldots \):
  - For \( w \in \{0,1\}^n \):
    - If \( V(\text{"P halts on 0"}, w) = 1 \) output 1
    - If \( V(\text{"P does not halt on 0"}, w) = 1 \) output 0

If \( P \) halts on 0 then under our assumption there exists \( w \) that proves this fact, and so when Algorithm \( A \) reaches \( n = |w| \) we will eventu-
ally find this \( w \) and output 1, unless we already halted before. But we cannot halt before and output a wrong answer because it would contradict the soundness of the proof system. Similarly, this shows that if \( P \) does not halt on 0 then (since we assume there is a proof of this fact too) our algorithm \( A \) will eventually halt and output 0.

\[\]
Since \( x^* \) is true and does not have a proof in \( V \), this means that we cannot carry the above argument in the system \( V \), which means that \( V \) cannot prove its own soundness (or even consistency: that there is no proof of both a statement and its negation). Using this idea, it’s not hard to get Gödel’s second incompleteness theorem, which says that every sufficiently rich \( V \) cannot prove its own consistency. That is, if we formalize the statement \( c^* \) that is true if and only if \( V \) is consistent (i.e., \( V \) cannot prove both a statement and the statement’s negation), then \( c^* \) cannot be proven in \( V \).

10.2 QUANTIFIED INTEGER STATEMENTS

There is something “unsatisfying” about Theorem 10.2. Sure, it shows there are statements that are unprovable, but they don’t feel like “real” statements about math. After all, they talk about programs rather than numbers, matrices, or derivatives, or whatever it is they teach in math courses. It turns out that we can get an analogous result for statements such as “there are no integers \( x \) and \( y \) such that \( x^2 - 2 = y^7 \),” or “there are integers \( x, y, z \) such that \( x^2 + y^6 = z^{11} \)” that only talk about natural numbers. \(^1\) It doesn’t get much more “real math” than this. Indeed, the 19th century mathematician Leopold Kronecker famously said that “God made the integers, all else is the work of man.”

To make this more precise, let us define the notion of quantified integer statements:

**Definition 10.3 — Quantified integer statements.** A quantified integer statement is a well-formed statement with no unbound variables involving integers, variables, the operators \( >, <, \times, +, -, = \), the logical operations \( \neg \) (NOT), \( \wedge \) (AND), and \( \lor \) (OR), as well as quantifiers of the form \( \exists_{x \in \mathbb{N}} \) and \( \forall_{y \in \mathbb{N}} \) where \( x, y \) are variable names.

We often care deeply about determining the truth of quantified integer statements. For example, the statement that Fermat’s Last Theorem is true for \( n = 3 \) can be phrased as the quantified integer statement

\[ -\exists_{a \in \mathbb{N}} \exists_{b \in \mathbb{N}} \exists_{c \in \mathbb{N}} (a > 0) \land (b > 0) \land (c > 0) \land (a \times a \times a + b \times b \times b = c \times c \times c) . \]

(10.2)

The twin prime conjecture, that states that there is an infinite number of numbers \( p \) such that both \( p \) and \( p + 2 \) are primes can be phrased as the quantified integer statement

\[ \forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \land PRIME(p) \land PRIME(p + 2) \]  

(10.3)

\(^1\) I do not know if these statements are actually true or false, see here.
where we replace an instance of $PRIME(q)$ with the statement $(q > 1) \land \forall_{a \in \mathbb{N}} \forall_{b \in \mathbb{N}} (a = 1) \lor (a = q) \lor \neg(a \times b = q)$.

The claim (mentioned in Hilbert’s quote above) that are infinitely many primes of the form $p = 2^n + 1$ can be phrased as follows:

$$\forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \land PRIME(p) \land (\forall_{k \in \mathbb{N}} (k \neq 2 \land PRIME(k)) \Rightarrow \neg DIVIDES(k, p - 1))$$

(10.4)

where $DIVIDES(a, b)$ is the statement $\exists_{c \in \mathbb{N}} b \times c = a$. In English, this corresponds to the claim that for every $n$ there is some $p > n$ such that all of $p - 1$’s prime factors are equal to 2.

**Syntactic sugar for quantified integer statements**

To make our statements more readable, we often use syntactic sugar and so write $x \neq y$ as shorthand for $\neg(x = y)$, and so on. Similarly, the “implication operator” $a \Rightarrow b$ is “syntactic sugar” or shorthand for $\neg a \lor b$, and the “if and only if operator” $a \iff b$ is shorthand for $(a \Rightarrow b) \land (b \Rightarrow a)$. We will also allow ourselves the use of “macros”: plugging in one quantified integer statement in another, as we did with $DIVIDES$ and $PRIME$ above.

Much of number theory is concerned with determining the truth of quantified integer statements. Since our experience has been that, given enough time (which could sometimes be several centuries) humanity has managed to do so for the statements that it cared enough about, one could (as Hilbert did) hope that eventually we would be able to prove or disprove all such statements. Alas, this turns out to be impossible:

**Theorem 10.4 — Gödel’s Incompleteness Theorem for quantified integer statements.** Let $V : \{0, 1\}^* \rightarrow \{0, 1\}$ a computable purported verification procedure for quantified integer statements. Then either:

- **V is not sound:** There exists a false statement $x$ and a string $w \in \{0, 1\}^*$ such that $V(x, w) = 1$.

  or

- **V is not complete:** There exists a true statement $x$ such that for every $w \in \{0, 1\}^*, V(x, w) = 0$.

Theorem 10.4 is a direct corollary of the following result, just as Theorem 10.2 was a direct corollary of the uncomputability of $HALTONZERO$:
Since a quantified integer statement is simply a sequence of symbols, we can easily represent it as a string. We will assume that every string represents some quantified integer statement, by mapping strings that do not correspond to such a statement to an arbitrary statement such as $\exists \in \mathbb{N} = 1$.

This is a special case of what's known as “Fermat’s Last Theorem” which states that $a^n + b^n = c^n$ has no solution in integers for $n > 2$. This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case $n = 11$ (along with all other so called “regular prime exponents”) was established by Kummer in 1850.

In the rest of this chapter, we will show the proof of ??.

10.3 DIOPHANTINE EQUATIONS AND THE MRDP THEOREM

Many of the functions people wanted to compute over the years involved solving equations. These have a much longer history than mechanical computers. The Babylonians already knew how to solve some quadratic equations in 2000BC, and the formula for all quadratics appears in the Bakhshali Manuscript that was composed in India around the 3rd century. During the Renaissance, Italian mathematicians discovered generalization of these formulas for cubic and quartic (degrees 3 and 4) equations. Many of the greatest minds of the 17th and 18th century, including Euler, Lagrange, Leibniz and Gauss worked on the problem of finding such a formula for quintic equations to no avail, until in the 19th century Ruffini, Abel and Galois showed that no such formula exists, along the way giving birth to group theory.

However, the fact that there is no closed-form formula does not mean we can not solve such equations. People have been solving higher degree equations numerically for ages. The Chinese manuscript Jiuzhang Suanshu from the first century mentions such approaches. Solving polynomial equations is by no means restricted only to ancient history or to students’ homeworks. The gradient descent method is the workhorse powering many of the machine learning tools that have revolutionized Computer Science over the last several years.

But there are some equations that we simply do not know how to solve by any means. For example, it took more than 200 years until people succeeded in proving that the equation $a^{11} + b^{11} = c^{11}$ has no solution in integers. The notorious difficulty of so called Diophantine equations (i.e., finding integer roots of a polynomial) motivated the
As usual, we assume some standard way to express numbers and text as binary strings. The constant 100 is of course arbitrary; the problem is known to be uncomputable even for polynomials of degree four and at most 58 variables. In fact the number of variables can be reduced to nine, at the expense of the polynomial having a larger (but still constant) degree. See Jones’s paper for more about this issue.

Alas, this turned out not to be the case for Diophantine equations. In 1970, Yuri Matiyasevich, building on a decades long line of work by Martin Davis, Hilary Putnam and Julia Robinson, showed that there is simply no method to solve such equations in general:

**Theorem 10.6 — MRDP Theorem.** Let $\chi : \{0, 1\}^* \rightarrow \{0, 1\}$ be the function that takes as input a string describing a 100-variable polynomial with integer coefficients $P(x_0, \ldots, x_{99})$ and outputs 1 if and only if there exists $z_0, \ldots, z_{99} \in \mathbb{N}$ s.t. $P(z_0, \ldots, z_{99}) = 0$.

Then $\chi$ is uncomputable.  

The difficulty in finding a way to distinguish between “code” such as NAND++ programs, and “static content” such as polynomials is just another manifestation of the phenomenon that code is the same as data. While a fool-proof solution for distinguishing between the two is inherently impossible, finding heuristics that do a reasonable job keeps many firewall and anti-virus manufacturers very busy (and finding ways to bypass these tools keeps many hackers busy as well).

**10.4 HARDNESS OF QUANTIFIED INTEGER STATEMENTS**

We will not prove the MRDP Theorem (Theorem 10.6). However, as we mentioned, we will prove the uncomputability of QIS (i.e., Theorem 10.5), which is a special case of the MRDP Theorem. The reason is that a Diophantine equation is a special case of a quantified integer statement where the only quantifier is $\exists$. This means that deciding the truth of quantified integer statements is a potentially harder problem than solving Diophantine equations, and so it is potentially easier to prove that QIS is uncomputable.

If you find the last sentence confusing, it is worthwhile to reread it until you are sure you follow its
logic. We are so accustomed to trying to find solutions for problems that it can sometimes be hard to follow the arguments for showing that problems are uncomputable.

Our proof of the uncomputability of $QIS$ (i.e. Theorem 10.5) will, as usual, go by reduction from the Halting problem, but we will do so in two steps:

1. We will first use a reduction from the Halting problem to show that deciding the truth of quantified mixed statements is uncomputable. Unquantified mixed statements involve both strings and integers. Since quantified mixed statements are a more general concept than quantified integer statements, it is easier to prove the uncomputability of deciding their truth.

2. We will then reduce the problem of quantified mixed statements to quantifier integer statements.

10.4.1 Step 1: Quantified mixed statements and computation histories

We define quantified mixed statements as statements involving not just integers and the usual arithmetic operators, but also string variables as well.

**Definition 10.7 — Quantified mixed statements.** A quantified mixed statement is a well-formed statement with no unbound variables involving integers, variables, the operators $>, <, \times, +, -, =$, the logical operations $\neg$ (NOT), $\land$ (AND), and $\lor$ (OR), as well as quantifiers of the form $\exists x \in \mathbb{N}$, $\exists y \in \{0, 1\}^*$, $\forall z \in \mathbb{N}$, $\forall w \in \{0, 1\}^*$, where $x, y, a, b$ are variable names. These also include the operator $|a|$ which returns the length of a string valued variable $a$, as well as the operator $a_i$ where $a$ is a string-valued variable and $i$ is an integer valued expression which is true if $i$ is smaller than the length of $a$ and the $i^{th}$ coordinate of $a$ is 1, and is false otherwise.

For example, the true statement that for every string $a$ there is a string $b$ that corresponds to $a$ in reverse order can be phrased as the following quantified mixed statement

$$\forall a \in \{0, 1\}^* \exists b \in \{0, 1\}^* (|a| = |b|) \land (\forall i \in \mathbb{N}, i < |a| \Rightarrow (a_i \leftrightarrow b_{|a|-i}). \tag{10.5}$$

Quantified mixed statements are more general than quantified integer statements, and so the following theorem is potentially easier to prove than Theorem 10.5:
Theorem 10.8 — Uncomputability of quantified mixed statements. Let $QMS : \{0, 1\}^* \rightarrow \{0, 1\}$ be the function that given a (string representation of) a quantified mixed statement outputs 1 if it is true and 0 if it is false. Then $QMS$ is uncomputable.

Proof Idea: The idea behind the proof is similar to that used in showing that one-dimensional cellular automata are Turing complete (Theorem 7.11) as well as showing that equivalence (or even “fullness”) of context free grammars is uncomputable (Theorem 9.20). We use the notion of a configuration of a NAND++ program as in Definition 7.12. Such a configuration can be thought of as a string $\alpha$ over some large-but-finite alphabet $\Sigma$ describing its current state, including the values of all arrays, scalars, and the index variable $i$. It can be shown that if $\alpha$ is the configuration at a certain step of the execution and $\beta$ is the configuration at the next step, then $\beta_j = \alpha_j$ for all $j$ outside of $\{i-1, i, i+1\}$ where $i$ is the value of $i$. In particular, every value $\beta_j$ is simply a function of $\alpha_{j-1, i, j+1}$. Using these observations we can write a quantified mixed statement $NEXT(\alpha, \beta)$ that will be true if and only if $\beta$ is the configuration encoding the next step after $\alpha$. Since a program $P$ halts on input $x$ if and only if there is a sequence of configurations $\alpha_0, \ldots, \alpha_t$ (known as a computation history) starting with the initial configuration with input $x$ and ending in a halting configuration, we can define a quantified mixed statement to determine if there is such a statement by taking a universal quantifier over all strings $H$ (for history) that encode a tuple $(\alpha_0, \alpha_1, \ldots, \alpha_t)$ and then checking that $\alpha_0$ and $\alpha_t$ are valid starting and halting configurations, and that $NEXT(\alpha_j, \alpha_{j+1})$ is true for every $j \in \{0, \ldots, t-2\}$.

Proof of Theorem 10.8. The proof will be obtained by a reduction from the Halting problem. Specifically, we will use the notion of a configuration of a NAND++ program (Definition 7.12) that we have seen in the context of proving that one dimensional cellular automata are Turing complete. We need the following facts about configurations:

- For every (well formed) NAND++ program $P$, there is a finite alphabet $\Sigma$, and a configuration of $P$ is a string $\alpha \in \Sigma^\ast$.
- A configuration $\alpha$ encodes all the state of the program at a particular iteration, including the array, scalar, and index variables.
- If $\alpha$ is a configuration, then $\beta = NEXT_P(\alpha)$ denotes the configuration of the computation after one more iteration. $\beta$ is a string over $\Sigma$ of length either $|\alpha|$ or $|\alpha| + 1$, and every coordinate of $\beta$ is a function of just three coordinates in $\alpha$. That is, for every $j \in \{0, \ldots, |\alpha|\}$,
\[ \beta_j = MAP_p(\alpha_{j-1}, \alpha_j, \alpha_{j+1}) \]

where \( MAP_p : \Sigma^3 \to \Sigma \) is some function depending on \( P \).\(^6\)

- There are simple conditions to check whether a string \( \alpha \) is a valid starting configuration corresponding to an input \( x \), as well as to check whether a string \( \alpha \) is an halting configuration. In particular these conditions can be phrased as quantified mixed statements.

  - A program \( P \) halts on input \( x \) if and only if there exists a sequence of configurations \( H = (\alpha^0, \alpha^1, \ldots, \alpha^{T-1}) \) such that (i) \( \alpha^0 \) is a valid starting configuration of \( P \) with input \( x \), (ii) \( \alpha^{T-1} \) is a valid halting configuration of \( P \), and (iii) \( \alpha^{i+1} = NEXT_P(\alpha^i) \) for every \( i \in \{0, \ldots, T - 2\} \).

Let \( U \) be a universal NAND++ program. Such a program exists by Theorem 8.1. We define \( HALT_U \) as the function such that \( HALT_U(w) = 1 \) if and only if \( U \) halts on the input \( w \). We claim that the function \( HALT_U \) is uncomputable. Indeed, for every NAND++ program \( P \) (which we identify with its representation as a string) and input \( x \in \{0, 1\}^* \) to \( P \), \( HALT(P, x) = HALT_U(\langle P, x \rangle) \) where \( \langle P, x \rangle \) is some encoding of the pair \( (P, x) \) as a string. Hence if we could compute \( HALT_U \) then we could compute \( HALT \), contradicting Theorem 8.3.

Let \( \Sigma \) be the alphabet needed to encode configurations of \( U \), and let \( \ell = \lceil \log(|\Sigma| + 1) \rceil \). Then we can encode any symbol in \( \Sigma \cup \{\;\} \) (where \( \;\) is some separator symbol we’ll use) as a string in \( \{0, 1\}^\ell \), and so in particular can encode a sequence \( \alpha^0; \alpha^1; \ldots; \alpha^T \) of configurations of \( U \) as a single binary string which we’ll also name as \( H \). Given any input \( w \in \{0, 1\}^* \), we will create a mixed integer statement \( \varphi_w \) that will have the following form:

\[
\varphi_w = \exists_{H \in \{0, 1\}}. H \text{ encodes a valid sequence of configurations of a halting computation of } U \text{ on } w
\]

(10.6)

The reasons we can encode this condition as an MIS are the following:

1. The conditions for checking that the initial configuration is valid are simple, and we can extract the first configuration from \( H \) by first looking at an index \( i \) which is a multiple of \( \ell \) such that \( H_i \cdots H_{i+\ell-1} \) encodes the separator symbol \( \;\); and such that \( i \) is the first such index. Another way to say it is that \( i \) is the position of the first separator if there exists \( k \) such that \( i = k \times \ell \) and \( H_{i-\ell}, \ldots, H_{i+\ell-1} \) and for every \( j \in \mathbb{N} \), if \( j < i \) then \( H_{i+j+\ell-1} \) does not encode \( \;\). This can be captured using the operators allowed in a quantified mixed statement and the \( \forall \) and \( \exists \) quantifiers.
2. We can similarly check that the last configuration is halting. Extracting the position $i$ that encodes the last separator can be done in a way analogous to that of extracting the first one.

3. We can define a quantified mixed predicate $NEXT(\alpha, \beta)$ that is true if and only if $\beta = NEXT_{\ell_t}(\beta)$ (i.e., $\beta$ encodes the configuration obtained by proceeding from $\alpha$ in one computational step). Indeed $NEXT(\alpha, \beta)$ is true if for every $i \in \{0, \ldots, |\beta|\}$ which is a multiple of $\ell$, $\beta_{\ell_{t_i}, \ldots, \ell_{t_i}+\ell-1} = MAP_U(\alpha_{\ell_{t_i-\ell}, \ldots, \ell_{t_i}+2\ell-1})$ where $MAP_U : \{0, 1\}^{3\ell} \to \{0, 1\}$ is the finite function above (identifying elements of $\Sigma$ with their encoding in $\{0, 1\}^\ell$). Since $MAP_U$ is a finite function, we can express it using the logical operations $AND, OR, NOT$ (for example by computing $MAP_U$ with $NAND$’s).

4. We can then write the condition that for every substring of $H$ that has the form $\alpha ENC(;) \beta$ with $\alpha, \beta \in \{0, 1\}^\ell$ and $ENC(;)$ being the encoding of the separator “;”, it holds that $NEXT(\alpha, \beta)$ is true.

Together the above yields a computable procedure that maps every input $w \in \{0, 1\}^\ast$ to $HALT_U$ into a quantified mixed statement $\varphi_w$ such that $HALT_U(w) = 1$ if and only if $QMS(\varphi_w) = 1$. This reduces computing $HALT_U$ to computing $QMS$, and hence the uncomputability of $HALT_U$ implies the uncomputability of $QMS$. $\blacksquare$

10.4.2 Step 2: Reducing mixed statements to integer statements
We now show how to prove Theorem 10.5 using Theorem 10.8. The idea is again a proof by reduction. We will show a transformation of every quantifier mixed statement $\varphi$ into a quantified integer statement $\xi$ that does not use string-valued variables such that $\varphi$ is true if and only if $\xi$ is true.

To remove string-valued variables from a statement, we encode them by integers. We will show that we can encode a string $\sigma \in \{0, 1\}^\ast$ by a pair of numbers $(X, n) \in \mathbb{N}$ s.t.

- $n = |\sigma|$
- There is a quantified integer statement $COORD(X, i)$ that for every $i < n$, will be true if $x_i = 1$ and will be false otherwise.

This will mean that we can replace a “for all” quantifier over strings such as $\forall_{\sigma \in \{0, 1\}^\ast}$ with a pair of quantifiers over integers of the form $\forall_{X \in \mathbb{N}} \forall_{n \in \mathbb{N}}$ (and similarly replace an existential quantifier of the form $\exists_{\sigma \in \{0, 1\}^\ast}$ with a pair of quantifiers $\exists_{X \in \mathbb{N}} \exists_{n \in \mathbb{N}}$). We can later replace all calls to $|\sigma|$ by $n$ and all calls to $x_i$ by $COORD(X, i)$. This means that if we are able to define $COORD$ via a quantified integer statement, then we obtain a proof of Theorem 10.5, since we can use it to map
every mixed quantified statement \( \varphi \) to an equivalent quantified integer statement \( \xi \) such that \( \xi \) is true if and only if \( \varphi \) is true, and hence \( QMS(\varphi) = QIS(\xi) \). Such a procedure implies that the task of computing \( QMS \) reduces to the task of computing \( QIS \), which means that the uncomputability of \( QMS \) implies the uncomputability of \( QIS \).

The above shows that proof of the theorem all boils down to finding the right encoding of strings as integers, and the right way to implement \( COORD \) as a quantified integer statement. To achieve this we use the following technical result:

**Lemma 10.9 — Constructible prime sequence.** There is a sequence of prime numbers \( p_0 < p_1 < p_2 < \cdots \) such that there is a quantified integer statement \( PCOORD(p, i) \) that is true if and only if \( p = p_i \).

Using Lemma 10.9 we can encode a \( x \in \{0, 1\}^* \) by the numbers \( (X, n) \) where \( X = \prod_{x_i=1} p_i \) and \( n = |x| \). We can then define the statement \( COORD(X, i) \) as

\[
∀_{p\in\mathbb{N}} PCOORD(p, i) \Rightarrow DIVIDES(p, X) \quad (10.7)
\]

where \( DIVIDES(a, b) \), as before, is defined as \( \exists_{c\in\mathbb{N}} a \times c = b \). Note that indeed if \( X, n \) encodes the string \( x \in \{0, 1\}^* \), then for every \( i < n \), \( COORD(X, i) = x_i \), since \( p_i \) divides \( X \) if and only if \( x_i = 1 \).

Thus all that is left to conclude the proof of Theorem 10.5 is to prove Lemma 10.9, which we now proceed to do.

**Proof.** The sequence of prime numbers we consider is the following: We fix \( C \) to be a sufficiently large constant (\( C = 2^{234} \) will do) and define \( p_i \) to be the smallest prime number that is in the interval \([i + C, (i + C + 1)^3 - 1] \). It is known that there exists such a prime number for every \( i \in \mathbb{N} \). Given this, the definition of \( PCOORD(p, i) \) is simple:

\[
(p > (i+C) \times (i+C) \times (i+C)) \land (p < (i+C+1) \times (i+C+1) \times (i+C+1)) \land (\forall_{p'} \neg PRIME(p') \lor (p' \leq i \lor (p' \geq p))) ,
\]

(10.8)

We leave it to the reader to verify that \( PCOORD(p, i) \) is true iff \( p = p_i \).

To sum up we have shown that for every quantified mixed statement \( \varphi \), we can compute a quantified integer statement \( \xi \) such that \( QMS(\varphi) = 1 \) if and only if \( QIS(\xi) = 1 \). Hence the uncomputability of \( QMS \) (Theorem 10.8) implies the uncomputability of \( QIS \), completing the proof of Theorem 10.5, and so also the proof of Gödel’s Incompleteness Theorem for quantified integer statements (Theorem 10.4).
that seem to have nothing to do with NAND++ programs or other computational models such as determining the satisfiability of diophantine equations.

- This also implies that for any sound proof system (and in particular every finite axiomatic system) \( S \), there are interesting statements \( X \) (namely of the form \( \forall F(x) = 0 \) for an uncomputable function \( F \)) such that \( S \) is not able to prove either \( X \) or its negation.

10.5 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

Exercise 10.1 — Gödel’s Theorem from uncomputability of \( QIS \). Prove Theorem 10.4 using Theorem 10.5

Exercise 10.2 — Expression for floor. Let \( FSQRT(n, m) = \forall j \in \mathbb{N} ((j \times j) > m) \lor (j \leq n) \). Prove that \( FSQRT(n, m) \) is true if and only if \( n = \lfloor \sqrt{m} \rfloor \).

Exercise 10.3 — Expression for computing the index. Recall that in ?? asked you to prove that at iteration \( t \) of a NAND++ program the the variable \( i \) is equal to \( t - r(r + 1) \) if \( t \leq (r + 1)^2 \) and equals \((r + 2)(r + 1)t\) otherwise, where \( r = \lfloor \sqrt{t + 1/4} - 1/2 \rfloor \). Prove that there is a quantified integer statement \( INDEX \) with parameters \( t, i \) such that \( INDEX(t, i) \) is true if and only if \( i \) is the value of \( i \) after \( t \) iterations.

Exercise 10.4 — Expression for computing the previous line. Give the following quantified integer expressions:

1. \( MOD(a, b, c) \) which is true if and only if \( b = a \mod c \). Note if a program has \( s \) lines then the line executed at step \( t \) is equal to \( t \mod s \).

2. Suppose that \( P \) is the three line NAND program listed below. Give a quantified integer statement \( LAST(n, t, t') \) such that \( LAST(t, t') \) is true if and only if \( t' - n \) is the largest step smaller than \( t - n \) in which the variable on the righthand side of the line executed at step \( t - n \) is written to. If this variable is an input variable \( x_i \) then let \( LAST(n, t, t') \) to be true if the current index location equals \( t' \) and
\[ t' < n. \]

\[
y_0 := \text{foo}_i \text{ NAND } \text{foo}_i
\]
\[
\text{foo}_i := x_i \text{ NAND } x_i
\]
\[
\text{loop} := \text{valid}_i \text{ NAND } \text{valid}_i
\]

**Exercise 10.5 — axiomatic proof systems.** For every representation of logical statements as strings, we can define an axiomatic proof system to consist of a finite set of strings \( A \) and a finite set of rules \( I_0, \ldots, I_{m-1} \) with \( I_j : \{0,1\}^{*} \to \{0,1\}^{*} \) such that a proof \((s_1, \ldots, s_n)\) that \( s_n \) is true is valid if for every \( i \), either \( s_i \in A \) or is some \( j \in [m] \) and are \( i_1, \ldots, i_{k_j} < i \) such that \( s_i = I_j(s_{i_1}, \ldots, s_{i_{k_j}}) \). A system is sound if whenever there is no false \( s \) such that there is a proof that \( s \) is true. Prove that for every uncomputable function \( F : \{0,1\}^{*} \to \{0,1\} \) and every sound axiomatic proof system \( S \) (that is characterized by a finite number of axioms and inference rules), there is some input \( x \) for which the proof system \( S \) is not able to prove neither that \( F(x) = 0 \) nor that \( F(x) \neq 0 \).  

\[ \text{TODO: Maybe add an exercise to give a MIS that corresponds to any regular expression.} \]
III
EFFICIENT ALGORITHMS
11

Efficient computation

“The problem of distinguishing prime numbers from composite and of resolving the latter into their prime factors is … one of the most important and useful in arithmetic … Nevertheless we must confess that all methods … are either restricted to very special cases or are so laborious … they try the patience of even the practiced calculator … and do not apply at all to larger numbers.”, Carl Friedrich Gauss, 1798

“For practical purposes, the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite.”, Jack Edmunds, “Paths, Trees, and Flowers”, 1963

“What is the most efficient way to sort a million 32-bit integers?”, Eric Schmidt to Barack Obama, 2008

“I think the bubble sort would be the wrong way to go.”, Barack Obama.

So far we have been concerned with which functions are computable and which ones are not. But now we return to quantitative considerations and study the time that it takes to compute functions mapping strings to strings, as a function of the input length. This is of course extremely important in the practice of computing, and the reason why we often care so much about the difference between $O(n \log n)$ time algorithm and $O(n^2)$ time one. In contexts such as introduction to programming courses, coding interviews, and actual algorithm design, terms such as “$O(n)$ running time” are often used in an informal way. That is, people don’t have a precise definition of what a linear-time algorithm is, but rather assume that “they’ll know

Learning Objectives:

• Describe at a high level some interesting computational problems.
• The difference between polynomial and exponential time.
• Examples of techniques for obtaining efficient algorithms
• Examples of how seemingly small differences in problems can make (at least apparent) huge differences in their computational complexity.
it when they see it”. However, in this course we will make precise definitions, using our mathematical models of computation. This will allow us to ask (and sometimes answer) questions such as:

- “Is there a function that can be computed in $O(n^2)$ time but not in $O(n)$ time?”
- “Are there natural problems for which the best algorithm (and not just the best known) requires $2^{\Omega(n)}$ time?”

In this chapter we will survey some examples of computational problems, for some of which we know efficient (e.g., $n^c$-time for a small constant $c$) algorithms, and for others the best known algorithms are exponential. We want to get a feel as to the kinds of problems that lie on each side of this divide and also see how some seemingly minor changes in formulation can make the (known) complexity of a problem “jump” from polynomial to exponential. We will not formally define the notion of running time in this chapter, and so will use the same “I know it when I see it” notion of an $O(n)$ or $O(n^2)$ time algorithms as one you’ve seen in introduction to computer science courses. In Chapter 12, we will define this notion precisely, using our NAND++ and NAND« programming languages.

One of the nice things about the theory of computation is that it turns out that, like in the context of computability, the details of the precise computational model or programming language don’t matter that much. Specifically, in this course, we will often not be as concerned with the difference between $O(n)$ and $O(n^2)$, as much as the difference between polynomial and exponential running time. One of the interesting phenomena of computing is that there is often a kind of a “threshold phenomenon” or “zero-one law” for running time, where many natural problems can either be solved in polynomial running time with a not-too-large exponent (e.g., something like $O(n^2)$ or $O(n^3)$), or require exponential (e.g., at least $2^{\Omega(n)}$ or $2^{\Omega(\sqrt{n})}$) time to solve. The reasons for this phenomenon are still not fully understood, but some light on this is shed by the concept of NP completeness, which we will encounter later. As we will see, questions about polynomial versus exponential time are often insensitive to the choice of the particular computational model, just like we saw that the question of whether a function $F$ is computable is insensitive to whether you use NAND++, $\lambda$-calculus, Turing machines, or Javascript as your model of computation.

### 11.1 PROBLEMS ON GRAPHS

We now present a few examples of computational problems that people are interested in solving. Many of the problems will involve graphs.
We have already encountered graphs in the context of Boolean circuits, but let us now quickly recall the basic notation. A graph $G$ consists of a set of vertices $V$ and edges $E$ where each edge is a pair of vertices. In a directed graph, an edge is an ordered pair $(u, v)$, which we sometimes denote as $\overrightarrow{uv}$. In an undirected graph, an edge is an unordered pair (or simply a set) \{u, v\} which we sometimes denote as $\overrightarrow{uv}$ or \(u \sim v\).\(^1\) We will assume graphs are undirected and simple (i.e., containing no parallel edges or self-loops) unless stated otherwise.

We typically will think of the vertices in a graph as simply the set $[n]$ of the numbers from 0 till $n - 1$. Graphs can be represented either in the adjacency list representation, which is a list of $n$ lists, with the $i^{th}$ list corresponding to the neighbors of the $i^{th}$ vertex, or the adjacency matrix representation, which is an $n \times n$ matrix $A$ with $A_{i,j}$ equalling 1 if the edge $\overrightarrow{uv}$ is present and equalling 0 otherwise.\(^2\) We can transform between these two representations using $O(n^2)$ operations, and hence for our purposes we will mostly consider them as equivalent. We will sometimes consider labeled or weighted graphs, where we assign a label or a number to the edges or vertices of the graph, but mostly we will try to keep things simple and stick to the basic notion of an unlabeled, unweighted, simple undirected graph.

There is a reason that graphs are so ubiquitous in computer science and other sciences. They can be used to model a great many of the data that we encounter. These are not just the “obvious” networks such as the road network (which can be thought of as a graph of whose vertices are locations with edges corresponding to road segments), or the web (which can be thought of as a graph whose vertices are web pages with edges corresponding to links), or social networks (which can be thought of as a graph whose vertices are people and the edges correspond to friend relation). Graphs can also denote correlations in data (e.g., graph of observations of features with edges corresponding to features that tend to appear together), causal relations (e.g., gene regulatory networks, where a gene is connected to gene products it derives), or the state space of a system (e.g., graph of configurations of a physical system, with edges corresponding to states that can be reached from one another in one step).

We now give some examples of computational problems on graphs. As mentioned above, to keep things simple, we will restrict our attention to undirected simple graphs. In all cases the input graph $G = (V, E)$ will have $n$ vertices and $m$ edges.

### 11.1.1 Finding the shortest path in a graph

The shortest path problem is the task of, given a graph $G = (V, E)$ and two vertices $s, t \in V$, to find the length of the shortest path between $s$ and $t$ (if such a path exists). That is, we want to find the smallest

\(^1\) An equivalent viewpoint is that an undirected graph is like a directed graph with the property that whenever the edge $\overrightarrow{uv}$ is present then so is the edge $\overleftarrow{vu}$.

\(^2\) In an undirected graph, the adjacency matrix $A$ is symmetric, in the sense that it satisfies $A_{i,j} = A_{j,i}$. 
number \( k \) such that there are vertices \( v_0, v_1, \ldots, v_k \) with \( v_0 = s, v_k = t \) and for every \( i \in \{0, \ldots, k - 1\} \) an edge between \( v_i \) and \( v_{i+1} \). Formally, we define \( \text{MINPATH} : \{0, 1\}^* \rightarrow \{0, 1\}^* \) to be the function that on input a triple \((G, s, t)\) (represented as a string) outputs the number \( k \) which is the length of the shortest path in \( G \) between \( s \) and \( t \) or a string representing no path if no such path exists. (In practice people often want to also find the actual path and not just its length; it turns out that the algorithms to compute the length of the path often yield the actual path itself as a byproduct, and so everything we say about the task of computing the length also applies to the task of finding the path.)

If each vertex has at least two neighbors then there can be an exponential number of paths from \( s \) to \( t \), but fortunately we do not have to enumerate them all to find the shortest path. We can do so by performing a breadth first search (BFS), enumerating \( s \)'s neighbors, and then neighbors' neighbors, etc., in order. If we maintain the neighbors in a list we can perform a BFS in \( O(n^2) \) time, while using a queue we can do this in \( O(m) \) time.\(^3\)

More formally, the algorithm for computing the function \( \text{MINPATH} \) can be described as follows:

**Algorithm BFSPATH:**

- **Input:** Graph \( G = (V, E) \), vertices \( s, t \)
- **Goal:** Find the length \( k \) of the shortest path \( v_0, v_1, \ldots, v_k \) such that \( v_0 = s, v_k = t \) and \( \{v_i, v_{i+1}\} \in E \) for every \( i \in [k] \), if such a path exists.
- **Operation:**
  1. We maintain a queue \( Q \) of vertices, initially \( Q \) contains only the pair \( s \).

\(^3\) A queue stores a list of elements in “First In First Out (FIFO)” order and so each “pop” operation removes an element from the queue in the order that they were “pushed” into it; see the Wikipedia page. Since we assume \( m \geq n - 1 \), \( O(m) \) is the same as \( O(n + m) \). Dijkstra’s algorithm is a well-known generalization of BFS to weighted graphs.
2. We maintain a dictionary \(^4\) \(D\) keyed by the vertices, for every vertex \(v\), \(D[v]\) is either equal to a natural number or to \(\infty\). Initially we set set \(D[s] = 0\) and \(D[v] = \infty\) for every \(v \in V \setminus \{s\}\).

3. While \(Q\) is not empty do the following:
   - (a) Pop a vertex \(v\) from the top of the queue.
   - (b) If \(v = t\) then halt and output \(D[v]\).
   - (c) Otherwise, for every neighbor \(w\) of \(v\) such that \(D[w] = \infty\), set \(D[w] = D[v] + 1\) and add \(w\) to the queue.

4. Output “no path”

Since we only add to the queue vertices \(w\) with \(D[w] = \infty\) (and then immediately set \(D[w]\) to an actual number), we never push to the queue a vertex more than once, and hence the algorithm takes \(n\) “push” and “pop” operations. It returns the correct answer since add the vertices to the queue in the order of their distance from \(s\), and hence we will reach \(t\) after we have explored all the vertices that are closer to \(s\) than \(t\). Hence algorithm BFSPATH computes MINPATH.

\(^4\) A dictionary or associative array data structure \(D\) allows to associate with every key \(v\) (which can be thought of as a string) a value \(D[v]\).

11.1.2 Finding the longest path in a graph
The longest path problem is the task of, given a graph \(G = (V, E)\) and two vertices \(s, t \in V\), to find the length of the longest simple (i.e., non
intersections path between $s$ and $t$. If the graph is a road network, then the longest path might seem less motivated than the shortest path, but of course graphs can be and are used to model a variety of phenomena, and in many such cases the longest path (and some of its variants) are highly motivated. In particular, finding the longest path is a generalization of the famous Hamiltonian path problem which asks for a maximally long simple path (i.e., path that visits all $n$ vertices once) between $s$ and $t$, as well as the notorious traveling salesman problem (TSP) of finding (in a weighted graph) a path visiting all vertices of cost at most $w$. TSP is a classical optimization problem, with applications ranging from planning and logistics to DNA sequencing and astronomy.

A priori it is not clear that finding the longest path should be harder than finding the shortest path, but this turns out to be the case. While we know how to find the shortest path in $O(n)$ time, for the longest path problem we have not been able to significantly improve upon the trivial brute force algorithm that tries all paths.

Specifically, in a graph of degree at most $d$, we can enumerate over all paths of length $k$ by going over the (at most $d$) neighbors of each vertex. This would take about $O(d^k)$ steps, and since the longest simple path can’t have length more than the number of vertices, this means that the brute force algorithms runs in $O(d^n)$ time (which we can bound by $O(n^n)$ since the maximum degree is $n$). The best algorithm for the longest path improves on this, but not by much: it takes $\Omega(c^n)$ time for some constant $c > 1$.

11.1.3 Finding the minimum cut in a graph

Given a graph $G = (V, E)$, a cut is a subset $S$ of $V$ such that $S$ is neither empty nor is it all of $V$. The edges cut by $S$ are those edges where one of their endpoints is in $S$ and the other is in $\overline{S} = V \setminus S$. We denote this set of edges by $E(S, \overline{S})$. If $s, t \in V$ then an $s, t$ cut is a cut such that $s \in S$ and $t \in \overline{S}$. (See Fig. 11.3.) The minimum $s, t$ cut problem is the task of finding, given $s$ and $t$, the minimum number $k$ such that there is an $s, t$ cut cutting $k$ edges (once again, the problem is also sometimes phrased as finding the set that achieves this minimum; it turns out that algorithms to compute the number often yield the set as well). Formally, we define $\text{MINCUT} : \{0, 1\}^* \rightarrow \{0, 1\}^*$ to be the function that on input a triple $(G, s, t)$ of a graph and two vertices (represented as a string), outputs the minimum number $k$ such that there exists a set $S$ containing $s$ and not $t$ with exactly $k$ edges that touch $S$ and its complement.

The minimum $s, t$ cut problem appears in many applications. Minimum cuts often correspond to bottlenecks. For example, in a communication network the minimum cut between $s$ and $t$ corresponds to the

At the moment the best record is $c \sim 1.65$ or so. Even obtaining an $O(2^n)$ time bound is not that simple, see Exercise 11.1.

One can also define the problem of finding the global minimum cut (i.e., the non-empty and non-everything set $S$ that minimizes the number of edges cut). A polynomial time algorithm for the minimum $s, t$ cut can be used to solve the global minimum cut in polynomial time as well (can you see why?).
Figure 11.2: A knight’s tour can be thought of as a maximally long path on the graph corresponding to a chessboard where we put an edge between any two squares that can be reached by one step via a legal knight move.

Figure 11.3: A cut in a graph $G = (V, E)$ is simply a subset $S$ of its vertices. The edges that are cut by $S$ are all those whose one endpoint is in $S$ and the other one is in $\overline{S} = V \setminus S$. The cut edges are colored red in this figure.
smallest number of edges that, if dropped, will disconnect $s$ from $t$. Similar applications arise in scheduling and planning. In the setting of image segmentation, one can define a graph whose vertices are pixels and whose edges correspond to neighboring pixels of distinct colors. If we want to separate the foreground from the background then we can pick (or guess) a foreground pixel $s$ and background pixel $t$ and ask for a minimum cut between them.

Here is an algorithm to compute $MINCUT$:

**Algorithm MINCUTNAIVE:**

- **Input:** Graph $G = (V, E)$ and two distinct vertices $s, t \in V$
- **Goal:** Return $k = \min_{S \subseteq V, s \in S, t \notin S} |E(S, \overline{S})|$
- **Operation:**
  1. Let $k_0 \leftarrow |E| + 1$
  2. For every set $S \subseteq V$ such that $s \in S$ and $t \notin T$ do:
     (a) Set $k \leftarrow 0$.
     (b) For every edge $\{u, v\} \in E$, if $u \in S$ and $v \notin S$ then set $k \leftarrow k + 1$.
     (c) If $k < k_0$ then let $k_0 \leftarrow k$
  3. Return $k_0$

It is an excellent exercise for you to pause at this point and verify: (i) that you understand what this algorithm does, (ii) that you understand why this algorithm will in fact return the value of the minimum cut in the graph, and (iii) that you can analyze the running time of this algorithm.

The precise running time of algorithm MINCUTNAIVE will depend on the data structures we use to store the graph and the sets, but even if we had the best data structures, the running time of MINCUTNAIVE will be terrible. Indeed, if a graph has $n$ vertices, then for every pair $s, t$ of distinct vertices, there are $2^{n-2}$ sets $S$ that contain $s$ but don’t contain $t$. (Can you see why?) Since we are enumerating over all of those in Step 2, even if we could compute for each such set $S$ the value $|E(S, \overline{S})|$ in constant time, our running time would still be exponential.

Since minimum cut is a problem we want to solve, this seems like bad news. After all, MINCUTNAIVE is the most natural algorithm to solve the problem, and if it takes exponential time, then perhaps the problem can’t be solved efficiently at all. However, this turns out not to be case. As we’ve seen in this course time and again, there is
a difference between the function \textit{MINCUT} and the algorithm \textit{MINCUTNAIVE} to solve it. There can be more than one algorithm to compute the same function, and some of those algorithms might be more efficient than others. Luckily this is one of those cases. There do exist much faster algorithms that compute \textit{MINCUT} in \textit{polynomial time} (which, as mentioned in the mathematical background lecture, we denote by \textit{poly}(n)).

There are several algorithms to do so, but many of them rely on the \textbf{Max-Flow Min-Cut Theorem} that says that the minimum cut between \textit{s} and \textit{t} equals the maximum amount of \textit{flow} we can send from \textit{s} to \textit{t}, if every edge has unit capacity. Specifically, imagine that every edge of the graph corresponded to a pipe that could carry one unit of water per one unit of time (say 1 liter of water per second). Now suppose we want to send a maximum amount of water per time unit from our source \textit{s} to the sink \textit{t}. If there is an \textit{s}, \textit{t}-cut of at most \textit{k} edges, then this maximum will be at most \textit{k}. Indeed, such a cut \textit{S} will be a “bottleneck” since at most \textit{k} units can flow from \textit{S} to its complement \textit{S}. The above reasoning can be used to show that the maximum flow from \textit{s} to \textit{t} is \textit{at most} the value of the minimum \textit{s}, \textit{t}-cut. The surprising and non-trivial content of the Max-Flow Min-Cut Theorem is that the maximum flow is also \textit{at least} the value of the minimum cut, and hence computing the cut is the same as computing the flow.

A flow on a graph \textit{G} of \textit{m} edges can be thought of as a vector \(x \in \mathbb{R}^m\) where for every edge \(e\), \(x_e\) corresponds to the amount of water per time-unit that flows on \(e\). We think of an edge \(e\) an an ordered pair \((u, v)\) (we can choose the order arbitrarily) and let \(x_e\) be the amount of flow that goes from \(u\) to \(v\). (If the flow is in the other direction then we make \(x_e\) negative.) Since every edge has capacity one, we know that \(-1 \leq x_e \leq 1\) for every edge \(e\). A valid flow has the property that the amount of water leaving the source \(s\) is the same as the amount entering the sink \(t\), and that for every other vertex \(v\), the amount of water entering and leaving \(v\) is the same.

Mathematically, we can write these conditions as follows:

\[
\sum_{e \in s} x_e + \sum_{e \in t} x_e = 0 \\
\sum_{e \not\in v} x_e = 0 \quad \forall e \in V \setminus \{s, t\} \\
-1 \leq x_e \leq 1 \quad \forall e \in E
\]  

(11.1)

where for every vertex \(v\), summing over \(e \ni v\) means summing over all the edges that touch \(v\).

The maximum flow problem can be thought of as the task of maximizing \(\sum_{e \ni s} x_e\) over all the vectors \(x \in \mathbb{R}^m\) that satisfy the above
conditions Eq. (11.1). This is a special case of a very general task known as linear programming, where one wants to find the maximum of $f(x)$ over $x \in \mathbb{R}^n$ that satisfies certain linear inequalities where $f : \mathbb{R}^n \to \mathbb{R}$ is a linear function. Luckily, there are polynomial-time algorithms for solving linear programming, and hence we can solve the maximum flow (and so, equivalently, minimum cut) problem in polynomial time. In fact, there are much better algorithms for maximum-flow/minimum-cut, even for weighted directed graphs, with currently the record standing at $O(\min\{m^{10/7}, m\sqrt{n}\})$.

### 11.1.4 Finding the maximum cut in a graph

We can also define the maximum cut problem of finding, given a graph $G = (V, E)$ the subset $S \subseteq V$ that maximizes the number of edges cut by $S$. Like its cousin the minimum cut problem, the maximum cut problem is also very well motivated. For example, it arises in VLSI design, and also has some surprising relation to analyzing the Ising model in statistical physics.

Once again, a priori it might not be clear that the maximum cut problem should be harder than minimum cut but this turns out to be the case. We do not know of an algorithm that solves this problem much faster than the trivial “brute force” algorithm that tries all $2^n$ possibilities for the set $S$.

### 11.1.5 A note on convexity

There is an underlying reason for the sometimes radical difference between the difficulty of maximizing and minimizing a function over a domain. If $D \subseteq \mathbb{R}^n$, then a function $f : D \to \mathbb{R}$ is convex if for every $x, y \in D$ and $p \in [0, 1]$ $f(px + (1 - p)y) \leq pf(x) + (1 - p)f(y)$. That is, $f$ applied to the $p$-weighted midpoint between $x$ and $y$ is smaller than the $p$-weighted average value of $f$. If $D$ itself is convex (which means that if $x, y$ are in $D$ then so is the line segment between them), then this means that if $x$ is a local minimum of $f$ then it is also a global minimum. The reason is that if $f(y) < f(x)$ then every point...
Figure 11.5: In the high dimensional case, if $f$ is a convex function (left figure) the global minimum is the only local minimum, and we can find it by a local-search algorithm which can be thought of as dropping a marble and letting it “slide down” until it reaches the global minimum. In contrast, a non-convex function (right figure) might have an exponential number of local minima in which any local-search algorithm could get stuck.

\[ z = px + (1 - p)y \]

on the line segment between $x$ and $y$ will satisfy $f(z) \leq pf(x) + (1 - p)f(y) < f(x)$ and hence in particular $x$ cannot be a local minimum. Intuitively, local minima of functions are much easier to find than global ones: after all, any “local search” algorithm that keeps finding a nearby point on which the value is lower, will eventually arrive at a local minima. Indeed, under certain technical conditions, we can often efficiently find the minimum of convex functions, and this underlies the reason problems such as minimum cut and shortest path are easy to solve. On the other hand, maximizing a convex function (or equivalently, minimizing a concave function) can often be a hard computational task. A linear function is both convex and concave, which is the reason both the maximization and minimization problems for linear functions can be done efficiently.

The minimum cut problem is not a priori a convex minimization task, because the set of potential cuts is discrete. However, it turns out that we can embed it in a continuous and convex set via the (linear) maximum flow problem. The “max flow min cut” theorem ensuring that this embedding is “tight” in the sense that the minimum “fractional cut” that we obtain through the maximum-flow linear program will be the same as the true minimum cut. Unfortunately, we don’t know of such a tight embedding in the setting of the maximum cut problem.

The issue of convexity arises time and again in the context of computation. For example, one of the basic tasks in machine learning is empirical risk minimization. That is, given a set of labeled examples $(x_1, y_1), \ldots, (x_m, y_m)$, where each $x_i \in \{0,1\}^n$ and $y_i \in \{0,1\}$, we want to find the function $h : \{0,1\}^n \rightarrow \{0,1\}$ from some class $H$ that minimizes the error in the sense of minimizing the number of $i$’s such that $h(x_i) \neq y_i$. Like in the minimum cut problem, to make this a better behaved computational problem, we often embed it in a continuous domain, including functions that could output a real number and replacing the condition $h(x_i) \neq y_i$ with minimizing some continuous
loss function $\ell(h(x), y)$.\(^\text{10}\) When this embedding is convex then we are guaranteed that the global minimizer is unique and can be found in polynomial time. When the embedding is non convex, we have no such guarantee and in general there can be many global or local minima. That said, even if we don’t find the global (or even a local) minima, this continuous embedding can still help us. In particular, when running a local improvement algorithm such as Gradient Descent, we might still find a function $h$ that is “useful” in the sense of having a small error on future examples from the same distribution.\(^\text{11}\)

### 11.2 BEYOND GRAPHS

Not all computational problems arise from graphs. We now list some other examples of computational problems that are of great interest.

#### 11.2.1 The 2SAT problem

A propositional formula $\varphi$ involves $n$ variables $x_1, \ldots, x_n$ and the logical operators AND ($\land$), OR ($\lor$), and NOT ($\neg$, also denoted as $\bar{\cdot}$). We say that such a formula is in conjunctive normal form (CNF for short) if it is an AND of ORs of variables or their negations (we call a term of the form $x_i$ or $\bar{x}_i$ a literal). For example, this is a CNF formula

\[
(x_7 \lor \bar{x}_{22} \lor x_{15}) \land (x_{37} \lor x_{22}) \land (x_{55} \lor \bar{x}_7)
\]

We say that a formula is a $k$-CNF it is an AND of ORs where each OR involves exactly $k$ literals. The 2SAT problem is to find out, given a 2-CNF formula $\varphi$, whether there is an assignment $x \in \{0, 1\}^n$ that satisfies $\varphi$, in the sense that it makes it evaluate to 1 or “True”.

Determining the satisfiability of Boolean formulas arises in many applications and in particular in software and hardware verification, as well as scheduling problems. The trivial, brute-force, algorithm for 2SAT will enumerate all the $2^n$ assignments $x \in \{0, 1\}^n$ but fortunately we can do much better.

The key is that we can think of every constraint of the form $\ell_i \lor \ell_j$ (where $\ell_i, \ell_j$ are literals, corresponding to variables or their negations) as an implication $\bar{\ell}_i \Rightarrow \ell_j$, since it corresponds to the constraints that if the literal $\ell'_i = \bar{\ell}_i$ is true then it must be the case that $\ell_j$ is true as well. Hence we can think of $\varphi$ as a directed graph between the $2n$ literals, with an edge from $\ell_i$ to $\ell_j$ corresponding to an implication from the former to the latter. It can be shown that $\varphi$ is unsatisfiable if and only if there is a variable $x_i$ such that there is a directed path from $x_i$ to $\bar{x}_i$ as well as a directed path from $\bar{x}_i$ to $x_i$ (see Exercise 11.2). This reduces 2SAT to the (efficiently solvable) problem of determining connectivity in directed graphs.

\(^{10}\) We also sometimes replace or enhance the condition that $h$ is in the class $H$ by adding a regularizing term of the form $R(h)$ to the minimization problem, where $R : H \rightarrow \mathbb{R}$ is some measure of the “complexity” of $h$. As a general rule, the larger or more “complex” functions $h$ we allow, the easier it is to fit the data, but the more danger we have of “overfitting”.

\(^{11}\) In machine learning parlance, this task is known as supervised learning. The set of examples $(x_1, y_1), \ldots, (x_m, y_m)$ is known as the training set, and the error on additional samples from the same distribution is known as the generalization error, and can be measured by checking $h$ against a test set that was not used in training it.
11.2.2 The 3SAT problem
The 3SAT problem is the task of determining satisfiability for 3CNFs. One might think that changing from two to three would not make that much of a difference for complexity. One would be wrong. Despite much effort, we do not know of a significantly better than brute force algorithm for 3SAT (the best known algorithms take roughly $1.3^n$ steps).

Interestingly, a similar issue arises time and again in computation, where the difference between two and three often corresponds to the difference between tractable and intractable. We do not fully understand the reasons for this phenomenon, though the notions of NP completeness we will see later does offer a partial explanation. It may be related to the fact that optimizing a polynomial often amounts to equations on its derivative. The derivative of a a quadratic polynomial is linear, while the derivative of a cubic is quadratic, and, as we will see, the difference between solving linear and quadratic equations can be quite profound.

11.2.3 Solving linear equations
One of the most useful problems that people have been solving time and again is solving $n$ linear equations in $n$ variables. That is, solve equations of the form

$$
\begin{align*}
a_{0,0}x_0 + a_{0,1}x_1 &+ \cdots + a_{0,n-1}x_{n-1} = b_0 \\
a_{1,0}x_0 + a_{1,1}x_1 &+ \cdots + a_{1,n-1}x_{n-1} = b_1 \\
\vdots &+ \vdots \\
a_{n-1,0}x_0 + a_{n-1,1}x_1 &+ \cdots + a_{n-1,n-1}x_{n-1} = b_{n-1}
\end{align*}
$$

(11.3)

where $\{a_{i,j}\}_{i,j\in[n]}$ and $\{b_i\}_{i\in[n]}$ are real (or rational) numbers. More compactly, we can write this as the equations $Ax = b$ where $A$ is an $n \times n$ matrix, and we think of $x, b$ are column vectors in $\mathbb{R}^n$.

The standard Gaussian elimination algorithm can be used to solve such equations in polynomial time (i.e., determine if they have a solution, and if so, to find it). As we discussed above, if we are willing to allow some loss in precision, we even have algorithms that handle linear inequalities, also known as linear programming. In contrast, if we insist on integer solutions, the task of solving for linear equalities or inequalities is known as integer programming, and the best known algorithms are exponential time in the worst case.

To analyze this fully we need to ensure that the bit complexity of the numbers involved does not grow too much, but fortunately we can indeed ensure this using . Also, as is usually the case when talking about real numbers, we do not care much for the distinction between solving equations exactly and solving them to arbitrarily good precision.
up to a constant factor, the number of digits in base 10, 16 or any other constant). The difference between the length of the input and the magnitude of the number itself can be of course quite profound. For example, most people would agree that there is a huge difference between having a billion (i.e. $10^9$) dollars and having nine dollars. Similarly there is a huge difference between an algorithm that takes $n$ steps on an $n$-bit number and an algorithm that takes $2^n$ steps.

One example, is the problem (discussed below) of finding the prime factors of a given integer $N$. The natural algorithm is to search for such a factor by trying all numbers from 1 to $N$, but that would take $N$ steps which is exponential in the input length, which is number of bits needed to describe $N$. It is an important and long open question whether there is such an algorithm that runs in time polynomial in the input length (i.e., polynomial in $\log N$).

11.2.4 Solving quadratic equations

Suppose that we want to solve not just linear but also equations involving quadratic terms of the form $a_{i,j,k}x^i_j x^k$. That is, suppose that we are given a set of quadratic polynomials $p_1, \ldots, p_m$ and consider the equations $\{p_i(x) = 0\}$. To avoid issues with bit representations, we will always assume that the equations contain the constraints $\{x_i^2 - x_i = 0\}_{i \in [n]}$. Since only 0 and 1 satisfy the equation $a^2 - a$, this assumption means that we can restrict attention to solutions in $\{0, 1\}^n$.

Solving quadratic equations in several variable is a classical and extremely well motivated problem. This is the generalization of the classical case of single-variable quadratic equations that generations of high school students grapple with. It also generalizes the quadratic assignment problem, introduced in the 1950’s as a way to optimize assignment of economic activities. Once again, we do not know a much better algorithm for this problem than the one that enumerates over all the $2^n$ possibilities.

11.3 MORE ADVANCED EXAMPLES

We now list a few more examples of interesting problems that are a little more advanced but are of significant interest in areas such as physics, economics, number theory, and cryptography.

11.3.1 Determinant of a matrix

The determinant of a $n \times n$ matrix $A$, denoted by $\det(A)$, is an extremely important quantity in linear algebra. For example, it is known that $\det(A) \neq 0$ if and only if $A$ is nonsingular, which means that it
A triangular matrix is one in which either all entries below the diagonal, or all entries above the diagonal, are zero. The cost for performing each arithmetic operation depends on the number of bits needed to represent each entry, and accounting for this can sometimes be subtle, though ultimately doable. has an inverse $A^{-1}$, and hence we can always uniquely solve equations of the form $Ax = b$ where $x$ and $b$ are $n$-dimensional vectors. More generally, the determinant can be thought of as a quantitative measure as to what extent $A$ is far from being singular. If the rows of $A$ are “almost” linearly dependent (for example, if the third row is very close to being a linear combination of the first two rows) then the determinant will be small, while if they are far from it (for example, if they are are orthogonal to one another, then the determinant will be large). In particular, for every matrix $A$, the absolute value of the determinant of $A$ is at most the product of the norms (i.e., square root of sum of squares of entries) of the rows, with equality if and only if the rows are orthogonal to one another.

The determinant can be defined in several ways. For example, it is known that det is the only function that satisfies the following conditions:

1. $\det(AB) = \det(A)\det(B)$ for every square matrices $A, B$.
2. For every $n \times n$ triangular matrix $T$ with diagonal entries $d_0, \ldots, d_{n-1}$, $\det(T) = \prod_{i=0}^n d_i$. In particular $\det(I) = 1$ where $I$ is the identity matrix.\footnote{A triangular matrix is one in which either all entries below the diagonal, or all entries above the diagonal, are zero.}
3. $\det(S) = -1$ where $S$ is a “swap matrix” that corresponds to swapping two rows or two columns of $I$. That is, there are two coordinates $a, b$ such that for every $i, j$, $S_{i,j} =\begin{cases} 1 & i = j \lor i \in \{a, b\} \\ 1 & \{i, j\} = \{a, b\} \\ 0 & \text{otherwise} \end{cases}$.

Note that conditions 1. and 2. together imply that $\det(A^{-1}) = \det(A)^{-1}$ for every invertible matrix $A$. Using these rules and the Gaussian elimination algorithm, it is possible to tell whether $A$ is singular or not, and in the latter case, decompose $A$ as a product of a polynomial number of swap matrices and triangular matrices. (Indeed one can verify that the row operations in Gaussian elimination corresponds to either multiplying by a swap matrix or by a triangular matrix.) Hence we can compute the determinant for an $n \times n$ matrix using a polynomial time of arithmetic operations.\footnote{The cost for performing each arithmetic operation depends on the number of bits needed to represent each entry, and accounting for this can sometimes be subtle, though ultimately doable.}

11.3.2 The permanent (mod 2) problem

Given an $n \times n$ matrix $A$, the permanent of $A$ is the sum over all permutations $\pi$ (i.e., $\pi$ is a member of the set $S_n$ of one-to-one and onto functions from $[n]$ to $[n]$) of the product $\prod_{i=0}^{n-1} A_{i,\pi(i)}$. The permanent of a matrix is a natural quantity, and has been studied in several contexts including combinatorics and graph theory. It also arises in physics where it can be used to describe the quantum state of multiple boson particles (see here and here).
If the entries of $A$ are integers, then we can also define a Boolean function $\text{perm}_2(A)$ which will output the result of the permanent modulo 2. A priori computing this would seem to require enumerating over all $n!$ possibilities. However, it turns out we can compute $\text{perm}_2(A)$ in polynomial time! The key is that modulo 2, $-x$ and $+x$ are the same quantity and hence the permanent modulo 2 is the same as taking the following quantity modulo 2:

$$\sum_{\pi \in S_n} \text{sign}(\pi) \prod_{i=0}^{n-1} A_{i,\pi(i)}$$  \hspace{1cm} (11.4)

where the sign of a permutation $\pi$ is a number in $\{+1, -1\}$ which can be defined in several ways, one of which is that $\text{sign}(\pi)$ equals $+1$ if the number of swaps that “Bubble” sort performs starting an array sorted according to $\pi$ is even, and it equals $-1$ if this number is odd.\(^\dagger\)

From a first look, Eq. (11.4) does not seem like it makes much progress. After all, all we did is replace one formula involving a sum over $n!$ terms with an even more complicated formula involving a sum over $n!$ terms. But fortunately Eq. (11.4) also has an alternative description: it is yet another way to describe the determinant of the matrix $A$, which as mentioned can be computed using a process similar to Gaussian elimination.

11.3.3 The permanent (mod 3) problem

Emboldened by our good fortune above, we might hope to be able to compute the permanent modulo any prime $p$ and perhaps in full generality. Alas, we have no such luck. In a similar “two to three” type of a phenomenon, we do not know of a much better than brute force algorithm to even compute the permanent modulo 3.

11.3.4 Finding a zero-sum equilibrium

A zero sum game is a game between two players where the payoff for one is the same as the penalty for the other. That is, whatever the first player gains, the second player loses. As much as we want to avoid them, zero sum games do arise in life, and the one good thing about them is that at least we can compute the optimal strategy.

A zero sum game can be specified by an $n \times n$ matrix $A$, where if player 1 chooses action $i$ and player 2 chooses action $j$ then player one gets $A_{i,j}$ and player 2 loses the same amount. The famous Min Max Theorem by John von Neumann states that if we allow probabilistic or “mixed” strategies (where a player does not choose a single action but rather a distribution over actions) then it does not matter who plays first and the end result will be the same. Mathematically the min max theorem is that if we let $\Delta_n$ be the set of probability distributions over $[n]$ (i.e., non-negative columns vectors in $\mathbb{R}^n$ whose entries sum to 1)
then

\[
\max_{p \in \Delta_n} \min_{q \in \Delta_n} p^\top A q = \min_{q \in \Delta_n} \max_{p \in \Delta_n} p^\top A q \quad \text{(11.5)}
\]

The min-max theorem turns out to be a corollary of linear programming duality, and indeed the value of Eq. (11.5) can be computed efficiently by a linear program.

### 11.3.5 Finding a Nash equilibrium

Fortunately, not all real-world games are zero sum, and we do have more general games, where the payoff of one player does not necessarily equal the loss of the other. John Nash won the Nobel prize for showing that there is a notion of equilibrium for such games as well. In many economic texts it is taken as an article of faith that when actual agents are involved in such a game then they reach a Nash equilibrium. However, unlike zero sum games, we do not know of an efficient algorithm for finding a Nash equilibrium given the description of a general (non zero sum) game. In particular this means that, despite economists’ intuitions, there are games for which natural strategies will take exponential number of steps to converge to an equilibrium.

### 11.3.6 Primality testing

Another classical computational problem, that has been of interest since the ancient greeks, is to determine whether a given number \( N \) is prime or composite. Clearly we can do so by trying to divide it with all the numbers in \( 2, \ldots, N - 1 \), but this would take at least \( N \) steps which is \textit{exponential} in its bit complexity \( n = \log N \). We can reduce these \( N \) steps to \( \sqrt{N} \) by observing that if \( N \) is a composite of the form \( N = PQ \) then either \( P \) or \( Q \) is smaller than \( \sqrt{N} \). But this is still quite terrible. If \( N \) is a 1024 bit integer, \( \sqrt{N} \) is about \( 2^{512} \), and so running this algorithm on such an input would take much more than the lifetime of the universe.

Luckily, it turns out we can do radically better. In the 1970’s, Rabin and Miller gave \textit{probabilistic} algorithms to determine whether a given number \( N \) is prime or composite in time \( \text{poly}(n) \) for \( n = \log N \). We will discuss the probabilistic model of computation later in this course. In 2002, Agrawal, Kayal, and Saxena found a deterministic \( \text{poly}(n) \) time algorithm for this problem. This is surely a development that mathematicians from Archimedes till Gauss would have found exciting.

### 11.3.7 Integer factoring

Given that we can efficiently determine whether a number \( N \) is prime or composite, we could expect that in the latter case we could also ef-
iciently find the factorization of $N$. Alas, no such algorithm is known. In a surprising and exciting turn of events, the non existence of such an algorithm has been used as a basis for encryptions, and indeed it underlies much of the security of the world wide web. We will return to the factoring problem later in this course. We remark that we do know much better than brute force algorithms for this problem. While the brute force algorithms would require $2^{O(n)}$ time to factor an $n$-bit integer, there are known algorithms running in time roughly $2^{O(\sqrt{n})}$ and also algorithms that are widely believed (though not fully rigorously analyzed) to run in time roughly $2^{O(n^{1/3})}$.

11.4 OUR CURRENT KNOWLEDGE

![Figure 11.6](image-url)

**Figure 11.6**: The current computational status of several interesting problems. For all of them we either know a polynomial-time algorithm or the known algorithms require at least $2^{n^c}$ for some $c > 0$. In fact for all except the factoring problem, we either know an $O(n^3)$ time algorithm or the best known algorithm require at least $2^{O(n)}$ time where $n$ is a natural parameter such that there is a brute force algorithm taking roughly $2^n$ or $n!$ time. Whether this “cliff” between the easy and hard problem is a real phenomenon or a reflection of our ignorance is still an open question.

The difference between an exponential and polynomial time algorithm might seem merely “quantitative” but it is in fact extremely significant. As we’ve already seen, the brute force exponential time algorithm runs out of steam very very fast, and as Edmonds says, in practice there might not be much difference between a problem where the best algorithm is exponential and a problem that is not solvable at all. Thus the efficient algorithms we mention above are widely used and power many computer science applications. Moreover, a polynomial-time algorithm often arises out of significant insight to the problem at hand, whether it is the “max-flow min-cut” result, the solvability of the determinant, or the group theoretic structure that

---

17 The “roughly” adjective above refers to neglecting factors that are polylogarithmic in $n$. 
enables primality testing. Such insight can be useful regardless of its computational implications.

At the moment we do not know whether the “hard” problems are truly hard, or whether it is merely because we haven’t yet found the right algorithms for them. However, we will now see that there are problems that do inherently require exponential time. We just don’t know if any of the examples above fall into that category.

11.5 LECTURE SUMMARY

• There are many natural problems that have polynomial-time algorithms, and other natural problems that we’d love to solve, but for which the best known algorithms are exponential.

• Often a polynomial time algorithm relies on discovering some hidden structure in the problem, or finding a surprising equivalent formulation for it.

• There are many interesting problems where there is an exponential gap between the best known algorithm and the best algorithm that we can rule out. Closing this gap is one of the main open questions of theoretical computer science.

11.6 EXERCISES

Exercise 11.1 — exponential time algorithm for longest path. The naive algorithm for computing the longest path in a given graph could take more than $n!$ steps. Give a $poly(n)2^n$ time algorithm for the longest path problem in $n$ vertex graphs.

Exercise 11.2 — 2SAT algorithm. For every 2CNF $\varphi$, define the graph $G_\varphi$ on $2n$ vertices corresponding to the literals $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$, such that there is an edge $\ell_i \ell_j$ iff the constraint $\overline{t}_i \lor \ell_j$ is in $\varphi$. Prove that $\varphi$ is unsatisfiable if and only if there is some $i$ such that there is a path from $x_i$ to $\overline{x}_i$ and from $x_i$ to $x_i$ in $G_\varphi$. Show how to use this to solve 2SAT in polynomial time.

\[18\text{ Hint: Use dynamic programming to compute for every } s, t \in [n] \text{ and } S \subseteq [n] \text{ the value } P(s, t, S) \text{ which equals } 1 \text{ if there is a simple path from } s \text{ to } t \text{ that uses exactly the vertices in } S. \]
11.7 BIBLIOGRAPHICAL NOTES

19

11.8 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

11.9 ACKNOWLEDGEMENTS
12

Modeling running time

“When the measure of the problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of … the order of difficulty of [an] algorithm … is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes”, Jack Edmonds, “Paths, Trees, and Flowers”, 1963

“The computational complexity of a sequence is to be measured by how fast a multitape Turing machine can print out the terms of the sequence. This particular abstract model of a computing device is chosen because much of the work in this area is stimulated by the rapidly growing importance of computation through the use of digital computers, and all digital computers in a slightly idealized form belong to the class of multitape Turing machines.”, Juris Hartmanis and Richard Stearns, “On the computational complexity of algorithms”, 1963

In Chapter 11 we saw examples of efficient algorithms, and made some claims about their running time, but did not give a mathematically precise definition for this concept. We do so in this chapter, using the NAND++ and NAND« models we have seen before. Since we think of programs that can take as input a string of arbitrary length, their running time is not a fixed number but rather what we are interested in is measuring the dependence of the number of steps the program takes on the length of the input. That is, for any program $P$, we will be interested in the maximum number of steps that $P$ takes on inputs of length $n$ (which we often denote as $T(n)$). For example, if a function $F$ can be computed by a NAND« (or NAND++ program/-Turing machine) program that on inputs of length $n$ takes $O(n)$ steps

---

1 NAND++ programs are a variant of Turing machines, while NAND« programs are a way to model RAM machines, and hence all of the discussion in this chapter applies to those and many other models as well.

2 Because we are interested in the maximum number of steps for inputs of a given length, this concept is often known as worst case complexity. The minimum number of steps (or “best case” complexity) to compute a function on length $n$ inputs is typically not a meaningful quantity since essentially every natural problem will have some trivially easy instances. However, the average case complexity (i.e., complexity on a “typical” or “random” input) is an interesting concept which we’ll return to when we discuss cryptography. That said, worst-case complexity is the most standard and basic of the complexity measures, and will be our focus in most of this course.

Learning Objectives:
- Formally modeling running time, and in particular notions such as $O(n)$ or $O(n^2)$ time algorithms.
- The classes $P$ and $\text{EXP}$ modelling polynomial and exponential time respectively.
- The time hierarchy theorem, that in particular says that for every $k \geq 1$ there are functions we can compute in $O(n^{k+1})$ time but cannot compute in $O(n^k)$ time.
- The class $P_{\text{poly}}$ of non uniform computation and the result that $P \subseteq P_{\text{poly}}$.
then we will think of $F$ as “efficiently computable”, while if any such program requires $2^{O(n)}$ steps to compute $F$ then we consider $F$ “intractable”.

### 12.1 FORMALLY DEFINING RUNNING TIME

We start by defining running time separately for both NAND« and NAND++ programs. We will later see that the two measures are closely related. Roughly speaking, we will say that a function $F$ is computable in time $T(n)$ there exists a NAND« program that when given an input $x$, halts and outputs $F(x)$ within at most $T(|x|)$ steps. The formal definition is as follow:

**Definition 12.1 — Running time.** Let $T : \mathbb{N} \to \mathbb{N}$ be some function mapping natural numbers to natural numbers. We say that a function $F : \{0, 1\}^* \to \{0, 1\}$ is *computable in $T(n)$ NAND« time* if there exists a NAND« program $P$ such that for every every sufficiently large $n$ and every $x \in \{0, 1\}^n$, when given input $x$, the program $P$ halts after executing at most $T(n)$ lines and outputs $F(x)$. 

Similarly, we say that $F$ is *computable in $T(n)$ NAND++ time* if there is a NAND++ program $P$ computing $F$ such that on every sufficiently large $n$ and $x \in \{0, 1\}^n$, on input $x$, $P$ executes at most $T(n)$ lines before it halts with the output $F(x)$.

We let $TIME_{\leq}(T(n))$ denote the set of Boolean functions that are computable in $T(n)$ NAND« time, and define $TIME_{\leq_+}(T(n))$ analogously. The set $TIME(T(n))$ (without any subscript) corresponds to $TIME_{\leq}(T(n))$.

---

**Definition 12.1** is not very complicated but is one of the most important definitions of this book. Please make sure to stop, re-read it, and make sure you understand it. Note that although they are defined using computational models, $TIME_{\leq}(T(n))$ and $TIME_{\leq_+}(T(n))$ are classes of functions, not of programs. If $P$ is a NAND++ program then a statement such as “$P$ is a member of $TIME_{\leq_+}(n^2)$” does not make sense.

In the definition of time complexity, we count the number of times a line is executed, not the number of lines in the program. For example, if a NAND++ program $P$ has 20 lines, and on some input $x$ it takes a 1,000 iterations of its loop before it halts, then the number of lines executed on this input is 20,000.

To make this count meaningful, we use the “vanilla” flavors of NAND++ and NAND«, “unpacking” any syntactic sugar. For example, if a NAND++ program $P$ contains a line with the syntactic sugar $f \leftarrow \ldots$
Unlike the notion of computability, the exact running time can be a function of the model we use. However, it turns out that if we only care about “coarse enough” resolution (as will most often be the case) then the choice of the model, whether it is NAND«, NAND++, or Turing or RAM machines of various flavors, does not matter. (This is known as the extended Church-Turing Thesis). Nevertheless, to be concrete, we will use NAND« programs as our “default” computational model for measuring time, which is why we say that $F$ is computable in $T(n)$ time without any qualifications, or write $TIME(T(n))$ without any subscript, we mean that this holds with respect to NAND« machines.

Why NAND«? Given that so far we have emphasized NAND++ as our “main” model of computation, the reader might wonder why we use NAND« as the default yardstick where running time is involved. As we will see, this choice does not make much difference, but NAND« or RAM Machines correspond more closely to the notion of running time as discussed in algorithms text or the practice of computer science.

12.1.1 Nice time bounds
The class $TIME(T(n))$ is defined in Definition 12.1 with respect to a time measure function $T(n)$. When considering time bounds, we will want to restrict our attention to “nice” bounds such as $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^{\sqrt{n}})$, $O(2^n)$, and so on. We do so to avoid pathological examples such as non-monotone functions (where the time to compute a function on inputs of size $n$ could be smaller than the time to compute it on shorter inputs) or other degenerate cases such as running time that is not sufficient to read the input or cases where the running time bound itself is hard to compute. Thus we make the following definition:

**Definition 12.2 — Nice functions.** A function $T : \mathbb{N} \to \mathbb{N}$ is a nice time bound function (or nice function for short) if:

1. $T(n) \geq n$
2. $T(n) \geq T(n')$ whenever $n \geq n'$
3. There is a NAND« program that on input numbers \( n, i \), given in binary, can compute in \( O(T(n)) \) steps the \( i \)-th bit of a prefix-free representation of \( T(n) \) (represented as a string in some prefix-free way).

All the “normal” time complexity bounds we encounter in applications such as \( T(n) = 100n \), \( T(n) = n^2 \log n \), \( T(n) = 2^{\sqrt{n}} \), etc. are “nice”. Hence from now on we will only care about the class \( \text{TIME}(T(n)) \) when \( T \) is a “nice” function. Condition 3. of Definition 12.2 means that we can compute the binary representation of \( T(n) \) in time which itself is roughly \( T(n) \). This condition is typically easily satisfied. For example, for arithmetic functions such as \( T(n) = n^3 \) or \( T(n) = \lfloor n \rfloor^{1.2} \log n \) we can typically compute the binary representation of \( T(n) \) much faster than that: we can do so in time which is polynomial in the number of bits in this representation. Since the number of bits is \( O(\log T(n)) \), any quantity that is polynomial in this number will be much smaller than \( T(n) \) for large enough \( n \).

The two main time complexity classes we will be interested in are the following:

- **Polynomial time**: A function \( F : \{0,1\}^* \to \{0,1\} \) is computable in polynomial time if it is in the class \( \text{P} = \cup_{c \in \{1,2,3,\ldots\}} \text{TIME}(n^c) \). That is, \( F \in \text{P} \) if there is an algorithm to compute \( F \) that runs in time at most polynomial (i.e., at most \( n^c \) for some constant \( c \)) in the length of the input.

- **Exponential time**: A function \( F : \{0,1\}^* \to \{0,1\} \) is computable in exponential time if it is in the class \( \text{EXP} = \cup_{c \in \{1,2,3,\ldots\}} \text{TIME}(2^{n^c}) \). That is, \( F \in \text{EXP} \) if there is an algorithm to compute \( F \) that runs in time at most exponential (i.e., at most \( 2^{n^c} \) for some constant \( c \)) in the length of the input.

In other words, these are defined as follows:

**Definition 12.3 — P and EXP.** Let \( F : \{0,1\}^* \to \{0,1\} \). We say that \( F \in \text{P} \) if there is a polynomial \( p : \mathbb{N} \to \mathbb{R} \) and a NAND« program \( P \) such that for every \( x \in \{0,1\}^* \), \( P(x) \) runs in at most \( p(|x|) \) steps and outputs \( F(x) \).

We say that \( F \in \text{EXP} \) if there is a polynomial \( p : \mathbb{N} \to \mathbb{R} \) and a NAND« program \( P \) such that for every \( x \in \{0,1\}^* \), \( P(x) \) runs in at most \( 2^{p(|x|)} \) steps and outputs \( F(x) \).

Please make sure you understand why Definition 12.3 and the bullets above define the same classes.
Since exponential time is much larger than polynomial time, \( P \subseteq EXP \). All of the problems we listed in Chapter 11 are in \( EXP \), but as we’ve seen, for some of them there are much better algorithms that demonstrate that they are in fact in the smaller class \( P \).

<table>
<thead>
<tr>
<th>( P )</th>
<th>( EXP ) (but not known to be in ( P ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>Longest Path</td>
</tr>
<tr>
<td>Min cut</td>
<td>Max cut</td>
</tr>
<tr>
<td>2SAT</td>
<td>3SAT</td>
</tr>
<tr>
<td>Linear eqs</td>
<td>Quad. eqs</td>
</tr>
<tr>
<td>Zerosum</td>
<td>Nash</td>
</tr>
<tr>
<td>Determinant</td>
<td>Permanent</td>
</tr>
<tr>
<td>Primality</td>
<td>Factoring</td>
</tr>
</tbody>
</table>

A table of the examples from Chapter 11. All these problems are in \( EXP \) but the only ones on the left column are currently known to be in \( P \) as well (i.e., they have a polynomial-time algorithm).

12.1.2 Non-boolean and partial functions (optional)

Most of the time we will restrict attention to computing functions that are total (i.e., defined on every input) and Boolean (i.e., have a single bit of output). However, Definition 12.1 naturally extends to non Boolean and to partial functions as well. We now describe this extension, although we will try to stick to Boolean total functions to the extent possible, so as to minimize the “cognitive overload” for the reader and amount of notation they need to keep in their head.

We will define \( TIME_{\leq}(T(n)) \) and \( TIME_{++}(T(n)) \) to be the generalization of \( TIME_{\leq}(T(n)) \) and \( TIME_{++}(T(n)) \) to functions that may be partial or have more than one bit of output, and define \( P \) and \( EXP \) similarly. Specifically the formal definition is as follows:

**Definition 12.4 — Time complexity for partial and non-Boolean functions.** Let \( T : \mathbb{N} \rightarrow \mathbb{N} \) be a nice function, and let \( F \) be a (possibly partial) function mapping \( \{0, 1\}^* \) to \( \{0, 1\}^* \). We say that \( F \) is in \( TIME_{\leq}(T(n)) \) (respectively \( TIME_{++}(T(n)) \)) if there exists a NAND\( \leq \) (respectively NAND++) program \( P \) such that for every sufficiently large \( n \) and \( x \in \{0, 1\}^n \) on which \( F \) is defined, on input \( x \) the program \( P \) halts after executing at most \( T(n) \) steps and outputs \( F(x) \).

We let \( TIME(T(n)) \) (without a subscript) denote the set \( TIME_{\leq}(T(n)) \) and let \( P = \cup_{c \in \{1,2,3,\ldots\}} TIME(nc) \) and \( EXP = \cup_{c \in \{1,2,3,\ldots\}} TIME(2^{nc}) \).

Strictly speaking, many of these problems correspond to non Boolean functions, but we will sometimes “abuse notation” and refer to non Boolean functions as belonging to \( P \) or \( EXP \). We can easily extend the definitions of these classes to non Boolean and partial functions. Also, for every non-Boolean function \( F : \{0, 1\}^* \rightarrow \{0, 1\}^* \), we can define a Boolean variant \( Bool(F) \) such that \( F \) can be computed in polynomial time if and only if \( Bool(F) \) is. See Exercise 12.2
12.2 EFFICIENT SIMULATION OF RAM MACHINES: NAND« VS NAND++

We have seen in Theorem 7.1 that for every NAND« program \( P \) there is a NAND++ program \( P' \) that computes the same function as \( P \). It turns out that the \( P' \) is not much slower than \( P \). That is, we can prove the following theorem:

**Theorem 12.5 — Efficient simulation of NAND« with NAND++**. There are absolute constants \( a, b \) such that for every function \( F \) and nice function \( T : \mathbb{N} \rightarrow \mathbb{N} \), if \( F \in \mathcal{O}(T(n)) \), then there is a NAND++ program \( P' \) that computes \( F \) in \( T'(n) = a \cdot T(n)^b \). That is, \( \mathcal{O}(T(n)) \subseteq \mathcal{O}(aT(n)^b) \).

The constant \( b \) can be easily shown to be at most five, and with more effort can be optimized further. Theorem 12.5 means that the definition of the classes \( \mathsf{P} \) and \( \mathsf{EXP} \) are robust to the choice of model, and will not make a difference whether we use NAND++ or NAND«.

The same proof also shows that Turing Machines can simulate NAND« programs (and hence RAM machines). In fact, similar results are known for many models including cellular automata, C/Python/-Javascript programs, parallel computers, and a great many other models, which justifies the choice of \( \mathsf{P} \) as capturing a technology-independent notion of tractability. As we discussed before, this equivalence between NAND++ and NAND« (as well as other models) allows us to pick our favorite model depending on the task at hand (i.e., “have our cake and eat it too”). When we want to design an algorithm, we can use the extra power and convenience afforded by NAND«. When we want to analyze a program or prove a negative result, we can restrict attention to NAND++ programs.

**Proof Idea**: The idea behind the proof is simple. It follows closely the proof of Theorem 7.1, where we have shown that every function \( F \) that is computable by a NAND« program \( P \) is computable by a NAND++ program \( P' \). To prove Theorem 12.5, we follow the exact same proof but just check that the overhead of the simulation of \( P \) by \( P' \) is polynomial. The proof has many details, but is not deep. It is therefore much more important that you understand the statement of this theorem than its proof. *

**Proof of Theorem 12.5**. As mentioned above, we follow the proof of Theorem 7.1 (simulation of NAND« programs using NAND++ programs) and use the exact same simulation, but with a more careful accounting of the number of steps that the simulation costs. Recall, that the simulation of NAND« works by “peeling off” features of
We will not provide the full details but will present the main ideas used in showing that every feature of NAND\textsuperscript{x} can be simulated by NAND++ with at most a polynomial overhead:

1. Recall that every NAND\textsuperscript{x} variable or array element can contain an integer between 0 and \(T\) where \(T\) is the number of lines that have been executed so far. Therefore if \(P\) is a NAND\textsuperscript{x} program that computes \(F\) in \(T(n)\) time, then on inputs of length \(n\), all integers used by \(P\) are of magnitude at most \(T(n)\). This means that the largest value \(i\) can ever reach is at most \(T(n)\) and so each one of \(P\)'s variables can be thought of as an array of at most \(T(n)\) indices, each of which holds a natural number of magnitude at most \(T(n)\), which can be represented using \(O(\log T(n))\) bits.

2. As in the proof of Theorem 7.1, we can think of the integer array \(\text{F\text{oo}}\) as a two dimensional bit array \(\text{F\text{oo\_2D}}\) (where \(\text{F\text{oo\_2D}}[i][j]\) is the \(j\)-th bit of \(\text{F\text{oo}}[i]\)) and then encode the latter as a one dimensional bit array \(\text{F\text{oo\_1D}}\) by mapping \(\text{F\text{oo\_2D}}[i][j]\) to \(\text{F\text{oo\_1D}}[\text{embed}(i,j)]\) where \(\text{embed} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}\) is some one-to-one function that embeds the two dimensional space \(\mathbb{N} \times \mathbb{N}\) into the one dimensional \(\mathbb{N}\). Specifically, if we use \(\text{embed}(i,j) = \frac{1}{2}(i+j)(i+j+1) + j\) as in Exercise 7.1, then we can see that if \(i,j \leq O(T(n))\), then \(\text{embed}(i,j) \leq O(T(n)^2)\). We also use the fact that \(\text{embed}\) is itself computable in polynomial time in the length of its input.

3. All the arithmetic operations on integers use the grade-school algorithms, that take time that is polynomial in the number of bits of the integers, which is \(\text{poly}(\log T(n))\) in our case.

4. In NAND++ one cannot access an array at an arbitrary location but rather only at the location of the index variable \(i\). Nevertheless, if \(\text{F\text{oo}}\) is an array that encodes some integer \(k \in \mathbb{N}\) (where \(k \leq T(n)\) in our case), then, as we’ve seen in the proof of Theorem 7.1, we can write NAND++ code that will set the index variable \(i\) to \(k\). Specifically, using enhanced NAND++ we can write a loop that will run \(k\) times (for example, by decrementing the integer represented by \(\text{F\text{oo}}\) until it reaches 0), to ensure that an array \(\text{Marker}\) that will equal to 0 in all coordinates except the \(k\)-th one. We can then decrement \(i\) until it reaches 0 and scan \(\text{Marker}\) until we reach the point that \(\text{Marker}[i] = 1\).

5. To transform the above from enhanced to standard (i.e., “vanilla”) NAND++, all that is left is to follow our proof of Theorem 6.7 and show we can simulate \(i++ = \text{foo}\) and \(i-- = \text{bar}\) in vanilla NAND++
using our “breadcrumbs” and “wait for the bus” techniques. To simulate $T$ steps of enhanced NAND++ we will need at most $O(T^2)$ steps of vanilla NAND++ (see Fig. 12.1). Indeed, suppose that the largest point that the index $i$ reached in the computation so far is $R$, and we are in the worst case where we are trying, for example, to increment $i$ while it is in a “decreasing” phase. Within at most $2R$ steps we will be back in the same position at an “increasing” phase. Using this argument we can see that in the worst case, if we need to simulate $T$ steps of enhanced NAND++ we will use $O(1 + 2 + \cdots + T) = O(T^2)$ steps of vanilla NAND++.

Together these observations imply that the simulation of $T$ steps of NAND« can be done in $O(T^n)$ steps of vanilla NAND++ for some constant $a$, i.e., time polynomial in $T$. (A rough accounting can show that this constant $a$ is at most five; a more careful analysis can improve it further though this does not matter much.)

**Figure 12.1**: The path an index variable takes in a NAND++ program. If we need to simulate $T$ steps of an enhanced NAND++ program using vanilla NAND++ then in the worst case we will need to wait at every time for the next time $i$ arrives at the same location, which will yield a cost of $O(1 + 2 + \cdots + T) = O(T^2)$ steps.

---

**Theorem 12.5** shows that the classes $P$ and $EXP$ are robust with respect to variation in the choice of the computational model. They are also robust with respect to our choice of the representation of the input. For example, whether we decide to represent graphs as adjacency matrices or adjacency lists will not make a difference as to whether a function on graphs is in $P$ or $EXP$. The reason is that changing from one representation to another at most squares the
size of the input, and a quantity is polynomial in $n$ if and only if it is polynomial in $n^2$.

More generally, for every function $F : \{0, 1\}^* \rightarrow \{0, 1\}$, the answer to the question of whether $F \in \text{P}$ (or whether $F \in \text{EXP}$) is unchanged by switching representations, as long as transforming one representation to the other can be done in polynomial time (which essentially holds for all reasonable representations).

### 12.3 EFFICIENT UNIVERSAL MACHINE: A NAND« INTERPRETER IN NAND«

We have seen in Theorem 8.1 the “universal program” or “interpreter” $U$ for NAND++. Examining that proof, and combining it with Theorem 12.5, we can see that the program $U$ has a polynomial overhead, in the sense that it can simulate $T$ steps of a given NAND++ (or NAND«) program $P$ on an input $x$ in $O(T^n)$ steps for some constant $a$. But in fact, by directly simulating NAND« programs, we can do better with only a constant multiplicative overhead.

**Theorem 12.6 — Efficient universality of NAND«.** There is a NAND« program $U$ that computes the partial function $\text{TIMEDEVAL} : \{0, 1\}^* \rightarrow \{0, 1\}^*$ defined as follows:

$$\text{TIMEDEVAL}(P, x, 1^T) = P(x)$$  \hspace{1cm} (12.1)

if $P$ is a valid representation of a NAND« program which produces an output on $x$ within at most $T$ steps. If $P$ does not produce an output within this time then $\text{TIMEDEVAL}$ outputs an encoding of a special fail symbol. Moreover, for every program $P$, the running time of $U$ on input $P, x, 1^T$ is $O(T)$. (The hidden constant in the $O$-notation can depend on the program $P$ but is at most polynomial in the length of $P$’s description as a string.).

**What does $1^T$ mean?** The function $\text{TIMEDEVAL}$ has a curious feature - its third input has the form $1^T$. We use this notation to indicate a string of $T$ ones. (For example, if we write $1^5$ we mean the string 11111 rather using this to mean the integer one to the fifth power, which is a cumbersome way to write one; it will be always clear from context whether a particular input is an integer or such a string.)

Why don’t we simply assume that the function $\text{TIMEDEVAL}$ gets the integer $T$ in the binary representation? The reason has to do with how we define time as a function of the input length. If we represent $T$ as a string using the binary basis, then
its length will be $O(\log T)$, which means that we could not say that a program that takes $O(T)$ steps to compute $\text{TIMEDEVAL}$ would actually be considered as running in time exponential in its input. Thus, when we want to allow programs to run in time that is polynomial (as opposed to logarithmic) in some parameter $m$, we will often provide them with input of the form $1^m$ (i.e., a string of ones of length $m$).

There is nothing deep about representing inputs this way: this is merely a convention. However, it is a widely used one, especially in cryptography, and so is worth getting familiar with.

Before reading the proof of Theorem 12.6, try to think how you would compute $\text{TIMEDEVAL}$ using your favorite programming language. That is, how you would write a program $\text{Timed\_Eval}(P, x, T)$ that gets a NAND$\times$ program $P$ (represented in some convenient form), a string $x$, and an integer $T$, and simulates $P$ for $T$ steps. You will likely find that your program requires $O(T)$ steps to perform this simulation. As in the case of Theorem 12.5, the proof of Theorem 12.6 is not very deep and it more important to understand its statement. If you understand how you would go about writing an interpreter for NAND$\times$ using a modern programming language such as Python, then you know everything you need to know about this theorem.

**Proof of Theorem 12.6.** To present a universal NAND$\times$ program in full we would need to describe a precise representation scheme, as well as the full NAND$\times$ instructions for the program. While this can be done, it is more important to focus on the main ideas, and so we just sketch the proof here. A specification of NAND$\times$ is given in the Appendix, and for the purposes of this simulation, we can simply use the representation of the code NAND$\times$ as an ASCII string.

The program $U$ gets as input a NAND$\times$ program $P$, an input $x$, and a time bound $T$ (given in the form $1^T$) and needs to simulate the execution of $P$ for $T$ steps. To do so, $U$ will do the following:

1. $U$ will maintain variables $\text{current\_line}$, and $\text{number\_steps}$ for the current line to be executed and the number of steps executed so far.

2. $U$ will scan the code of $P$ to find the number $t$ of unique variable names that $P$ uses. If we denote these names by $\text{var}_{i_0}, \ldots, \text{var}_{i_{t-1}}$ then $U$ maintains an array $\text{Var\_numbers}$ that contains a list of
pairs of the form \((\text{var}_s, s)\) for \(s \in [t]\). Using \(\text{Var\_numbers}\) we can translate the name of a variable to a number in \([t]\) that corresponds to its index.

3. \(U\) will maintain an array \(\text{Var\_values}\) that will contain the current values of all \(P\)'s variables. If the \(s\)-th variable of \(P\) is a scalar variable, then its value will be stored in location \(\text{Var\_values}[s]\). If it is an array variable then the value of its \(i\)-th element will be stored in location \(\text{Var\_values}[t \cdot i + s]\).

4. To simulate a single step of \(P\), the program \(U\) will recover the line corresponding to \(\text{line\_counter}\) and execute it. Since NAND« has a constant number of arithmetic operations, we can simulate choosing which operation to execute with a sequence of a constantly many if-then-else’s. When executing these operations, \(U\) will use the variable \(\text{step\_counter}\) that keeps track of the iteration counter of \(P\).

Simulating a single step of \(P\) will take \(O(|P|)\) steps for the program \(U\) where \(|P|\) is the length of the description of \(P\) as a string (which in particular is an upper bound on the number \(t\) of variable \(P\) uses). Hence the total running time will be \(O(|P|T)\) which is \(O(T)\) when suppressing constants that depend on the program \(P\).

To be a little more concrete, here is some “pseudocode” description of the program \(U\):\(^6\)

```python
def U(P, x, 1^T):
    t = \text{number\_variable\_identifiers}(P)  # number of distinct identifiers used in \(P\)
    \# if \(s\) in \([t]\) corresponds to scalar then
    \# \text{Var\_values}[s] is value of variable corresponding to \(s\).
    \# if \(s\) corresponds to array then
    \# \text{Var\_values}[t \cdot i + s] is value of variable corresponding to \(s\) at position \(i\)
```

\(^6\) We use Python-like syntax in this pseudocode, but it is not valid Python code.
def varid(name):
    # scan the array Var_numbers and
    # return the number between 0 and t-1
    ...

def get_scalar_value(name):
    return Var_values[varid(name)]

def get_array_value(name, i):
    return Var_values[t*i+varid(name)]

def set_scalar_value(name, val):
    Var_values[varid(name)] = val

def set_array_value(name, i, val):
    Var_values[t*i+varid(name)] = val

for i=0..|x|-1:
    set_array_value("X", i, x[i])
    set_array_value("Xvalid", i, 1)

current_line = 0
number_steps = 0

do {
    line = P[current_line] # extract current line of P
    # code to execute line
    # We use get/set procedures above to update vars
    ...
    # Update counters
    current_line = current_line + 1 (mod L)
    number_steps = number_steps + 1
}

} until get_scalar_value("loop")==0 or
    (number_steps >= T)

# Produce output:
if get_scalar_value("loop")==1: return "FAIL"
m = smallest m s.t.
    get_array_value("Yvalid", m)=0
return [get_array_value("Y", i) for i=0..m-1]

12.4 TIME HIERARCHY THEOREM

We have seen that there are uncomputable functions, but are there functions that can be computed, but only at an exorbitant cost? For example, is there a function that can be computed in time \(2^n\), but cannot be computed in time \(2^{0.9n}\)? It turns out that the answer is Yes:

**Theorem 12.7 — Time Hierarchy Theorem.** For every nice function \(T\), there is a function \(f : \{0,1\}^* \rightarrow \{0,1\}\) in \(TIME(T(n) \log n) \setminus TIME(T(n))\). \(^7\)

Note that in particular this means that \(P\) is strictly contained in \(\text{EXP}\).

**Proof Idea:** In the proof of Theorem 8.3 (the uncomputability of the Halting problem), we have shown that the function \(HALT\) cannot be computed in any finite time. An examination of the proof shows that it gives something stronger. Namely, the proof shows that if we fix our computational budget to be \(T\) steps, then not only we can’t distinguish between programs that halt and those that do not, but cannot even distinguish between programs that halt within at most \(T'\) steps and those that take more than that (where \(T'\) is some number depending on \(T\)). Therefore, the proof of Theorem 12.7 follows the ideas of the uncomputability of the halting problem, but again with a more careful accounting of the running time. \(^\star\)

If you fully understand the proof of Theorem 8.3, then reading the following proof should not be hard. If you don’t, then this is an excellent opportunity to review this reasoning.

**Proof of Theorem 12.7.** Recall the Halting function \(HALT : \{0,1\}^* \rightarrow \{0,1\}\) that was defined as follows: \(HALT(P, x)\) equals 1 for every program \(P\) and input \(x\) such that \(P\) halts on input \(x\), and is equal to 0 otherwise. We cannot use the Halting function of course, as it is uncomputable and hence not in \(TIME(T'(n))\) for any function \(T'\). However, we will use the following variant of it:

We define the **Bounded Halting** function \(HALT_b(P, x)\) to equal 1 for every NAND\(\ast\) program \(P\) such that \(|P| \leq \log \log |x|\), and such that \(P\) halts on the input \(x\) within \(100T(|x|)\) steps. \(HALT_b\) equals 0 on all other inputs. \(^8\)

Theorem 12.7 is an immediate consequence of the following two claims:

**Claim 1:** \(HALT_b \in TIME(T(n) \log n)\)

and

\(^7\)There is nothing special about \(\log n\), and we could have used any other efficiently computable function that tends to infinity with \(n\).

\(^8\)The constant 100 and the function \(\log \log n\) are rather arbitrary, and are chosen for convenience in this proof.
Claim 2: $\text{HALT}_T \notin \text{TIME}(T(n))$.

Please make sure you understand why indeed the theorem follows directly from the combination of these two claims. We now turn to proving them.

Proof of claim 1: We can easily check in linear time whether an input has the form $P, x$ where $|P| \leq \log \log |x|$. Since $T(\cdot)$ is a nice function, we can evaluate it in $O(T(n))$ time. Thus, we can perform the check above, compute $T(|P| + |x|)$ and use the universal NAND« program of Theorem 12.6 to evaluate $\text{HALT}_T$ in at most $\text{poly}(|P|)T(n)$ steps.\(^9\) Since $(\log \log n)^a = o(\log n)$ for every $a$, this will be smaller than $T(n) \log n$ for every sufficiently large $n$, hence completing the proof.

Proof of claim 2: This proof is the heart of Theorem 12.7, and is very reminiscent of the proof that $\text{HALT}$ is not computable. Assume, toward the sake of contradiction, that there is some NAND« program $P^*$ that computes $\text{HALT}_T(P, x)$ within $T(|P| + |x|)$ steps. We are going to show a contradiction by creating a program $\mathcal{F}$ and showing that under our assumptions, if $Q$ runs for less than $T(n)$ steps when given (a padded version of) its own code as input then it actually runs for more than $T(n)$ steps and vice versa. (It is worth re-reading the last sentence twice or thrice to make sure you understand this logic. It is very similar to the direct proof of the uncomputability of the halting problem where we obtained a contradiction by using an assumed “halting solver” to construct a program that, given its own code as input, halts if and only if it does not halt.)

We will define $Q$ to be the program that on input a string $z$ does the following:

1. If $z$ does not have the form $z = P1^m$ where $P$ represents a NAND« program and $|P| < 0.1 \log \log m$ then return 0.

2. Compute $b = P^*(P, z)$ (at a cost of at most $T(|P| + |z|)$ steps, under our assumptions).

3. If $b = 1$ then $Q$ goes into an infinite loop, otherwise it halts.

We chose $m$ sufficiently large so that $|Q| < 0.001 \log \log m$ where $|Q|$ denotes the length of the description of $Q$ as a string. We will reach a contradiction by splitting into cases according to whether or not $\text{HALT}_T(Q, Q1^m)$ equals 0 or 1.

On the one hand, if $\text{HALT}_T(Q, Q1^m) = 1$, then under our assumption that $P^*$ computes $\text{HALT}_T$, $Q$ will go into an infinite loop on input $z = Q1^m$, and hence in particular $Q$ does not halt within $100T(|Q| + m)$ steps on the input $z$. But this contradicts our assumption that $\text{HALT}_T(Q, Q1^m) = 1$.

\(^9\) Recall that we use $\text{poly}(m)$ to denote a quantity that is bounded by $am^b$ for some constants $a, b$ and every sufficiently large $m$. 
This means that it must hold that $HALT_T^*(Q, Q^{m+1}) = 0$. But in this case, since we assume $P^*$ computes $HALT_T$, $Q$ does not do anything in phase 3 of its computation, and so the only computation costs come in phases 1 and 2 of the computation. It is not hard to verify that Phase 1 can be done in linear and in fact less than $5|z|$ steps. Phase 2 involves executing $P^*$, which under our assumption requires $T(|Q| + m)$ steps. In total we can perform both phases in less than $10T(|Q| + m)$ in steps, which by definition means that $HALT_T(Q, Q^{m+1}) = 1$, but this is of course a contradiction. This completes the proof of Claim 2 and hence of Theorem 12.7.

The time hierarchy theorem tells us that there are functions we can compute in $O(n^{2})$ time but not $O(n)$, in $2^n$ time, but not $2\sqrt{n}$, etc.. In particular there are most definitely functions that we can compute in time $2^n$ but not $O(n)$. We have seen that we have no shortage of natural functions for which the best known algorithm requires roughly $2^n$ time, and that many people have invested significant effort in trying to improve that. However, unlike in the finite vs. infinite case, for all of the examples above at the moment we do not know how to rule out even an $O(n)$ time algorithm. We will however see that there is a single unproven conjecture that would imply such a result for most of these problems.

**Figure 12.2**: Some complexity classes and some of the functions we know (or conjecture) to be contained in them.

---

**R Time hierarchy for NAND++ and Turing machines**

The time hierarchy theorem relies on the existence of an efficient universal NAND++ program, as proven in Theorem 12.6. We have mentioned that other
models, such as NAND++ programs and Turing machines, are polynomially

12.5 UNROLLING THE LOOP: UNIFORM VS NON UNIFORM COMPUTATION

We have now seen two measures of “computation cost” for functions. For a finite function $G : \{0,1\}^n \rightarrow \{0,1\}^m$, we said that $G \in \text{SIZE}(T)$ if there is a $T$-line NAND program that computes $G$. We saw that every function mapping $\{0,1\}^n$ to $\{0,1\}^m$ can be computed using at most $O(m2^n)$ lines. For infinite functions $F : \{0,1\}^* \rightarrow \{0,1\}^*$, we can define the “complexity” by the smallest $T$ such that $F \in \text{TIME}(T(n))$.

Is there a relation between the two?

For simplicity, let us restrict attention to Boolean (i.e., single-bit output) functions $F : \{0,1\}^* \rightarrow \{0,1\}$. For every such function, define $F_n : \{0,1\}^n \rightarrow \{0,1\}$ to be the restriction of $F$ to inputs of size $n$. We have seen two ways to define that $F$ is computable within a roughly $T(n)$ amount of resources:

1. There is a single algorithm $P$ that computes $F$ within $T(n)$ steps on all inputs of length $n$. In such a case we say that $F$ is uniformly computable (or more often, simply “computable”) within $T(n)$ steps.

2. For every $n$, there is a $T(n)$ NAND program $Q_n$ that computes $F_n$. In such a case we say that $F$ has can be computed via a non uniform $T(n)$ bounded sequence of algorithms.

Unlike the first condition, where there is a single algorithm or “recipe” to compute $F$ on all possible inputs, in the second condition we allow the restriction $F_n$ to be computed by a completely different program $Q_n$ for every $n$. One can see that the second condition is much more relaxed, and hence we might expect that every function satisfying the first condition satisfies the second one as well (up to a small overhead in the bound $T(n)$). This indeed turns out to be the case:

**Theorem 12.8** — Nonuniform computation contains uniform computation. There is some $c \in \mathbb{N}$ s.t. for every nice $T : \mathbb{N} \rightarrow \mathbb{N}$ and $F : \{0,1\}^* \rightarrow \{0,1\}$ in $\text{TIME}_\oplus(T(n))$ and every sufficiently large $n \in \mathbb{N}$, $F_n$ is in $\text{SIZE}(cT(n))$.

**Proof Idea:** To prove **Theorem 12.8** we use the technique of “unraveling the loop”. That is, we can use “copy paste” to replace a program...
$P$ that uses a loop that iterates for at most $T$ times with a “loop free” program that has about $T$ times as many lines as $P$.

Let us give an example using C-like syntax. Suppose we had a program of the form:

```c
do {
    // some code
} while (loop==1)
```

and we had the guarantee that the program would iterate the loop for at most $4$ times before it breaks.

Then we could change it to an equivalent loop-free program of the following form:

```c
// some code
if (loop) { // some code }
if (loop) { // some code }
if (loop) { // some code }
if (loop) { // some code }
```

That is all there is to the proof of Theorem 12.8.

**Proof of Theorem 12.8.** The proof follows by the argument of “unraveling the loop”. If $P$ is a NAND++ program of $L$ lines and $T : \mathbb{N} \to \mathbb{N}$ is a function such that for every input $x \in \{0,1\}^n$, $P$ halts after executing at most $T(n)$ lines (and hence iterating at most $\lfloor T(n)/L \rfloor$ times) then we can obtain a NAND program $Q$ on $n$ inputs as follows:

```c
P{i<0}
IF (loop) P{i<1}
IF (loop) P{i<0}
IF (loop) P{i<1}
IF (loop) P{i<2}
IF (loop) P{i<1}
IF (loop) P{i<0}
IF (loop) P{i<-1}
...
IF (loop) P{i<-R}
```

where for every number $j$, we denote by $P[i<-j]$ the NAND program that is obtained by replacing all references of the form $P[i]$. 
The constant $4$ can be improved, but this does not really make much difference. (which are allowed in NAND++, but illegal in NAND that has no index variable $i$) with references of the form $\text{Foo}[i]$ (which are allowed in NAND++, but illegal in NAND that has no index variable $i$) with references of the form $\text{Foo}[i]$ (which are allowed in NAND, since $\text{Foo}$ is simply a number). Whenever we see a reference to the variable $X\text{valid}[i]$ in the program we will replace it with one or zero depending on whether $i < n$. Similarly, we will replace all references to $X[i]$ for $i \geq n$ with zero. (We can use our standard syntactic sugar to create the constant zero and one variables."

We simply repeat the lines of the form $\text{IF (loop)} P[i\leftarrow j]$ for $[T(n)/L] - 1$ times, replacing each time $j$ by $0, 1, 2, \ldots$ as in the definition of (standard or “vanilla”) NAND++ in Section 6.1.3. We replace $\text{IF}$ with the appropriate syntactic sugar, which will incur a multiplicative overhead of at most $4$ in the number of lines. After this replacement, each line of the form $\text{IF (loop)} P[i\leftarrow j]$ corresponds to at most $4L$ lines of standard sugar-free NAND. Thus the total cost is at most $4L \cdot \frac{T(n)}{E} \leq 4 \cdot T(n)$ lines.  

By combining Theorem 12.8 with Theorem 12.5, we get that if $U \in \mathcal{H}(\mathcal{K}(w))$ then there are some constants $\varepsilon, \varepsilon$ such that for every large enough $w$, $U \in \mathcal{K}(\varepsilon \mathcal{H}(w))$. (In fact, by direct inspection of the proofs we can see that $\varepsilon = \varepsilon = 5$ would work.)

12.5.1 Algorithmic transformation of NAND++ to NAND and “Proof by Python” (optional)

The proof of Theorem 12.8 is algorithmic, in the sense that the proof yields a polynomial-time algorithm that given a NAND++ program $P$ and parameters $T$ and $n$, produces a NAND program $Q$ of $O(T)$ lines that agrees with $P$ on all inputs $x \in \{0,1\}^n$ (as long as $P$ runs for less than $T$ steps these inputs.) Thus the same proof gives the following theorem:

**Theorem 12.9 — NAND++ to NAND compiler.** There is an $O(n)$-time NAND$^*$ program $\text{COMPILE}$ such that on input a NAND++ program $P$, and strings of the form $1^n, 1^m, 1^T$ outputs a NAND program $Q_P$ of at most $O(T)$ lines with $n$ bits of inputs and $m$ bits of output satisfying the following property.

For every $x \in \{0,1\}^n$, if $P$ halts on input $x$ within fewer than $T$ steps and outputs some string $y \in \{0,1\}^m$, then $Q_P(x) = y$.

We omit the proof of the Theorem 12.9 since it follows in a fairly straightforward way from the proof of Theorem 12.8. However, for the sake of concreteness, here is a Python implementation of the function $\text{COMPILE}$. (The reader can feel free to skip it.)

For starters, let us consider an imperfect but very simple program that unrolls the loop. The following program will work correctly for the case that $m = 1$ and that the underlying NAND++ program had
def COMPILE(P, T, n):
    '''
    Gets P = NAND++ program
    T - time bound, n - number of inputs, single
    output
    Produces NAND program of T lines that computes
    the restriction of P to inputs of length n and T
    steps.

    assumes program contains "one" and "zero"
    variables and that Y[0] is never modified after
    the correct value is
    written, so it is safe to run for an additional
    number of loops.
    '''
    numlines = P.count("\n")

    result = ""
    for t in range(T // numlines):
        i = index(t)  # value of i in T-th iteration
        Xvalid_i = ('one' if i < n else 'zero')
        X_i = ('X[i]' if i < n else 'zero')
        Q =
            P.replace('Validx[i]', Xvalid_i).replace('X[i]', X_i)
        result += Q.replace('[i]', f'[{i}]')
    return result

The index function takes a number t and returns the value of
the index variable i in the t-th iteration. Recall that this value in
NAND++ follows the sequence 0, 1, 0, 1, 2, 1, 0, 1, 2, ... and it can be
computed in Python as follows:

from math import sqrt
def index(k):
    return min([abs(k-int(r))*(int(r)+1)) for r in
            [sqrt(k)-0.5, sqrt(k)+0.5]])

Below is a more “robust” implementation of COMPILE, that works
for an arbitrarily large number of outputs, and makes no assumptions
on the structure of the input program.

def COMPILE(P, T, n, m):
Gets $P = \text{NAND PP program}$
$T$ - time bound, $n$ - number of inputs, $m$ - number of outputs

Produces $\text{NAND program of } O(T)$ lines that computes the restriction of $P$ to inputs of length $n$ and $T$ steps

```python
lines = [l for l in P.split(\'\n\') if l] # lines of $P$

# initialization
result = r'''temp = \text{NAND}(X[0],X[0])
one = \text{NAND}(X[0],temp)
zero = \text{NAND}(one,one)
nothalted = \text{NAND}(X[0],temp)
halted = \text{NAND}(one,one)
'''

# assign_to = IF(halted,assign_to,new_value)
IFCODE = r'''
iftemp_0 = \text{NAND}(new_value,nothalted)
iftemp_1 = \text{NAND}(assign_to,halted)
assign_to = \text{NAND}(iftemp_0,iftemp_1)
'''

UPDATEHALTED = r'''halted = \text{NAND}(nothalted,loop)
nothalted = \text{NAND}(halted,halted)
'''

for t in range($T$ // len(lines)):
    j = index(t)
    for line in lines:
        if j >= m:
            line = line.replace('Y[i]', 'temp')
        if j < n:
            line =
            line.replace('Xvalid[i]', 'one')
        else:
            line =
            line.replace('Xvalid[i]', 'zero')
        line = line.replace('X[i]', 'zero')```
Since NAND\textsuperscript{\textless} programs can be simulated by NAND++ programs with polynomial overhead, we see that we can simulate a $T(n)$ time NAND\textsuperscript{\textless} program on length \( n \) inputs with a $\mathit{poly}(T(n))$ size NAND program.

12.5.2 The class \( P_{/\text{poly}} \)

We can define the “non uniform” analog of the class \( P \) as follows:

**Definition 12.10** $\rightarrow P_{/\text{poly}}$. For every \( F : \{0,1\}^* \rightarrow \{0,1\} \), we say that \( F \in P_{/\text{poly}} \) if there is some polynomial \( p : \mathbb{N} \rightarrow \mathbb{R} \) such that for every \( n \in \mathbb{N}, F_n \in \mathit{SIZE}(p(n)) \) where \( F_n \) is the restriction of \( F \) to inputs in \( \{0,1\}^n \).

Theorem 12.8 implies that \( P \subseteq P_{/\text{poly}} \).
that the restriction of $F$ to $\{0,1\}^n$ is computable by a Boolean circuit of $\text{poly}(n)$ size (say with gates in the set $\land, \lor, \neg$ though any universal gateset will do); see Fig. 12.3.

The notation $P_{/\text{poly}}$ is used for historical reasons. It was introduced by Karp and Lipton, who considered this class as corresponding to functions that can be computed by polynomial-time Turing Machines (or equivalently, NAND++ programs) that are given for any input length $n$ a polynomial in $n$ long advice string. That this is an equivalent characterization is shown in the following theorem:

**Theorem 12.11 — $P_{/\text{poly}}$ characterization by advice.** Let $F : \{0,1\}^* \to \{0,1\}$. Then $F \in P_{/\text{poly}}$ if and only if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$, a polynomial-time NAND++ program $P$ and a sequence $\{a_n\}_{n \in \mathbb{N}}$ of strings, such that for every $n \in \mathbb{N}$:

- $|a_n| \leq p(n)$
- For every $x \in \{0,1\}^n$, $P(a_n, x) = F(x)$.

**Proof.** We only sketch the proof. For the “only if” direction, if $F \in P_{/\text{poly}}$ then we can use for $a_n$ simply the description of the corresponding NAND program $Q_n$, and for $P$ the program that computes in polynomial time the $\text{NANDEVAL}$ function that on input an $n$-input NAND program $Q$ and a string $x \in \{0,1\}^n$, outputs $Q(n) >$

For the “if” direction, we can use the same “unrolling the loop” technique of Theorem 12.8 to show that if $P$ is a polynomial-time NAND++ program, then for every $n \in \mathbb{N}$, the map $x \mapsto P(a_n, x)$ can be
computed by a polynomial size NAND program $Q_n$.

To make sure you understand the definition of $\mathbf{P}/\mathbf{poly}$, I highly encourage you to work out fully the details of the proof of Theorem 12.11.

### 12.5.3 Simulating NAND with NAND++?

**Theorem 12.8** shows that every function in $\text{TIME}(T(n))$ is in $\text{SIZE}(\text{poly}(T(n)))$. One can ask if there is an inverse relation. Suppose that $F$ is such that $F_n$ has a “short” NAND program for every $n$. Can we say that it must be in $\text{TIME}(T(n))$ for some “small” $T$? The answer is an emphatic no. Not only is $\mathbf{P}/\mathbf{poly}$ not contained in $\mathbf{P}$, in fact $\mathbf{P}/\mathbf{poly}$ contains functions that are uncomputable!

**Theorem 12.12** — $\mathbf{P}/\mathbf{poly}$ contains uncomputable functions. There exists an uncomputable function $F : \{0,1\}^* \rightarrow \{0,1\}$ such that $F \in \mathbf{P}/\mathbf{poly}$.

**Proof Idea:** Since $\mathbf{P}/\mathbf{poly}$ corresponds to non uniform computation, a function $F$ is in $\mathbf{P}/\mathbf{poly}$ if for every $n \in \mathbb{N}$, the restriction $F_n$ to inputs of length $n$ has a small circuit/program, even if the circuits for different values of $n$ are completely different from one another. In particular, if $F$ has the property that for every equal-length inputs $x$ and $x'$, $F(x) = F(x')$ then this means that $F_n$ is either the constant function zero or the constant function one for every $n \in \mathbb{N}$. Since the constant function has a (very!) small circuit, such a function $F$ will always be in $\mathbf{P}/\mathbf{poly}$ (indeed even in smaller classes). Yet by a reduction from the Halting problem, we can obtain a function with this property that is uncomputable. ⋆

**Proof of ??**. Consider the following “unary halting function” $UH : \{0,1\}^* \rightarrow \{0,1\}$ defined as follows. We let $S : \mathbb{N} \rightarrow \{0,1\}^*$ be the function that on input $n \in \mathbb{N}$, outputs the string that corresponds to the binary representation of the number $n$ without the most significant 1 digit. Note that $S$ is onto. For every $x \in \{0,1\}$, we define $UH(x) = \text{HALTONZERO}(S(|x|))$. That is, if $n$ is the length of $x$, then $UH(x) = 1$ if and only if the string $S(n)$ encodes a NAND++ program that halts on the input 0.

$UH$ is uncomputable, since otherwise we could compute $\text{HALTONZERO}$ by transforming the input program $P$ into the integer $n$ such that $P = S(n)$ and then running $UH(1^n)$ (i.e., $UH$ on the string of $n$ ones). On the other hand, for every $n$, $UH_n(x)$ is either equal to 0 for all inputs $x$ or equal to 1 on all inputs $x$, and
hence can be computed by a NAND program of a constant number of lines.

The issue here is of course uniformity. For a function \( F : \{0,1\}^* \to \{0,1\} \), if \( F \) is in \( TIME(T(n)) \) then we have a single algorithm that can compute \( F_n \) for every \( n \). On the other hand, \( F_n \) might be in \( SIZE(T(n)) \) for every \( n \) using a completely different algorithm for every input length. For this reason we typically use \( \text{P}/\text{poly} \) not as a model of efficient computation but rather as a way to model inefficient computation. For example, in cryptography people often define an encryption scheme to be secure if breaking it for a key of length \( n \) requires more then a polynomial number of NAND lines. Since \( \text{P} \subseteq \text{P}/\text{poly} \), this in particular precludes a polynomial time algorithm for doing so, but there are technical reasons why working in a non uniform model makes more sense in cryptography. It also allows to talk about security in non asymptotic terms such as a scheme having “128 bits of security”.

### Non uniformity in practice

While it can sometimes be a real issue, in many natural settings the difference between uniform and non-uniform computation does not seem to so important. In particular, in all the examples of problems not known to be in \( \text{P} \) we discussed before: longest path, 3SAT, factoring, etc., these problems are also not known to be in \( \text{P}/\text{poly} \) either. Thus, for “natural” functions, if you pretend that \( TIME(T(n)) \) is roughly the same as \( SIZE(T(n)) \), you will be right more often than wrong.

### 12.5.4 Uniform vs. Nonuniform computation: A recap

To summarize, the two models of computation we have described so far are:

- **NAND programs**, which have no loops, can only compute finite functions, and the time to execute them is exactly the number of lines they contain. These are also known as *straightline programs* or *Boolean circuits*.

- **NAND++ programs**, which include loops, and hence a single program can compute a function with unbounded input length. These are equivalent (up to polynomial factors) to Turing Machines or (up to polylogarithmic factors) to RAM machines.

For a function \( F : \{0,1\}^* \to \{0,1\} \) and some nice time bound \( T : \mathbb{N} \to \mathbb{N} \), we know that:
Large scale quantum computers have not yet been built, and even if they are constructed, we have no proof that they would offer super polynomial advantage over “classical” computing devices. However, there seems to be no fundamental physical obstacle to constructing them, and there are strong reasons to conjecture that they do in fact offer such an advantage.

12.6 EXTENDED CHURCH-TURING THESIS

We have mentioned the Church-Turing thesis, that posits that the definition of computable functions using NAND++ programs captures the definition that would be obtained by all physically realizable computing devices. The extended Church Turing is the statement that the same holds for efficiently computable functions, which is typically interpreted as saying that NAND++ programs can simulate every physically realizable computing device with polynomial overhead.

In other words, the extended Church Turing thesis says that for every scalable computing device C (which has a finite description but can be in principle used to run computation on arbitrarily large inputs), there are some constants a, b such that for every function F : \{0, 1\}^n \to \{0, 1\} that C can compute on n length inputs using an S(n) amount of physical resources, F is in TIME(aS(n)^b).

All the current constructions of scalable computational models and programming language conform to the Extended Church-Turing Thesis, in the sense that they can be with polynomial overhead by Turing Machines (and hence also by NAND++ or NAND« programs). Consequently, the classes P and EXP are robust to the choice of model, and we can use the programming language of our choice, or high level descriptions of an algorithm, to determine whether or not a problem is in P.

Like the Church-Turing thesis itself, the extended Church-Turing thesis is in the asymptotic setting and does not directly yield an experimentally testable prediction. However, it can be instantiated with more concrete bounds on the overhead, which would yield predictions such as the Physical Extended Church-Turing Thesis we mentioned before, which would be experimentally testable.

In the last hundred+ years of studying and mechanizing computation, no one has yet constructed a scalable computing device (or even gave a convincing blueprint) that violates the extended Church Turing Thesis. However, quantum computing, if realized, will pose a serious challenge to this thesis.\(^{11}\) However, even if the promises of quantum computing are fully realized, the extended Church-Turing thesis is
“morally” correct, in the sense that, while we do need to adapt the thesis to account for the possibility of quantum computing, its broad outline remains unchanged. We are still able to model computation mathematically, we can still treat programs as strings and have a universal program, and we still have hierarchy and uncomputability results. Moreover, for most (though not all!) concrete problems we care about, the prospect of quantum computing does not seem to change their time complexity. In particular, out of all the example problems mentioned in Chapter 11, as far as we know, the complexity of only one—integer factoring—is affected by modifying our model to include quantum computers as well.

\[12\] Quantum computing is not a challenge to the (non extended) Church Turing thesis, as a function is computable by a quantum computer if and only if it is computable by a “classical” computer or a NAND++ program. It is only the running time of computing the function that can be affected by moving to the quantum model.

**Lecture Recap**

- We can define the time complexity of a function using NAND++ programs, and similarly to the notion of computability, this appears to capture the inherent complexity of the function.
- There are many natural problems that have polynomial-time algorithms, and other natural problems that we’d love to solve, but for which the best known algorithms are exponential.
- The definition of polynomial time, and hence the class \(P\), is robust to the choice of model, whether it is Turing machines, NAND++, NAND«, modern programming languages, and many other models.
- The time hierarchy theorem shows that there are some problems that can be solved in exponential, but not in polynomial time. However, we do not know if that is the case for the natural examples that we described in this lecture.
- By “unrolling the loop” we can show that every function computable in time \(T(n)\) can be computed by a sequence of NAND programs (one for every input length) each of size at most \(\text{poly}(T(n))\).
Exercise 12.1 — Equivalence of different definitions of P and EXP. Prove that the classes $P$ and $EXP$ defined in Definition 12.3 are equal to $\bigcup_{c \in \{1,2,3,\ldots\}} TIME(n^c)$ and $\bigcup_{c \in \{1,2,3,\ldots\}} TIME(2^{n^c})$ respectively. If $S_1, S_2, S_3, \ldots$ is a collection of sets then the set $S = \bigcup_{c \in \{1,2,3,\ldots\}} S_c$ is the set of all elements $e$ such that there exists some $c \in \{1,2,3,\ldots\}$ where $e \in S_c$.

Exercise 12.2 — Boolean functions. For every function $F : \{0,1\}^* \rightarrow \{0,1\}^*$, define $Bool(F)$ to be the function mapping $\{0,1\}^*$ to $\{0,1\}$ such that on input a (string representation of a) triple $(x, i, \sigma)$ with $x \in \{0,1\}^*$, $i \in \mathbb{N}$ and $\sigma \in \{0,1\}$,

$$
Bool(F)(x, i, \sigma) = \begin{cases} 
F(x)_i & \sigma = 0, i < |F(x)| \\
1 & \sigma = 1, i < |F(x)| \\
0 & \text{otherwise}
\end{cases}
$$

(12.2)

where $F(x)_i$ is the $i$-th bit of the string $F(x)$.

Prove that $F \in \overline{P}$ if and only if $Bool(F) \in P$.

Exercise 12.3 — Composition of polynomial time. Prove that if $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$ are in $\overline{P}$ then their composition $F \circ G$, which is the function $H$ s.t. $H(x) = F(G(x))$, is also in $\overline{P}$.

Exercise 12.4 — Non composition of exponential time. Prove that there is some $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$ s.t. $F, G \in EXP$ but $F \circ G$ is not in $EXP$.

Exercise 12.5 — Oblivious program. We say that a NAND++ program $P$ is oblivious if there is some functions $T : \mathbb{N} \rightarrow \mathbb{N}$ and $i : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ such that for every input $x$ of length $n$, it holds that:

* $P$ halts when given input $x$ after exactly $T(n)$ steps.

* For $t \in \{1, \ldots, T(n)\}$, after $P$ executes the $t^{th}$ step of the execution the value of the index $i$ is equal to $t(n, i)$. In particular this value does not depend on $x$ but only on its length.

13 TODO: check that this works, idea is that we can do bounded halting.

14 An oblivious program $P$ cannot compute functions whose output length is not a function of the input length, though this is not a real restriction, as we can always embed variable output functions in fixed length ones using some special “end of output” marker.
Exercise 12.6 — Alternative characterization of \( P \). Prove that for every \( F : \{0, 1\}^* \rightarrow \{0, 1\}, F \in P \) if and only if there exists a polynomial time NAND++ program \( P \) such that \( P(1^n) \) outputs a NAND program \( Q_n \) that computes the restriction of \( F \) to \( \{0, 1\}^n \).

12.8 BIBLIOGRAPHICAL NOTES

12.9 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

12.10 ACKNOWLEDGEMENTS
13

Polynomial-time reductions

Let us consider several of the problems we have encountered before:

• Finding the longest path in a graph
• Finding the maximum cut in a graph
• The 3SAT problem: deciding whether a given 3CNF formula has a satisfying assignment.
• Solving quadratic equations over \( n \) variables \( x_0, \ldots , x_{n-1} \in \mathbb{R} \).

All of these have the following properties:

• These are important problems, and people have spent significant effort on trying to find better algorithms for them.
• Each one of these problems is a search problem, whereby we search for a solution that is “good” in some easy to define sense (e.g., a long path, a satisfying assignment, etc.).
• Each of these problems has trivial exponential time algorithms that involve enumerating all possible solutions.
• At the moment, for all these problems the best known algorithms are not much better than the trivial one in the worst case.

In this chapter and in Chapter 14 we will see that, despite their apparent differences, we can relate these problems by their complexity. In fact, it turns out that all these problems are computationally equivalent, in the sense that solving one of them immediately implies solving the others. This phenomenon, known as \textbf{NP completeness}, is one of the surprising discoveries of theoretical computer science, and we will see that it has far-reaching ramifications.

Learning Objectives:
• Introduce the notion of polynomial-time reductions as a way to relate the complexity of problems to one another.
• See several examples of such reductions.
• 3SAT as a basic starting point for reductions.
**13.0.1 Decision problems**

For reasons of technical conditions rather than anything substantial, we will concern ourselves with decision problems (i.e., Yes/No questions) or in other words Boolean (i.e., one-bit output) functions. Thus, we will model all the problems as functions mapping \( \{0, 1\}^* \) to \( \{0, 1\} \):

- The **3SAT** problem can be phrased as the function \( 3SAT : \{0, 1\}^* \to \{0, 1\} \) that maps a 3CNF formula \( \varphi \) to 1 if there exists some assignment \( x \) that satisfies it, and to 0 otherwise.\(^1\)

- The **quadratic equations** problem corresponds to the function \( QUADEQ : \{0, 1\}^* \to \{0, 1\} \) that maps a set of quadratic equations \( E \) to 1 if there is an assignment \( x \) that satisfies all equations, and to 0 otherwise.

- The **longest path** problem corresponds to the function \( LONGPATH : \{0, 1\}^* \to \{0, 1\} \) that maps a graph \( G \) and a number \( k \) to 1 if there is a simple\(^2\) path in \( G \) of length at least \( k \), and maps \( (G, k) \) to 0 otherwise. The longest path problem is a generalization of the well-known Hamiltonian Path Problem of determining whether a path of length \( n \) exists in a given \( n \) vertex graph.

- The **maximum cut** problem corresponds to the function \( MAXCUT : \{0, 1\}^* \to \{0, 1\} \) that maps a graph \( G \) and a number \( k \) to 1 if there is a cut in \( G \) that cuts at least \( k \) edges, and maps \( (G, k) \) to 0 otherwise.

**13.1 REDUCTIONS**

Suppose that \( F, G : \{0, 1\}^* \to \{0, 1\} \) are two functions. How can we show that they are “computationally equivalent”? The idea is that we show that an efficient algorithm for \( F \) would imply an efficient algorithm for \( G \) and vice versa. The key to this is the notion of a reduction. Roughly speaking, we will say that \( F \) reduces to \( G \) (denoted as \( F \leq_p G \)) if \( F \) is “no harder” than \( G \), in the sense that a polynomial-time algorithm for \( G \) implies a polynomial-time algorithm for \( F \). The formal definition is as follows:\(^3\)

**Definition 13.1 — Reductions.** Let \( F, G : \{0, 1\}^* \to \{0, 1\}^* \). We say that \( F \) reduces to \( G \), denoted by \( F \leq_p G \), if there is a polynomial-time computable \( R : \{0, 1\}^* \to \{0, 1\}^* \) such that for every \( x \in \{0, 1\}^* \),

\[
F(x) = G(R(x)).
\]

We say that \( F \) and \( G \) have equivalent complexity if \( F \leq_p G \) and \( G \leq_p F \).

\(^1\) We assume some representation of formulas as strings, and define the function to output 0 if its input is not a valid representation. We will use the same convention for all the other functions below.

\(^2\) Recall that a simple path in a graph is one that does not visit any vertex more than once. For the shortest path problem we can assume that a path is simple without loss of generality since removing a loop (a portion of the path that starts from the same vertex and returns to it) only makes the path shorter. For the longest path problem we need to make this restriction to avoid “degenerate” paths such as paths that repeat endlessly the same loop.

\(^3\) Several notions of reductions are defined in the literature. The notion defined in Definition 13.1 is often known as a mapping reduction, many to one reduction or a Karp reduction.
Figure 13.1: If $F \leq_p G$ then we can transform a polynomial-time algorithm $B$ that computes $G$ into a polynomial-time algorithm $A$ that computes $F$. To compute $F(x)$ we can run the reduction $R$ guaranteed by the fact that $F \leq_p G$ to obtain $y = F(x)$ and then run our algorithm $B$ for $G$ to compute $G(y)$.

Solved Exercise 13.1 — Reductions and $P$. Prove that if $F \leq_p G$ and $G \in P$ then $F \in P$.

Solution: Suppose there was an algorithm $B$ that compute $F$ in time $p(n)$ where $p$ is its input size. Then, Eq. (13.1) directly gives an algorithm $A$ to compute $F$ (see Fig. 13.1). Indeed, on input $x \in \{0,1\}^*$, Algorithm $A$ will run the polynomial-time reduction $R$ to obtain $y = R(x)$ and then return $B(y)$. By Eq. (13.1), $G(R(x)) = F(x)$ and hence Algorithm $A$ will indeed compute $F$.

We now show that $A$ runs in polynomial time. By assumption, $R$ can be computed in time $q(n)$ for some polynomial $q$. In particular, this means that $|y| \leq q(|x|)$ (as just writing down $y$ takes $|y|$ steps). Thus, computing $B(y)$ will take at most $p(|y|) \leq p(q(|x|))$ steps. Thus the total running time of $A$ on inputs of length $n$ is at most the time to compute $y$, which is bounded by $q(n)$, and the time to compute $B(y)$, which is bounded by $p(q(n))$, and since the composition of two polynomials is a polynomial, $A$ runs in polynomial time.

Since we think of $F \leq_p G$ as saying that (as far as polynomial-time computation is concerned) $F$ is “easier or equal in difficulty to” $G$, we would expect that if $F \leq_p G$ and $G \leq_p H$, then it would hold that $F \leq_p H$. Indeed this is the case:

Lemma 13.2 For every $F, G, H : \{0,1\}^* \to \{0,1\}$, if $F \leq_p G$ and
\[ G \leq_p H \text{ then } F \leq_p H. \]

We leave the proof of Lemma 13.2 as Exercise 13.2. Pausing now and doing this exercise is an excellent way to verify that you understood the definition of reductions.

Polynomial reductions, completeness and soundness  We have seen reductions before in the context of proving the uncomputability of problems such as HALTONZERO and others. The most crucial difference between the notion in Definition 13.1 and previously occurring notions is that in the context of relating the time complexity of problems, we need the reduction to be computable in polynomial time, as opposed to merely computable. Definition 13.1 also restricts reductions to have a very specific format. That is, to show that \( F \leq_p G \), rather than allowing a general algorithm for \( F \) that uses a “magic box” that computes \( G \), we only allow an algorithm that computes \( F(x) \) by outputting \( G(R(x)) \). This restricted form is convenient for us, but people have defined and used more general reductions as well.

Since both \( F \) and \( G \) are Boolean functions, the condition \( F(x) = G(R(x)) \) in Eq. (13.1) is equivalent to the following two implications: (i) if \( F(x) = 1 \) then \( G(R(x)) = 1 \), and (ii) if \( G(R(x)) = 1 \) then \( F(x) = 1 \). Traditionally, condition (i) is known as completeness and condition (ii) is known as soundness. We can think of this as saying that the reduction \( R \) is complete if every 1-input of \( F \) (i.e. \( x \) such that \( F(x) = 1 \)) is mapped by \( R \) to a 1-input of \( G \), and that it is sound if no 0-input of \( F \) will ever be mapped to a 1-input of \( G \). As we will see below, it is often the case that establishing (ii) is the more challenging part.

13.2 SOME EXAMPLE REDUCTIONS

We will now use reductions to relate the computational complexity of the problems mentioned above – 3SAT, Quadratic Equations, Maximum Cut, and Longest Path. We start by reducing 3SAT to the latter three problems, demonstrating that solving any one of them will solve 3SAT. Along the way we will introduce one more problem: the independent set problem. Like the others, it shares the characteristics that it is an important and well-motivated computational problem, and that the best known algorithm for it takes exponential time. In Chapter 14 we will show the other direction: reducing each one of these problems to 3SAT in one fell swoop.
Figure 13.2: Our first stage in showing equivalence is to reduce 3SAT to the three other problems.

13.2.1 Reducing 3SAT to quadratic equations

Let us now see our first example of a reduction. Recall that in the quadratic equation problem, the input is a list of $n$-variate polynomials $p_0, \ldots, p_{m-1}: \mathbb{R}^n \to \mathbb{R}$ that are all of degree at most two (i.e., they are quadratic) and with integer coefficients. The task is to find out whether there is a solution $x \in \mathbb{R}^n$ to the equations $p_0(x) = 0, p_1(x) = 0, \ldots, p_{m-1}(x) = 0$.

For example, the following is a set of quadratic equations over the variables $x_0, x_1, x_2$:

\begin{align*}
x_0^2 - x_0 &= 0 \\
x_1^2 - x_1 &= 0 \\
x_2^2 - x_2 &= 0 \\
1 - x_0 - x_1 + x_0 x_1 &= 0
\end{align*}

You can verify that $x \in \mathbb{R}^3$ satisfies this set of equations if and only if $x \in \{0, 1\}^3$ and $x_0 \lor x_1 = 1$.

We will show how to reduce 3SAT to the problem of Quadratic Equations.

\begin{theorem} — Hardness of quadratic equations. \end{theorem}

\begin{equation}
3SAT \leq_p QUADEQ \tag{13.3}
\end{equation}

where 3SAT is the function that maps a 3SAT formula $\varphi$ to 1 if it is satisfiable and to 0 otherwise, and QUADEQ is the function that maps a set $E$ of quadratic equations over $\{0, 1\}^n$ to 1 if it has a solution and to 0 otherwise.

**Proof Idea:** At the end of the day, a 3SAT formula can be thought of as a list of equations on some variables $x_0, \ldots, x_{n-1}$. Namely, the equations are that each of the $x_i$’s should be equal to either 0 or 1, and that the variables should satisfy some set of constraints which corresponds to the OR of three variables or their negation. To show that
3SAT \leq_p QUADEQ we need to give a polynomial-time reduction that maps a 3SAT formula \( \varphi \) into a set of quadratic equations \( E \) such that \( E \) has a solution if and only if \( \varphi \) is satisfiable. The idea is that we can transform a 3SAT formula \( \varphi \) first to a set of cubic equations by mapping every constraint of the form \((x_{12} \lor x_{15} \lor x_{24}) \land \cdots \) into an equation of the form \((1 - x_{12})x_{15}(1 - x_{24}) = 0\). We can then turn this into a quadratic equation by mapping any cubic equation of the form \( x_{i,j}x_k = 0 \) into the two quadratic equations \( y_{i,j} = x_i x_j \) and \( y_{i,j}x_k = 0 \).

\[
\begin{aligned}
\text{Proof of Theorem 13.3.} & \quad \text{To prove Theorem 13.3 we need to give a polynomial-time transformation of every 3SAT formula \( \varphi \) into a set of quadratic equations \( E \), and prove that \( 3\text{SAT}(\varphi) = \text{QUADEQ}(E) \).}

& \quad \text{We now describe the transformation of a formula \( \varphi \) to equations \( E \) and show the completeness and soundness conditions. Recall that a 3SAT formula \( \varphi \) is a formula such as \((x_{17} \lor x_{101} \lor x_{57}) \land (x_{18} \lor \ldots \land \cdots) \). That is, \( \varphi \) is composed of the AND of \( m \) 3SAT clauses where a 3SAT clause is the OR of three variables or their negation. A quadratic equations instance \( E \) is composed of a list of equations, each of involving a sum of variables or their products, such as \( x_{19}x_{52} - x_{12} + 2x_{33} = 2 \), etc. We will include the constraints \( x_i^2 - x_i = 0 \) for every \( i \in [n] \) in our equations, which means that we can restrict attention to assignments where \( x_i \in \{0,1\} \) for every \( i \).

& \quad \text{There is a natural way to map a 3SAT instance into a set of cubic equations \( E' \), and that is to map a clause such as \((x_{17} \lor x_{101} \lor x_{57}) \) (which is equivalent to the negation of \( x_{17} \land x_{101} \land x_{57} \)) to the equation \((1 - x_{17})x_{101}(1 - x_{57}) = 0 \). Therefore, we can map a formula \( \varphi \) with \( n \) variables \( m \) clauses into a set \( E' \) of \( m + n \) cubic equations on \( n \) variables (that is, one equation per each clause, plus one equation of the form \( x_i^2 - x_i = 0 \) for each variable to ensure that its value is in \( \{0,1\} \)) such that every assignment \( a \in \{0,1\}^n \) to the \( n \) variables satisfies the original formula if and only if it satisfies the equations of \( E' \).}

& \quad \text{To make the equations \textit{quadratic} we introduce for every two distinct } i,j \in [n] \text{ a variable } y_{i,j} \text{ and include the constraint } y_{i,j} - x_i x_j = 0 \text{ in the equations. This is a quadratic equation that ensures that } y_{i,j} = x_i x_j \text{ for every such } i,j \in [n]. \text{ Now we can turn any cubic equation in the } x \text{'s into a quadratic equation in the } x \text{ and } y \text{ variables. For example, we can “open up the parentheses” of an equation such as } (1 - x_{17})x_{101}(1 - x_{57}) = 0 \text{ to } x_{101} - x_{17}x_{101} - x_{101}x_{57} + x_{17}x_{101}x_{57} = 0. \text{ We can now replace the cubic term } x_{17}x_{101}x_{57} \text{ with the quadratic term } y_{17,101}x_{57}. \text{ This can be done for every cubic equation in the same way, replacing any cubic term } x_i x_j x_k \text{ with the term } y_{i,j} x_k. \text{ The end result will be a set of } m + n + \binom{n}{2} \text{ equations (one equation per clause, one equation per}}
\end{aligned}
\]
variable to ensure $x_i^2 - x_i = 0$, and one equation per pair $i, j$ to ensure $y_{i,j} = x_i x_j = 0$ on the $n + \binom{n}{2}$ variables $x_0, \ldots, x_{n-1}$ and $y_{i,j}$ for all pairs of distinct variables $i, j$.

To complete the proof we need to show that if we transform $\varphi$ to $E$ in this way then the 3SAT formula $\varphi$ is satisfiable if and only if the equations $E$ have a solution. This is essentially immediate from the construction, but as this is our first reduction, we spell this out fully:

- **Completeness**: We claim that if $\varphi$ is satisfiable then the equations $E$ have a solution. To prove this we need to show how to transform a satisfying assignment $a \in \{0, 1\}^n$ to the variables of $\varphi$ (that is, $a_i$ is the value assigned to $x_i$) to a solution to the variables of $E$. Specifically, if $a \in \{0, 1\}^n$ is such an assignment then by design $a$ satisfies all the cubic equations $E'$ that we constructed above. But then, if we assign to the $n + \binom{n}{2}$ variables the values $a_0, \ldots, a_{n-1}$ and $\{a_i, a_j\}$ for all $\{i, j\} \subseteq [n]$ then by construction this will satisfy all the quadratic equations of $E$ as well.

- **Soundness**: We claim that if the equations $E$ have a solution then $\varphi$ is satisfiable. Indeed, suppose that $z \in \mathbb{R}^{n+\binom{n}{2}}$ is a solution to the equations $E$. A priori $z$ could be any vector of $n + \binom{n}{2}$ numbers, but the fact that $E$ contains the equations $x_i^2 - x_i = 0$ and $y_{i,j} - x_i x_j = 0$ means that if $z$ satisfies these equations then the values it assigns to $x_i$ must be in $\{0, 1\}$ for every $i$, and the value it assigns to $y_{i,j}$ must be $x_i x_j$ for every $\{i, j\} \subseteq [n]$. Therefore by the way we constructed our equations, the value assigned $x_i$ must be a solution of the original cubic equations $E'$ and hence also of the original formula $\varphi$, which in particular implies $\varphi$ is satisfiable.

This reduction can be easily implemented in about a dozen lines of Python or any other programming language, see Fig. 13.3.

\[ \binom{n}{2} = \frac{1}{2} n(n-1) \] is the number of all size two subsets of $[n]$. We consider $\{i, j\}$ to be the same as $\{j, i\}$ since $x_i x_j = x_j x_i$ for every $i, j$.

---

**Figure 13.3**: Reducing 3SAT to satisfiability of quadratic equations. On the righthand side is Python code implementing the reduction of Theorem 13.3 and on the lefthand side is the output of this reduction on an example 3SAT instance.

### 13.3 THE INDEPENDENT SET PROBLEM

For a graph $G = (V, E)$, an independent set (also known as a stable set) is a subset $S \subseteq V$ such that there are no edges with both endpoints
in $S$ (in other words, $E(S, S) = \emptyset$). Every “singleton” (set consisting of a single vertex) is trivially an independent set, but finding larger independent sets can be challenging. The maximum independent set problem (henceforth simply “independent set”) is the task of finding the largest independent set in the graph. The independent set problem is naturally related to scheduling problems: if we put an edge between two conflicting tasks, then an independent set corresponds to a set of tasks that can all be scheduled together without conflicts. But it also arises in very different settings, including trying to find structure in protein-protein interaction graphs.

To phrase independent set as a decision problem, we think of it as a function $ISET : \{0, 1\}^* \to \{0, 1\}$ that on input a graph $G$ and a number $k$ outputs 1 if and only if the graph $G$ contains an independent set of size at least $k$. We will now reduce 3SAT to Independent set.

Theorem 13.4 — Hardness of Independent Set. $3SAT \leq_p ISET$.

Proof Idea: The idea is that finding a satisfying assignment to a 3SAT formula corresponds to satisfying many local constraints without creating any conflicts. One can think of “$x_{17} = 0$” and “$x_{17} = 1$” as two conflicting events, and of the constraints $x_{17} \lor \overline{x}_5 \lor x_9$ as creating a conflict between the events “$x_{17} = 0$”, “$x_5 = 1$” and “$x_9 = 0$”, saying that these three cannot simultaneously co-occur. Using these ideas, we can we can think of solving a 3SAT problem as trying to schedule non conflicting events, though the devil is, as usual, in the details.

Proof of Theorem 13.4. Given a 3SAT formula $\varphi$ on $n$ variables and with $m$ clauses, we will create a graph $G$ with $3m$ vertices as follows: (see Fig. 13.4 for an example)

- A clause $C$ in $\varphi$ has the form $C = y \lor y' \lor y''$ where $y, y', y''$ are literals (variables or their negation). For each such clause $C$, we will add three vertices to $G$, and label them $(C, y)$, $(C, y')$, and $(C, y'')$ respectively. We will also add the three edges between all pairs of these vertices, so they form a triangle. Since there are $m$ clauses in $\varphi$, the graph $G$ will have $3m$ vertices.

- In addition to the above edges, we also add an edge between every pair vertices of the form $(C, y)$ and $(C', y')$ where $y$ and $y'$ are conflicting literals. That is, we add an edge between $(C, y)$ and $(C, y')$ if there is an $i$ such that $y = x_i$ and $y' = \overline{x}_i$ or vice versa.

The above construction of $G$ based on $\varphi$ can clearly be carried out in polynomial time. Hence to prove the theorem we need to show that $\varphi$...
is satisfiable if and only if $G$ contains an independent set of $m$ vertices.

We now show both directions of this equivalence:

**Part 1: Completeness.** The “completeness” direction is to show that if $\varphi$ has a satisfying assignment $x^*$, then $G$ has an independent set $S^*$ of $m$ vertices. Let us now show this.

Indeed, suppose that $\varphi$ has a satisfying assignment $x^* \in \{0,1\}^n$. Then for every clause $C = y \lor y' \lor y''$ of $\varphi$, one of the literals $y, y', y''$ must evaluate to true under the assignment $x^*$ (as otherwise it would not satisfy $\varphi$). We let $S$ be a set of $m$ vertices that is obtained by choosing for every clause $C$ one vertex of the form $(C, y)$ such that $y$ evaluates to true under $x^*$. (If there is more than one such vertex for the same $C$, we arbitrarily choose one of them.)

We claim that $S$ is an independent set. Indeed, suppose otherwise that there was a pair of vertices $(C, y)$ and $(C', y')$ in $S$ that have an edge between them. Since we picked one vertex out of each triangle corresponding to a clause, it must be that $C \neq C'$. Hence the only way that there is an edge between $(C, y)$ and $(C', y')$ is if $y$ and $y'$ are conflicting literals (i.e. $y = \overline{x}_i$ and $y' = x_i$ for some $i$). But that would that they can’t both evaluate to true under the assignment $x^*$, which contradicts the way we constructed the set $S$. This completes the proof of the completeness condition.

**Part 2: Soundness.** The “soundness” direction is to show that if $G$ has an independent set $S^*$ of $m$ vertices, then $\varphi$ has a satisfying assignment $x^* \in \{0,1\}^n$. Let us now show this.

Indeed, suppose that $G$ has an independent set $S^*$ with $m$ vertices. We will define an assignment $x^* \in \{0,1\}^n$ for the variables of $\varphi$ as follows. For every $i \in [n]$, we set $x^*_i$ according to the following rules:

- If $S^*$ contains a vertex of the form $(C, x_i)$ then we set $x^*_i = 1$.
- If $S^*$ contains a vertex of the form $(C, \overline{x}_i)$ then we set $x^*_i = 0$.
- If $S^*$ does not contain a vertex of either of these forms, then it does not matter which value we give to $x^*_i$, but for concreteness we’ll set $x^*_i = 0$.

The first observation is that $x^*$ is indeed well defined, in the sense that the rules above do not conflict with one another, and ask to set $x^*_i$ to be both 0 and 1. This follows from the fact that $S^*$ is an independent set and hence if it contains a vertex of the form $(C, x_i)$ then it cannot contain a vertex of the form $(C', \overline{x}_i)$.

We now claim that $x^*$ is a satisfying assignment for $\varphi$. Indeed, since $S^*$ is an independent set, it cannot have more than one vertex inside each one of the $m$ triangles $(C, y), (C, y'), (C, y'')$ corresponding to a clause of $\varphi$. Hence since $|S^*| = m$, it must have exactly one vertex in each such triangle. For every clause $C$ of $\varphi$, if $(C, y)$ is the vertex in
$S^*$ in the triangle corresponding to $C$, then by the way we defined $x^*$, the literal $y$ must evaluate to true, which means that $x^*$ satisfies this clause. Therefore $x^*$ satisfies all clauses of $\varphi$, which is the definition of a satisfying assignment.

This completes the proof of Theorem 13.4.

Figure 13.4: The reduction of 3SAT to Independent Set. On the righthand side is Python code that implements this reduction. On the lefthand side is a sample output of the reduction. We use black for the “triangle edges” and red for the “conflict edges”. Note that the satisfying assignment $x^* = 0110$ corresponds to the independent set $(0, \neg x_3), (1, \neg x_0), (2, x_2)$.

13.4 REDUCING INDEPENDENT SET TO MAXIMUM CUT

Theorem 13.5 — Hardness of Max Cut. $\text{IS} \leq_p \text{MAXCUT}$

Proof Idea: We will map a graph $G$ into a graph $H$ such that a large independent set in $G$ becomes a partition cutting many edges in $H$. We can think of a cut in $H$ as coloring each vertex either “blue” or “red”. We will add a special “source” vertex $s^*$, connect it to all other vertices, and assume without loss of generality that it is colored blue. Hence the more vertices we color red, the more edges from $s^*$ we cut. Now, for every edge $u, v$ in the original graph $G$ we will add a special “gadget” which will be a small subgraph that involves $u, v, s^*$, and two other additional vertices. We design the gadget in a way so that if the red vertices are not an independent set in $G$ then the corresponding cut in $H$ will be “penalized” in the sense that it would not cut as many edges. Once we set for ourselves this objective, it is not hard to find a gadget that achieves it—see the proof below.

Proof of Theorem 13.5. We will transform a graph $G$ of $n$ vertices and $m$ edges into a graph $H$ of $n + 1 + 2m$ vertices and $n + 5m$ edges in the following way: the graph $H$ will contain all vertices of $G$ (though not the edges between them!) and in addition to that will contain:

* A special vertex $s^*$ that is connected to all the vertices of $G$
For every edge \( e = \{u, v\} \in E(G) \), two vertices \( e_0, e_1 \) such that \( e_0 \) is connected to \( u \) and \( e_1 \) is connected to \( v \), and moreover we add the edges \( \{e_0, e_1\}, \{e_0, s^*\}, \{e_1, s^*\} \) to \( H \).

Theorem 13.5 will follow by showing that \( G \) contains an independent set of size at least \( k \) if and only if \( H \) has a cut cutting at least \( k + 4m \) edges. We now prove both directions of this equivalence:

**Part 1: Completeness.** If \( I \) is an independent \( k \)-sized set in \( G \), then we can define \( S \) to be a cut in \( H \) of the following form: we let \( S \) contain all the vertices of \( I \) and for every edge \( e = \{u, v\} \in E(G) \), if \( u \in I \) and \( v \notin I \) then we add \( e_1 \) to \( S \); if \( u \notin I \) and \( v \in I \) then we add \( e_0 \) to \( S \); and if \( u \notin I \) and \( v \notin I \) then we add both \( e_0 \) and \( e_1 \) to \( S \). (We don’t need to worry about the case that both \( u \) and \( v \) are in \( I \) since it is an independent set.) We can verify that in all cases the number of edges from \( S \) to its complement in the gadget corresponding to \( e \) will be four (see Fig. 13.5). Since \( s^* \) is not in \( S \), we also have \( k \) edges from \( s^* \) to \( I \), for a total of \( k + 4m \) edges.

**Part 2: Soundness.** Suppose that \( S \) is a cut in \( H \) that cuts at least \( C = k + 4m \) edges. We can assume that \( s^* \) is not in \( S \) (otherwise we can “flip” \( S \) to its complement \( \overline{S} \), since this does not change the size of the cut). Now let \( I \) be the set of vertices in \( S \) that correspond to the original vertices of \( G \). If \( I \) was an independent set of size \( k \) then would be done. This might not always be the case but we will see that if \( I \) is not an independent set then its also larger than \( k \). Specifically, we define \( m_{in} = |E(I, I)| \) be the set of edges in \( G \) that are contained in \( I \) and let \( m_{out} = m - m_{in} \) (i.e., if \( I \) is an independent set then \( m_{in} = 0 \) and \( m_{out} = m \)). By the properties of our gadget we know that for every edge \( \{u, v\} \) of \( G \), we can cut at most three edges when both \( u \) and \( v \) are in \( S \), and at most four edges otherwise. Hence the number \( C \) of edges cut by \( S \) satisfies \( C \leq |I| + 3m_{in} + 4m_{out} = |I| + 3m_{in} + 4(m - m_{in}) = |I| + 4m - m_{in} \). Since \( C = k + 4m \) we get that \( |I| - m_{in} \geq k \). Now we can transform \( I \) into an independent set \( I' \) by going over every one of the \( m_{in} \) edges that are inside \( I \) and removing one of the endpoints of the edge from it. The resulting set \( I' \) is an independent set in the graph \( G \) of size \( |I| - m_{in} \geq k \) and so this concludes the proof of the soundness condition.

13.5 REDUCING 3SAT TO LONGEST PATH

One of the most basic algorithms in Computer Science is Dijkstra’s algorithm to find the shortest path between two vertices. We now show that in contrast, an efficient algorithm for the longest path problem would imply a polynomial-time algorithm for 3SAT.
In the reduction of independent set to max cut, we have a “gadget” corresponding to every edge $e = \{u, v\}$ in the original graph. If we think of the side of the cut containing the special source vertex $s^*$ as “blue” and the other side as “red”, then the leftmost and center figures show that if $u$ and $v$ are not both red then we can cut four edges from the gadget. In contrast, by enumerating all possibilities one can verify that if both $u$ and $v$ are red, then no matter how we color the intermediate vertices $e_0, e_1$, we will cut at most three edges from the gadget.

The reduction of independent set to max cut. On the righthand side is Python code implementing the reduction. On the lefthand side is an example output of the reduction where we apply it to the independent set instance that is obtained by running the reduction of Theorem 13.4 on the 3CNF formula $(\overline{x}_0 \lor \overline{x}_1 \lor \overline{x}_2) \land (\overline{x}_2 \lor x_1 \lor \overline{x}_3) \land (\overline{x}_1 \lor x_2 \lor x_3)$. 
**Theorem 13.6 — Hardness of longest path.**

$$3SAT \leq_p LONGPATH \quad (13.4)$$

**Proof Idea:** To prove Theorem 13.6 need to show how to transform a 3CNF formula $\varphi$ into a graph $G$ and two vertices $s, t$ such that $G$ has a path of length at least $k$ if and only if $\varphi$ is satisfiable. The idea of the reduction is sketched in Fig. 13.7 and Fig. 13.8. We will construct a graph that contains a potentially long “snaking path” that corresponds to all variables in the formula. We will add a “gadget” corresponding to each clause of $\varphi$ in a way that we would only be able to use the gadgets if we have a satisfying assignment.  

**Proof of Theorem 13.6.** We build a graph $G$ that “snakes” from $s$ to $t$ as follows. After $s$ we add a sequence of $n$ long loops. Each loop has an “upper path” and a “lower path”. A simple path cannot take both the upper path and the lower path, and so it will need to take exactly one of them to reach $s$ from $t$.

Our intention is that a path in the graph will correspond to an assignment $x \in \{0, 1\}^n$ in the sense that taking the upper path in the $i^{th}$ loop corresponds to assigning $x_i = 1$ and taking the lower path corresponds to assigning $x_i = 0$. When we are done snaking through all the $n$ loops corresponding to the variables to reach $t$ we need to pass through $m$ “obstacles”: for each clause $j$ we will have a small gadget consisting of a pair of vertices $s_j, t_j$ that have three paths between them. For example, if the $j^{th}$ clause had the form $x_{17} \lor x_{55} \lor x_{72}$ then one path would go through a vertex in the lower loop corresponding to $x_{17}$, one path would go through a vertex in the upper loop corresponding to $x_{55}$ and the third would go through the lower loop corresponding to $x_{72}$. We see that if we went in the first stage according to a satisfying assignment then we will be able to find a free vertex to travel from $s_j$ to $t_j$. We link $t_1$ to $s_2$, $t_2$ to $s_3$, etc and link $t_m$ to $t$. Thus a satisfying assignment would correspond to a path from $s$ to $t$ that goes through one path in each loop corresponding to the variables, and one path in each loop corresponding to the clauses. We can make the loop corresponding to the variables long enough so that we must take the entire path in each loop in order to have a fighting chance of getting a path as long as the one corresponds to a satisfying assignment. But if we do that, then the only way if we are able to reach $t$ is if the paths we took corresponded to a satisfying assignment, since otherwise we will have one clause $j$ where we cannot reach $t_j$ from $s_j$ without using a vertex we already used before. ■
Figure 13.7: We can transform a 3SAT formula $\varphi$ into a graph $G$ such that the longest path in the graph $G$ would correspond to a satisfying assignment in $\varphi$. In this graph, the black colored part corresponds to the variables of $\varphi$ and the blue colored part corresponds to the vertices. A sufficiently long path would have to first “snake” through the black part, for each variable choosing either the “upper path” (corresponding to assigning it the value True) or the “lower path” (corresponding to assigning it the value False). Then to achieve maximum length the path would traverse through the blue part, where to go between two vertices corresponding to a clause such as $x_{17} \lor \overline{x}_{32} \lor x_{57}$, the corresponding vertices would have to have been not traversed before.

Figure 13.8: The graph above with the longest path marked on it, the part of the path corresponding to variables is in green and part corresponding to the clauses is in pink.
Lecture Recap

- The computational complexity of many seemingly unrelated computational problems can be related to one another through the use of reductions.
- If \( F \leq_p G \) then a polynomial-time algorithm for \( G \) can be transformed into a polynomial-time algorithm for \( F \).
- Equivalently, if \( F \leq_p G \) and \( F \) does not have a polynomial-time algorithm then neither does \( G \).
- We’ve developed many techniques to show that \( 3SAT \leq_p F \) for interesting functions \( F \). Sometimes we can do so by using transitivity of reductions: if \( 3SAT \leq_p G \) and \( G \leq_p F \) then \( 3SAT \leq_p F \).

13.6 EXERCISES

**Disclaimer** Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

**Exercise 13.1** Prove ??

**Exercise 13.2** — Transitivity of reductions. Prove that if \( F \leq_p G \) and \( G \leq_p H \) then \( F \leq_p H \).

13.7 BIBLIOGRAPHICAL NOTES

Reduction of independent set to max cut taken from these notes.

13.8 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

13.9 ACKNOWLEDGEMENTS
14

NP, NP completeness, and the Cook-Levin Theorem

“In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually”, Richard Karp, 1972

“Sad to say, but it will be many more years, if ever before we really understand the Mystical Power of Twoness… 2-SAT is easy, 3-SAT is hard, 2-dimensional matching is easy, 3-dimensional matching is hard. Why? oh, Why?”
Eugene Lawler

14.1 THE CLASS NP

So far we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path. But to show that these problems are computationally equivalent we need to give reductions in the other direction, reducing each one of these problems to 3SAT as well. It turns out we can reduce all three problems to 3SAT in one fell swoop.

In fact, this result extends far beyond these particular problems. All of the problems we discussed in Chapter 13, and a great many other problems, share the same commonality: they are all search problems, where the goal is to decide, given an instance $x$, whether there exists a solution $y$ that satisfies some condition that can be verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable; in Max-Cut the instance is a graph and the solution is a cut in the graph; and so on and so forth. It turns out that every such search problem can be reduced to 3SAT.

To make this precise, we make the following mathematical definition: we define the class NP to contain all Boolean functions that correspond to a search problem of the form above— that is, functions

Learning Objectives:
- Introduce the class NP capturing a great many important computational problems
- NP-completeness: evidence that a problem might be intractable.
- The P vs NP problem.
that output 1 on $x$ if and only if there exists a solution $w$ such that the pair $(x, w)$ satisfies some polynomial-time checkable condition. Formally, $\text{NP}$ is defined as follows:

**Definition 14.1 — NP.** We say that $F : \{0, 1\}^* \to \{0, 1\}$ is in $\text{NP}$ if there exists some constants $a, b \in \mathbb{N}$ and $V : \{0, 1\}^* \to \{0, 1\}$ such that $V \in \text{P}$ and for every $x \in \{0, 1\}^n$,

$$F(x) = 1 \iff \exists w \in \{0, 1\}^{an}, \text{ s.t. } V(xw) = 1. \quad (14.1)$$

In other words, for $F$ to be in $\text{NP}$, there needs to exist some polynomial-time computable verification function $V$, such that if $F(x) = 1$ then there must exist $w$ (of length polynomial in $|x|$) such that $V(xw) = 1$, and if $F(x) = 0$ then for every such $w$, $V(xw) = 0$. Since the existence of this string $w$ certifies that $F(x) = 1$, $w$ is often referred to as a certificate, witness, or proof that $F(x) = 1$.

See also Fig. 14.1 for an illustration of Definition 14.1. The name $\text{NP}$ stands for “nondeterministic polynomial time” and is used for historical reasons; see the bibliographical notes. The string $w$ in Eq. (14.1) is sometimes known as a solution, certificate, or witness for the instance $x$. 

---

**NP and proof systems** The definition of $\text{NP}$ means that for every $F \in \text{NP}$ and string $x \in \{0, 1\}^*$, $F(x) = 1$ if and only if there is a short and efficiently verifiable proof of this fact. That is, we can think of the function $V$ in Definition 14.1 as a verifier algorithm, similar to what we’ve seen in Section 10.1. The verifier checks whether a given string $w \in \{0, 1\}^*$ is a valid proof for the statement “$F(x) = 1$.” Essential practice...
For example, as shown below, 3 mrbj ∈ NP, but the function 3 mrbj that on input a 3CNF formula \( \varphi \) outputs 1 if and only if \( \varphi \) is not satisfiable is not known (nor believed) to be in NP.

Initially all proof systems considered in mathematics involve line-by-line checks that can be carried out in polynomial time. Thus the heart of NP is asking for statements that have short (i.e., polynomial in the size of the statements) proof. Indeed, as we will see in Chapter 15, Kurt Gödel phrased the question of whether NP = P as asking whether "the mental work of a mathematician [in proving theorems] could be completely replaced by a machine".

Definition 14.1 is asymmetric in the sense that there is a difference between an output of 1 and an output of 0. You should make sure you understand why this definition does not guarantee that if \( F \in \text{NP} \) then the function 1 - \( F \) (i.e., the map \( x \mapsto 1 - F(x) \)) is in \text{NP} as well. In fact, it is believed that there do exist functions \( F \) satisfying \( F \in \text{NP} \) but 1 - \( F \not\in \text{NP}. \)

This is in contrast to the class P which (as you should verify) does satisfy that if \( \varphi \in \text{P} \) then 1 - \( \varphi \) is in \text{P} as well.

14.1.1 Examples of NP functions

Example 14.2 — 3SAT \in \text{NP}. 3SAT is in \text{NP} since for every \( \ell \)-variable formula \( \varphi \), 3SAT(\( \varphi \)) = 1 if and only if there exists a satisfying assignment \( x \in \{0,1\}^\ell \) such that \( \varphi(x) = 1 \), and we can check this condition in polynomial time.

The above reasoning explains why 3SAT is in \text{NP}, but since this is our first example, we will now belabor the point and expand out in full formality what is the precise representation of the witness \( w \) and the algorithm \( V \) that demonstrate that 3SAT is in \text{NP}.

Specifically, we can represent a 3CNF formula \( \varphi \) with \( k \) variables and \( m \) clauses as a string of length \( n = O(m \log k) \), since every one of the \( m \) clauses involves three variables and their negation, and the identity of each variable can be represented using \( \lceil \log_2 k \rceil \). We assume that every variable participates in some clause (otherwise it can be ignored) and hence that \( m \geq k \), which in particular means that \( n \) is larger than both \( m \) and \( k \).

We can represent an assignment to the \( k \) variables using a \( k \)-length string, which, since \( n > k \), can be “padded” to a string \( w \in \{0,1\}^n \) in some standard way. (For example, if \( y \in \{0,1\}^k \) is the assignment, we can let \( w = y10^{n-k-1} \); given the string \( w \) we can “read off” \( y \), by chopping off all the zeroes at the end of \( w \) until we encounter the first 1, which we remove as well.)

Now checking whether a given assignment \( y \in \{0,1\}^k \) satisfies
a given $k$-variable $3CNF$ $\varphi$ can be done in polynomial time through the following algorithm $V$:

**Algorithm $V$:**

**Input:**
1. $3CNF$ formula $\varphi$ with $k$ variables and $m$ clauses (encoded as a string of length $n = O(m \log k)$)
2. Assignment $y \in \{0,1\}^k$ to the variables of $\varphi$ (encoded using padding as a string $w \in \{0,1\}^n$)

**Output:** 1 if and only if $y$ satisfies $\varphi$.

**Operation:**
1. For every clause $C = (\ell_1 \lor \ell_2 \lor \ell_3)$ of $\varphi$ (where $\ell_1, \ell_2, \ell_3$ are literals), if all three literals evaluate to false under the assignment $y$ then halt and output 0.
2. Output 1.

Algorithm $V$ runs in time polynomial in the length $n$ of $\varphi$’s description as a string. Indeed there are $m$ clauses, and checking the evaluation of a literal of the form $y_i$ or $\neg y_j$ can be done by scanning the $k$-length string $y$, and hence the running time of Algorithm $V$ is at most $O(mk) = O(n^2)$, as both $k$ and $m$ are smaller than $n$.

By its definition the algorithm outputs 1 if and only if the assignment $y$ satisfies all the clauses of the $3CNF$ formula $\varphi$, which means that $3SAT(\varphi) = 1$ if and only if there exists some $w \in \{0,1\}^n$ such that $V(\varphi w) = 1$ which is precisely the condition needed to show that $3SAT \in NP$ per Definition 14.1.

**Padding a witness** The “padding trick” we used in Example 14.2 can always be used to expand a witness of length smaller than $an^k$ to a witness of exactly that length. Therefore one can think of the condition Eq. (14.1) in Definition 14.1 as simply stipulating that the “solution” $w$ to the problem $x$ is of length at most polynomial in $|x|$.

Here are some more examples for problems in $NP$. For each one of these problems we merely sketch how the witness is represented and why it is efficiently checkable, but working out the details can be a good way to get more comfortable with Definition 14.1:

- **QUADEQ** is in $NP$ since for every $t$-variable instance of quadratic
equations $E$, $QUADEQ(E) = 1$ if and only if there exists an assignment $x \in \{0, 1\}^l$ that satisfies $E$. We can check the condition that $x$ satisfies $E$ in polynomial time by enumerating over all the equations in $E$, and for each such equation $e$, plug in the values of $x$ and verify that $e$ is satisfied.

- $ISET$ is in $\textbf{NP}$ since for every graph $G$ and integer $k$, $ISET(G, k) = 1$ if and only if there exists a set $S$ of $k$ vertices that contains no pair of neighbors in $G$. We can check the condition that $S$ is an independent set of size $\geq k$ in polynomial time by first checking that $|S| \geq k$ and then enumerating over all edges $\{u, v\}$ in $G$, and for each such edge verify that either $u \notin S$ or $v \notin S$.

- $LONPATH$ is in $\textbf{NP}$ since for every graph $G$ and integer $k$, $LONPATH(G, k) = 1$ if and only if there exists a simple path $P$ in $G$ that is of length at least $k$. We can check the condition that $P$ is a simple path of length $k$ in polynomial time by checking that it has the form $(v_0, v_1, \ldots, v_k)$ where each $v_i$ is a vertex in $G$, no $v_i$ is repeated, and for every $i \in [k]$, the edge $\{v_i, v_{i+1}\}$ is present in the graph.

- $MAXCUT$ is in $\textbf{NP}$ since for every graph $G$ and integer $k$, $MAXCUT(G, k) = 1$ if and only if there exists a cut $(S, \overline{S})$ in $G$ that cuts at least $k$ edges. We can check that condition that $(S, \overline{S})$ is a cut of value at least $k$ in polynomial time by checking that $S$ is a subset of $G$’s vertices and enumerating over all the edges $\{u, v\}$ of $G$, counting those edges such that $u \in S$ and $v \notin S$ or vice versa.

### 14.1.2 Basic facts about $\textbf{NP}$

The definition of $\textbf{NP}$ is one of the most important definitions of this book, and is worth while taking the time to digest and internalize. The following solved exercises establish some basic properties of this class. As usual, I highly recommend that you try to work out the solutions yourself.

**Solved Exercise 14.1 — Verifying is no harder than solving.** Prove that $P \subseteq \textbf{NP}$.

**Solution:** Suppose that $F \in P$. Define the following function $V$: $V(x0^n) = 1$ iff $n = |x|$ and $F(x) = 1$. ($V$ outputs 0 on all other inputs.) Since $F \in P$ we can clearly compute $V$ in polynomial time as well.

Let $x \in \{0, 1\}^n$ be some string. If $F(x) = 1$ then $V(x0^n) = 1$. On the other hand, if $F(x) = 0$ then for every $w \in \{0, 1\}^n$, $V(xw) = 0$. Therefore, setting $a = b = 1$, we see that $V$ satisfies Eq. (14.1), and establishes that $F \in \textbf{NP}$. 


One function that is believed to lie outside \( \text{NP} \) is the function \( 3\text{mb} ) defined as \( 3\text{mb}(\varphi) = 1 - 3\text{mb}(\varphi) \) for every 3CNF formula \( \varphi \). The conjecture that \( 3\text{mb} \notin \text{NP} \) is known as the "\( \text{NP} \neq \text{coNP} \)" conjecture. It implies the \( \text{P} \neq \text{NP} \) conjecture (can you see why?).

NP does not mean non-polynomial! People sometimes think that \( \text{NP} \) stands for "non polynomial time". As Solved Exercise 14.1 shows, this is far from the truth, and in fact every polynomial-time computable function is in \( \text{NP} \) as well.

If \( F \) is in \( \text{NP} \) it certainly does not mean that \( F \) is hard to compute (though it does not, as far as we know, necessarily mean that it’s easy to compute either). Rather, it means that \( F \) is easy to verify, in the technical sense of Definition 14.1.

Solved Exercise 14.2 — NP is in exponential time. Prove that \( \text{NP} \subseteq \text{EXP} \).

Solution: Suppose that \( F \in \text{NP} \) and let \( V \) be the polynomial-time computable function that satisfies Eq. (14.1) and \( a, b \) the corresponding constants. Then the following is an exponential-time algorithm \( A \) to compute \( F \):

\[
\begin{align*}
\text{Algorithm } A:
\text{Input: } & x \in \{0,1\}^*, \text{ let } n = |x| \\
\text{Operation:} & \\
& 1. \text{ For every } w \in \{0,1\}^{an^b}, \text{ if } V(xw) = 1 \text{ then halt and output 1.} \\
& 2. \text{ Output } 0.
\end{align*}
\]

Since \( V \in \text{P} \), for every \( x \in \{0,1\}^n \), Algorithm \( A \) runs in time \( \text{poly}(n)^2n^b \). Moreover by Eq. (14.1), \( A \) will output 1 on \( x \) if and only if \( F(x) = 1 \).

Solved Exercise 14.1 and Solved Exercise 14.2 together imply that

\[
\text{P} \subseteq \text{NP} \subseteq \text{EXP}.
\] (14.2)

The time hierarchy theorem (Theorem 12.7) implies that \( \text{P} \subseteq \text{EXP} \) and hence at least one of the two inclusions \( \text{P} \subseteq \text{NP} \) or \( \text{NP} \subseteq \text{EXP} \) is strict. It is believed that both of them are in fact strict inclusions. That is, it is believed that there are functions in \( \text{NP} \) that cannot be computed in polynomial time (this is the \( \text{P} \neq \text{NP} \) conjecture) and that there are functions \( F \) in \( \text{EXP} \) for which we cannot even efficiently certify that \( F(x) = 1 \) for a given input \( x \).

One function \( F \) that is believed to lie outside \( \text{NP} \) is the function \( 3\text{SAT} \) defined as \( 3\text{SAT}(\varphi) = 1 - 3\text{SAT}(\varphi) \) for every 3CNF formula \( \varphi \). The conjecture that \( 3\text{SAT} \notin \text{NP} \) is known as the "\( \text{NP} \neq \text{coNP} \)" conjecture. It implies the \( \text{P} \neq \text{NP} \) conjecture (can you see why?).
verify” than $G$. That is, regardless of whether or not it is in $P$, if $G$ has the property that solutions to it can be efficiently verified, then so does $F$.

**Solved Exercise 14.3 — Reductions and NP.** Let $F, G : \{0, 1\}^* \rightarrow \{0, 1\}$. Show that if $F \leq_p G$ and $G \in \text{NP}$ then $F \in \text{NP}$.

**Solution:** Suppose that $G$ is in NP and in particular there exists $a, b$ and $V \in P$ such that for every $y \in \{0, 1\}^*$, $G(y) = 1 \iff \exists w \in \{0, 1\}^a \forall y \in \{0, 1\}^b V(yw) = 1$. Define $V'(x, w) = 1$ iff $V(R(x)w) = 1$ where $R$ is the polynomial-time reduction demonstrating that $F \leq_p G$.

Then for every $x \in \{0, 1\}^*$,

$$F(x) = 1 \iff G(R(x)) = 1 \iff \exists w \in \{0, 1\}^a \forall y \in \{0, 1\}^b V(R(x)w) = 1 \iff \exists w \in \{0, 1\}^a \forall y \in \{0, 1\}^b V'(x, w) = 1$$

(14.3)

Since there are some constants $a', b'$ such that $|R(x)| \leq a'|x|^{b'}$ for every $x \in \{0, 1\}^*$, by simple padding we can modify $V'$ to an algorithm that certifies that $F \in \text{NP}$.

---

**14.2 FROM NP TO 3SAT: THE COOK-LEVIN THEOREM**

We have seen several examples of problems for which we do not know if their best algorithm is polynomial or exponential, but we can show that they are in NP. That is, we don’t know if they are easy to solve, but we do know that it is easy to verify a given solution. There are many, many, many more examples of interesting functions we would like to compute that are easily shown to be in NP. What is quite amazing is that if we can solve 3SAT then we can solve all of them!

The following is one of the most fundamental theorems in Computer Science:

**Theorem 14.3 — Cook-Levin Theorem.** For every $F \in \text{NP}$, $F \leq_p 3\text{SAT}$.

We will soon show the proof of Theorem 14.3, but note that it immediately implies that QUADEQ, LONGPATH, and MAXCUT all reduce to 3SAT. Combining it with the reductions we’ve seen in Chapter 13, it implies that all these problems are equivalent! For example, to reduce QUADEQ to LONGPATH, we can first reduce QUADEQ to 3SAT using Theorem 14.3 and use the reduction we’ve seen in Theorem 13.6 from 3SAT to LONGPATH. That is, since QUADEQ ∈ NP, Theorem 14.3 implies that QUADEQ ≤ₚ 3SAT, and Theorem 13.6 implies that 3SAT ≤ₚ LONGPATH, which by the transitivity of reductions (Lemma 13.2) means that QUADEQ ≤ₚ LONGPATH. Similarly, since LONGPATH ∈ NP, we can use The-
Theorem 14.3 and Theorem 13.3 to show that $LONGPATH \leq_p 3SAT \leq_p QUADEQ$, concluding that $LONGPATH$ and $QUADEQ$ are computationally equivalent.

There is of course nothing special about $QUADEQ$ and $LONGPATH$ here: by combining Theorem 14.3 with the reductions we saw, we see that just like $3SAT$, every $F \in \textbf{NP}$ reduces to $LONGPATH$, and the same is true for $QUADEQ$ and $MAXCUT$. All these problems are in some sense “the hardest in $\textbf{NP}$” since an efficient algorithm for any one of them would imply an efficient algorithm for all the problems in $\textbf{NP}$. This motivates the following definition:

**Definition 14.4 — NP-hardness and NP-completeness.** We say that $G : \{0, 1\}^* \rightarrow \{0, 1\}$ is **NP hard** if for every $F \in \textbf{NP}$, $F \leq_p G$.

We say that $G : \{0, 1\}^* \rightarrow \{0, 1\}$ is **NP complete** if $G$ is NP hard and $G$ is in $\textbf{NP}$.

The Cook-Levin Theorem (Theorem 14.3) can be rephrased as saying that $3SAT$ is NP hard, and since it is also in NP, this means that $3SAT$ is NP complete. Together with the reductions of Chapter 13, Theorem 14.3 shows that despite their superficial differences, $3SAT$, quadratic equations, longest path, independent set, and maximum cut, are all NP-complete. Many thousands of additional problems have been shown to be NP-complete, arising from all the sciences, mathematics, economics, engineering and many other fields.³

### 14.2.1 What does this mean?

As we’ve seen in Solved Exercise 14.1, $P \subseteq \textbf{NP}$. The most famous conjecture in Computer Science is that this containment is strict. That is, it is widely conjectured that $P \neq \textbf{NP}$. One way to refute the conjecture that $P \neq \textbf{NP}$ is to give a polynomial-time algorithm for even a single one of the NP-complete problems such as 3SAT, Max Cut, or the thousands of others that have been studied in all fields of human endeavors. The fact that these problems have been studied by so many people, and yet not a single polynomial-time algorithm for any of them has been found, supports that conjecture that indeed $P \neq \textbf{NP}$. In fact, for many of these problems (including all the ones we mentioned above), we don’t even know of a $2^{o(n)}$-time algorithm! However, to the frustration of computer scientists, we have not yet been able to prove that $P \neq \textbf{NP}$ or even rule out the existence of an $O(n)$-time algorithm for 3SAT. Resolving whether or not $P = \textbf{NP}$ is known as the $P$ vs $\textbf{NP}$ problem. A million-dollar prize has been offered for the solution of this problem, a popular book has been written, and every year a new paper comes out claiming a proof of $P = \textbf{NP}$ or $P \neq \textbf{NP}$, only to wither under scrutiny.⁴ The following 120 page survey of Aaronson, as well

³ For some partial lists, see this Wikipedia page and this website.

⁴ The following web page keeps a catalog of these failed attempts. At the time of this writing, it lists about 110 papers claiming to resolve the question, of which about 60 claim to prove that $P = \textbf{NP}$ and about 50 claim to prove that $P \neq \textbf{NP}$. ⁵
as chapter 3 in Wigderson’s upcoming book are excellent sources for summarizing what is known about this problem.

![Figure 14.2: The world if P \(\neq\) NP (left) and P = NP (right). In the former case the set of NP-complete problems is disjoint from P and Ladner’s theorem shows that there exist problems that are neither in P nor are NP-complete. (There are remarkably few natural candidates for such problems, with some prominent examples being decision variants of problems such as integer factoring, lattice shortest vector, and finding Nash equilibria.) In the latter case that P = NP the notion of NP-completeness loses its meaning, as essentially all functions in P (save for the trivial constant zero and constant one functions) are NP-complete.](image)

One of the mysteries of computation is that people have observed a certain empirical “zero-one law” or “dichotomy” in the computational complexity of natural problems, in the sense that many natural problems are either in P (often in \(TIME(O(n))\) or \(TIME(O(n^2))\)), or they are are NP hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, with not too many examples where the best running time is some strange intermediate complexity such as \(2^{\sqrt{\log n}}\). However, it is believed that there exist problems in NP that are neither in P nor are NP-complete, and in fact a result known as “Ladner’s Theorem” shows that if P \(\neq\) NP then this is indeed the case (see also Exercise 14.1 and Fig. 14.2).

14.2.2 The Cook-Levin Theorem: Proof outline

We will now prove the Cook-Levin Theorem, which is the underpinning to a great web of reductions from 3SAT to thousands of problems across great many fields. Some problems that have been shown to be NP-complete include: minimum-energy protein folding, minimum surface-area foam configuration, map coloring, optimal Nash equilibrium, quantum state entanglement, minimum supersequence of a genome, minimum codeword problem, shortest vector in a lat-

5 TODO: maybe add examples of NP hard problems as a barrier to understanding - problems from economics, physics, etc. that prevent having a closed-form solutions

6 TODO: maybe include knots
Figure 14.3: A rough illustration of the (conjectured) status of problems in exponential time. Darker colors correspond to higher running time, and the circle in the middle is the problems in $P$. $NP$ is a (conjectured to be proper) superclass of $P$ and the $NP$-complete problems (or $NPC$ for short) are the “hardest” problems in $NP$, in the sense that a solution for one of them implies a solution for all other problems in $NP$. It is conjectured that all the $NP$-complete problems require at least $\exp(n^\epsilon)$ time to solve for a constant $\epsilon > 0$, and many require $\exp(\Omega(n))$ time. The permanent is not believed to be contained in $NP$ though it is $NP$-hard, which means that a polynomial-time algorithm for it implies that $P = NP$. 
nce, minimum genus knots, positive Diophantine equations, integer programming, and many many more. The worst-case complexity of all these problems is (up to polynomial factors) equivalent to that of 3SAT, and through the Cook-Levin Theorem, to all problems in NP.

To prove Theorem 14.3 we need to show that $F \leq_p 3SAT$ for every $F \in NP$. We will do so in three stages. We define two intermediate problems: \textit{NANDSAT} and \textit{3NAND}. We will shortly show the definitions of these two problems, but Theorem 14.3 will follow from combining the following three results:

1. \textit{NANDSAT} is NP hard (Lemma 14.5).
2. \textit{NANDSAT} $\leq_p 3NAND$ (Lemma 14.7).
3. \textit{3NAND} $\leq_p 3SAT$ (Lemma 14.8).

By the transitivity of reductions, it will follow that for every $F \in NP$,

$$F \leq_p NANDSAT \leq_p 3NAND \leq_p 3SAT \quad (14.4)$$

hence establishing Theorem 14.3.

We will prove these three results Lemma 14.5, Lemma 14.7 and Lemma 14.8 one by one, providing the requisite definitions as we go along.

\textbf{14.3 THE NANDSAT PROBLEM, AND WHY IT IS NP HARD.}

We define the \textit{NANDSAT} problem as follows. On input a string $Q \in \{0, 1\}^*$, we define $NANDSAT(Q) = 1$ if and only if $Q$ is a valid representation of an $n$-input and single-output NAND program and there exists some $w \in \{0, 1\}^n$ such that $Q(w) = 1$. While we don’t need this to prove Lemma 14.5, note that NANDSAT is in NP since we can verify that $Q(w) = 1$ using the polynomial-time algorithm for evaluating NAND programs.\footnote{Q is a NAND program and not a NAND++ program, and hence it is only defined on inputs of some particular size $n$. Evaluating $Q$ on any input $w \in \{0, 1\}^n$ can be done in time polynomial in the number of lines of $Q$.} We now prove that NANDSAT is NP hard.

\textbf{Lemma 14.5} NANDSAT is NP hard.

\textbf{Proof Idea:} To prove Lemma 14.5 we need to show that for every $F \in NP$, $F \leq_p NANDSAT$. The high-level idea is that by the definition of NP, there is some NAND++ program $P^*$ and some polynomial $T(\cdot)$ such that $F(x) = 1$ if and only if there exists some $w \in \{0, 1\}^{|x|}$ such that $P^*(wx)$ outputs 1 within $T(|x|)$ steps. Now by “unrolling the loop” of the NAND++ program $P^*$ we can convert it into an $O(T(n))$ NAND program $Q'$ with $n + an^b$ inputs and a single output such that for every $x \in \{0, 1\}^n$ and $w \in \{0, 1\}^{an^b}$, $Q'(wx) = P^*(wx)$. on input $x \in \{0, 1\}$ that on input $w$ will simulate $P^*(wx)$ for $T(|x|)$ steps.
The next step is to hardwire the input $x$ to $Q'$ to obtain an $O(T(n))$ line NAND program $Q$ with $m = an^b$ inputs such that for every $w \in \{0,1\}^m$, $Q'(w) = Q(xw)$. By construction it will be the case that for every $x \in \{0,1\}^n$, $F(x) = 1$ if and only if there exists $w \in \{0,1\}^{an^b}$ such that $Q(w) = 1$, and hence this shows that $F \leq_p \text{NANDSAT}$. *

**Proof of Lemma 14.5.** We now present the details. Let $F \in \text{NP}$. To prove Lemma 14.5 we need to give a polynomial-time computable function that will map every $x^* \in \{0,1\}^*$ to a NAND program $Q$ such that $F(x) = \text{NANDSAT}(Q)$.

Let $x^* \in \{0,1\}^*$ be such a string and let $n = |x^*|$ be its length. By Definition 14.1 there exists $V \in \text{P}$ and $a, b \in \mathbb{N}$ such that $F(x^*) = 1$ if and only if there exists $w \in \{0,1\}^{an^b}$ such that $V(x^*w) = 1$.

Let $m = an^b$. Since $V \in \text{P}$ there is some NAND++ program $P^*$ that computes $V$ on inputs of the form $xw$ with $x \in \{0,1\}^n$ and $w \in \{0,1\}^m$ in at most $(n + m)^c$ time for some constant $c$. Using our “unrolling the loop NAND++ to NAND compiler” of Theorem 12.9, we can obtain a NAND program $Q'$ that has $n+m$ inputs and at most $O((n+m)^c)$ lines such that $Q'(xw) = P^*(xw)$ for every $x \in \{0,1\}^n$ and $w \in \{0,1\}^m$.

Now we can use the following simple but useful “hardwiring” technique to obtain a program:

**Lemma 14.6 — Hardwiring Lemma.** Given a $T$-line NAND program $Q'$ of $n + m$ inputs and $x^* \in \{0,1\}^n$, we can obtain in polynomial a program $Q$ with $m$ inputs of $T + 3$ lines such that for every $w \in \{0,1\}^m$, $Q(w) = Q'(x^*w)$.

**Proof.** To compute $Q$, we simply do a “search and replace” for all references in $Q'$ to $x[i]$ for $i \in [n]$, and transform them to either the variable zero or one depending on whether $x^*_i$ is equal to 0 or 1 respectively. By adding three lines to the beginning of $Q'$, we can ensure that the zero and one variables will have the correct value.
The only thing that then remains to do another search and replace to transform all references to the variables $X[\omega], \ldots, X[\omega + \omega - 1]$ to the variables $X[0], \ldots, X[\omega - 1]$ so that the $m$ inputs to the new program $Q$ will correspond to last $m$ inputs of the original program $Q'$. See Fig. 14.4 for an implementation of this reduction in Python.

Using Lemma 14.6, we obtain a program $Q$ of $m$ inputs such that $Q(w) = Q'(x^w) = P'(x^w)$ for every $w \in \{0, 1\}^m$. Since we know that $F(x^*) = 1$ if and only if there exists $w \in \{0, 1\}^m$ such that $P'(x^w) = 1$, this means that $F(x^*) = 1$ if and only if $NANDSAT(Q) = 1$, which is what we wanted to prove.

**Figure 14.4**: Given an $T$-line NAND program $Q$ that has $n + m$ inputs and some $x^* \in \{0, 1\}^n$, we can transform $Q$ into a $T + 3$ line NAND program $Q'$ that computes the map $w \mapsto Q(x^w)$ for $w \in \{0, 1\}^m$ by simply adding code to compute the zero and one constants, replacing all references to $X[i]$ with either zero or one depending on the value of $x^*_i$, and then replacing the remaining references to $X[j]$ with $X[j - n]$. Above is Python code that implements this transformation, as well as an example of its execution on a simple program.

### 14.4 THE 3$NAND$ PROBLEM

The 3$NAND$ problem is defined as follows: the input is a logical formula $\varphi$ on a set of variables $z_0, \ldots, z_{r-1}$ which is an AND of constraints of the form $z_i = NAND(z_j, z_k)$. For example, the following is a 3$NAND$ formula with 5 variables and 3 constraints:

$$(z_3 = NAND(z_0, z_2)) \land (z_1 = NAND(z_0, z_2)) \land (z_4 = NAND(z_3, z_1))$$

(14.5)

The output of 3$NAND$ on input $\varphi$ is 1 if and only if there is an assignment to the variables of $\varphi$ that makes it evaluate to “true” (that is, there is some assignment $z \in \{0, 1\}^r$ satisfying all of the constraints of $\varphi$). As usual, we can represent $\varphi$ as a string, and so think of 3$NAND$ as a function mapping $\{0, 1\}^*$ to $\{0, 1\}$. We now prove that 3$NAND$ is NP hard:

**Lemma 14.7** $NANDSAT \leq_p 3NAND$. 
**Proof Idea:** To prove Lemma 14.7 we need to give a polynomial-time map from every NAND program Q to a 3NAND formula Ψ such that there exists w such that Q(w) = 1 if and only if there exists z satisfying Ψ. For every line i of Q, we define a corresponding variable zi of Ψ. If the line i has the form \( \text{foo} = \text{NAND}(\text{bar}, \text{blah}) \) then we will add the clause \( z_i = \text{NAND}(z_j, z_k) \) where j and k are the last lines in which bar and blah were written to. We will also set variables corresponding to the input variables, as well as add a clause to ensure that the final output is 1. The resulting reduction can be implemented in about a dozen lines of Python, see ??.

**Figure 14.5:** Python code to reduce an instance \( Q \) of \( \text{NANDSAT} \) to an instance \( \Psi \) of \( 3\text{NAND} \). In the example above we transform the NAND program \( \text{xor5} \) which has 5 input variables and 16 lines, into a \( 3\text{NAND} \) formula \( \Psi \) that has 24 variables and 20 clauses. Since \( \text{xor5} \) outputs 1 on the input 1, 0, 0, 1, 1, there exists an assignment \( z \in \{0, 1\}^5 \) to \( \Psi \) such that \( (z_0, z_1, z_2, z_3, z_4) = (1, 0, 0, 1, 1) \) and \( \Psi \) evaluates to true on z.

**Proof of Lemma 14.7.** To prove Lemma 14.7 we need to give a reduction from \( \text{NANDSAT} \) to \( 3\text{NAND} \). Let Q be a NAND program with n inputs, one output, and m lines. We can assume without loss of generality that Q contains the variables one and zero as usual.

We map Q to a 3NAND formula Ψ as follows:

- Ψ has \( m + n \) variables \( z_0, \ldots, z_{m+n-1} \).
- The first \( n \) variables \( z_0, \ldots, z_{n-1} \) will correspond to the inputs of Q. The next \( m \) variables \( z_n, \ldots, z_{n+m-1} \) will correspond to the m lines of Q.
- For every \( \ell \in \{n, n+1, \ldots, n+m\} \), if the \( \ell - n \)-th line of the program Q is \( \text{foo} = \text{NAND}(\text{bar}, \text{blah}) \) then we add to Ψ the constraint
\[ z_\ell = \text{NAND}(z_j, z_k) \] where \( j - n \) and \( k - n \) correspond to the last lines in which the variables \text{bar} and \text{blah} (respectively) were written to. If one or both of \text{bar} and \text{blah} was not written to before then we use \( z_{\ell_0} \) instead of the corresponding value \( z_j \) or \( z_k \) in the constraint, where \( \ell_0 - n \) is the line in which \text{zero} is assigned a value. If one or both of \text{bar} and \text{blah} is an input variable \( X[i] \) then we use \( z_{\ell_0} \) instead of the corresponding value \( z_j \) or \( z_k \) in the constraint.

Let \( \ell^* \) be the last line in which the output \( y_0 \) is assigned a value. Then we add the constraint \( z_{\ell^*} = \text{NAND}(z_{\ell_0}, z_{\ell_0}) \) where \( \ell_0 - n \) is as above the last line in which \text{zero} is assigned a value. Note that this is effectively the constraint \( z_{\ell^*} = \text{NAND}(0, 0) = 1 \).

To complete the proof we need to show that there exists \( w \in \{0, 1\}^n \) s.t. \( Q(w) = 1 \) if and only if there exists \( z \in \{0, 1\}^{n+m} \) that satisfies all constraints in \( \Psi \). We now show both sides of this equivalence.

\textbf{Part I: Completeness.} Suppose that there is \( w \in \{0, 1\}^n \) s.t. \( Q(w) = 1 \). Let \( z \in \{0, 1\}^{n+m} \) be defined as follows: for \( i \in [n] \), \( z_i = w_i \) and for \( i \in \{n, n + 1, \ldots, n + m\} \) \( z_i \) equals the value that is assigned in the \((i - n)\)th line of \( Q \) when executed on \( w \). Then by construction \( z \) satisfies all of the constraints of \( \Psi \) (including the constraint that \( z_{\ell^*} = \text{NAND}(0, 0) = 1 \) since \( Q(w) = 1 \)).

\textbf{Part II: Soundness.} Suppose that there exists \( z \in \{0, 1\}^{n+m} \) satisfying \( \Psi \). Soundness will follow by showing that \( Q(z_0, \ldots, z_{n-1}) = 1 \) (and hence in particular there exists \( w \in \{0, 1\}^n \), namely \( z = z_0 \cdots z_{n-1} \), such that \( Q(w) = 1 \)). To do this we will prove the following claim \((*)\): for every \( \ell \in [m] \), \( z_{\ell+n} \) equals the value assigned in the \( \ell \)-th step of the execution of the program \( Q \) on \( z_0, \ldots, z_{n-1} \). Note that because \( z \) satisfies the constraints of \( \Psi \), \((*)\) is sufficient to prove the soundness condition since these constraints imply that the last value assigned to the variable \( y_0 \) in the execution of \( Q \) on \( z_0 \cdots w_{n-1} \) is equal to 1. To prove \((*)\) suppose, towards a contradiction, that it is false, and let \( \ell \) be the smallest number such that \( z_{\ell+n} \) is not equal to the value assigned in the \( \ell \)-th step of the execution of \( Q \) on \( z_0, \ldots, z_{n-1} \). But since \( z \) satisfies the constraints of \( \Psi \), we get that \( z_{\ell+n} = \text{NAND}(z_i, z_j) \) where (by the assumption above that \( \ell \) is smallest with this property) these values do correspond to the values last assigned to the variables on the righthand side of the assignment operator in the \( \ell \)-th line of the program. But this means that the value assigned in the \( \ell \)-th step is indeed simply the NAND of \( z_i \) and \( z_j \), contradicting our assumption on the choice of \( \ell \).
From $3NAND$ to $3SAT$

To conclude the proof of Theorem 14.3, we need to show Lemma 14.8 and show that $3NAND \leq_p 3SAT$:

**Lemma 14.8** $3NAND \leq_p 3SAT$.

**Proof Idea:** To prove Lemma 14.8 we need to map a 3NAND formula $\varphi$ into a 3SAT formula $\psi$ such that $\varphi$ is satisfiable if and only if $\psi$ is. The idea is that we can transform every NAND constraint of the form $a = NAND(b, c)$ into the AND of ORs involving the variables $a, b, c$ and their negations, where each of the ORs contains at most three terms. The construction is fairly straightforward, and the details are given below. *

It is a good exercise for you to try to find a 3CNF formula $\xi$ on three variables $a, b, c$ such that $\xi(a, b, c)$ is true if and only if $a = NAND(b, c)$. Once you do so, try to see why this implies a reduction from $3NAND$ to $3SAT$, and hence completes the proof of Lemma 14.8.

**Proof of Lemma 14.8.** The constraint

$$z_i = NAND(z_j, z_k) \quad (14.6)$$

is satisfied if $z_i = 1$ whenever $(z_j, z_k) \neq (1, 1)$. By going through all cases, we can verify that Eq. (14.6) is equivalent to the constraint

$$(\overline{z_i} \lor \overline{z_j} \lor \overline{z_k}) \land (z_i \lor z_j) \land (z_i \lor z_k) \quad (14.7)$$
Indeed if \( z_j = z_k = 1 \) then the first constraint of Eq. (14.7) is only true if \( z_i = 0 \). On the other hand, if either of \( z_j \) or \( z_k \) equals 0 then unless \( z_i = 1 \) either the second or third constraints will fail. This means that, given any 3NAND formula \( \varphi \) over \( n \) variables \( z_0, \ldots, z_{n-1} \), we can obtain a 3SAT formula \( \psi \) over the same variables by replacing every 3NAND constraint of \( \varphi \) with three 3OR constraints as in Eq. (14.7). Because of the equivalence of Eq. (14.6) and Eq. (14.7), the formula \( \psi \) satisfies that \( \psi(z_0, \ldots, z_{n-1}) = \varphi(z_0, \ldots, z_{n-1}) \) for every assignment \( z_0, \ldots, z_{n-1} \in \{0, 1\}^n \) to the variables. In particular \( \psi \) is satisfiable if and only if \( \varphi \) is, thus completing the proof.

### 14.6 WRAPPING UP

We have shown that for every function \( F \) in \( \textbf{NP} \), \( F \leq_p \text{NANDSAT} \leq_p 3\text{NAND} \leq_p 3\text{SAT} \), and so 3SAT is \( \textbf{NP} \)-hard. Since in Chapter 13 we saw that 3SAT \( \leq_p \text{QUADEQ}, 3\text{SAT} \leq_p \text{ISET}, 3\text{SAT} \leq_p \text{MAXCUT} \) and 3SAT \( \leq_p \text{LONGPATH} \), all these problems are \( \textbf{NP} \)-hard as well. Finally, since all the aforementioned problems are in \( \textbf{NP} \), they are all in fact \( \textbf{NP} \)-complete and have equivalent complexity. There are thousands of other natural problems that are \( \textbf{NP} \)-complete as well. Finding a polynomial-time algorithm for any one of them will imply a polynomial-time algorithm for all of them.

---

**Lecture Recap**

- Many of the problems for which we don’t know polynomial-time algorithms are \( \textbf{NP} \)-complete, which means that finding a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for all of them.
- It is conjectured that \( \textbf{NP} \neq \textbf{P} \) which means that we believe that polynomial-time algorithms for these problems are not merely unknown but are nonexistent.
- While an \( \textbf{NP} \)-hardness result means for example that a full-fledged “textbook” solution to a problem such as MAX-CUT that is as clean and gen-

---

Figure 14.7: Code and example output for the reduction given in Lemma 14.8 of 3NAND to 3SAT.
Figure 14.8: An instance of the independent set problem obtained by applying the reductions $NANDSAT \leq_p 3NAND \leq_p 3SAT \leq_p ISAT$ starting with the xor NAND program.

eral as the algorithm for MIN-CUT probably does not exist, it does not mean that we need to give up whenever we see a MAX-CUT instance. Later in this course we will discuss several strategies to deal with $NP$-hardness, including average-case complexity and approximation algorithms.

14.7 EXERCISES

**Disclaimer** Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

**Exercise 14.1 — Poor man’s Ladner’s Theorem.** Prove that if there is no $n^O(\log^2 n)$ time algorithm for $3SAT$ then there is some $F \in NP$ such that $F \notin P$ and $F$ is not $NP$ complete.\(^9\)

\(^9\) **Hint:** Use the function $F$ that on input a formula $\varphi$ and a string of the form $1^t$, outputs 1 if and only if $\varphi$ is satisfiable and $t = |\varphi|\log|\varphi|$.

14.8 BIBLIOGRAPHICAL NOTES

Eugene Lawler’s quote on the “mystical power of twoness” was taken from the wonderful book “The Nature of Computation” by Moore and Mertens. See also this memorial essay on Lawler by Lenstra.

\(^{10}\) TODO: credit surveys of Avi, Madhu
14.9 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

14.10 ACKNOWLEDGEMENTS
15
What if P equals NP?

“You don’t have to believe in God, but you should believe in The Book.”, Paul Erdős, 1985. ¹

“No more half measures, Walter”, Mike Ehrmantraut in “Breaking Bad”, 2010.

“The evidence in favor of \( P \neq \text{NP} \) and its algebraic counterpart is so overwhelming, and the consequences of their failure are so grotesque, that their status may perhaps be compared to that of physical laws rather than that of ordinary mathematical conjectures.”, Volker Strassen, laudation for Leslie Valiant, 1986.

“Suppose aliens invade the earth and threaten to obliterate it in a year’s time unless human beings can find the [fifth Ramsey number]. We could marshal the world’s best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the [sixth Ramsey number], however, we would have no choice but to launch a preemptive attack.”, Paul Erdős, as quoted by Graham and Spencer, 1990. ²

We have mentioned that the question of whether \( P = \text{NP} \), which is equivalent to whether there is a polynomial-time algorithm for \( 3SAT \), is the great open question of Computer Science. But why is it so important? In this chapter, we will try to figure out the implications of such an algorithm.

First, let us get one qualm out of the way. Sometimes people say, “What if \( P = \text{NP} \) but the best algorithm for \( 3SAT \) takes \( n^{100} \) time?” Well, \( n^{100} \) is much larger than, say, \( 2^{\sqrt{n}} \) for any input shorter than \( 10^{60} \) bits.

¹ Paul Erdős (1913-1996) was one of the most prolific mathematicians of all times. Though he was an atheist, Erdős often referred to “The Book” in which God keeps the most elegant proof of each mathematical theorem.

² The \( k \)-th Ramsey number, denoted as \( R(k, k) \), is the smallest number \( n \) such that for every graph \( G \) on \( n \) vertices, both \( G \) and its complement contain a \( k \)-sized independent set. If \( P = \text{NP} \) then we can compute \( R(k, k) \) in time polynomial in \( 2^k \), while otherwise it can potentially take closer to \( 2^{2^{2k}} \) steps.
which is way, way larger than the world’s total storage capacity (estimated at a “mere” $10^{21}$ bits or about 200 exabytes at the time of this writing). So another way to phrase this question is to say, “what if the complexity of 3SAT is exponential for all inputs that we will ever encounter, but then grows much smaller than that?” To me this sounds like the computer science equivalent of asking, “what if the laws of physics change completely once they are out of the range of our telescopes?” Sure, this is a valid possibility, but wondering about it does not sound like the most productive use of our time.

So, as the saying goes, we’ll keep an open mind, but not so open that our brains fall out, and assume from now on that:

- There is a mathematical god,
- and
- She does not “pussyfoot around” or take “half measures”. If God decided to make 3SAT easy, then 3SAT will have a $10^6 \cdot n$ (or at worst $10^6 n^2$)-time algorithm (i.e., 3SAT will be in \( \text{TIME}(cn) \) or \( \text{TIME}(cn^2) \) for a not-too-large constant \( c \)). If she decided to make 3SAT hard, then for every \( n \in \mathbb{N} \), 3SAT on \( n \) variables cannot be solved by a NAND program of fewer than \( 2^{10^{-6}n} \) lines.\(^3\)

So far, most of our evidence points to the latter possibility of 3SAT being exponentially hard, but we have not ruled out the former possibility either. In this chapter we will explore some of its consequences.

### 15.1 SEARCH-TO-DECISION REDUCTION

A priori, having a fast algorithm for 3SAT might not seem so impressive. Sure, it will allow us to decide the satisfiability of not just 3CNF formulas but also of quadratic equations, as well as find out whether there is a long path in a graph, and solve many other decision problems. But this is not typically what we want to do. It’s not enough to know if a formula is satisfiable— we want to discover the actual satisfying assignment. Similarly, it’s not enough to find out if a graph has a long path— we want to actually find the path.

It turns out that if we can solve these decision problems, we can solve the corresponding search problems as well:

#### Theorem 15.1 — Search vs Decision

Suppose that \( \text{P} = \text{NP} \). Then for every polynomial-time algorithm \( V \) and \( a, b \in \mathbb{N} \), there is a polynomial-time algorithm \( \text{FIND}_V \) such that for every \( x \in \{0, 1\}^a \), if there exists \( y \in \{0, 1\}^{an^2} \) satisfying \( V(xy) = 1 \), then \( \text{FIND}_V(x) \) finds some string \( y' \) satisfying this condition.
To understand what the statement of Theorem 15.1 means, let us look at the special case of the MAXCUT problem. It is not hard to see that there is a polynomial-time algorithm VERIFHCUT such that $\text{VERIFHCUT}(G, k, S) = 1$ if and only if $S$ is a subset of $G$'s vertices that cuts at least $k$ edges. Theorem 15.1 implies that if $P = \text{NP}$ then there is a polynomial-time algorithm FINDCUT that on input $G, k$ outputs a set $S$ such that $\text{VERIFHCUT}(G, k, S) = 1$ if such a set exists. This means that if $P = \text{NP}$, by trying all values of $k$ we can find in polynomial time a maximum cut in any given graph. We can use a similar argument to show that if $P = \text{NP}$ then we can find a satisfying assignment for every satisfiable 3CNF formula, find the longest path in a graph, solve integer programming, and so on and so forth.

Proof Idea: The idea behind the proof of Theorem 15.1 is simple; let us demonstrate it for the special case of $3SAT$. (In fact, this case is not so “special” — since $3SAT$ is $\text{NP}$-complete, we can reduce the task of solving the search problem for MAXCUT or any other problem in $\text{NP}$ to the task of solving it for $3SAT$.) Suppose that $P = \text{NP}$ and we are given a satisfiable 3CNF formula $\varphi$, and we now want to find a satisfying assignment $y$ for $\varphi$. Define $3SAT_0(\varphi)$ to output 1 if there is a satisfying assignment $y$ for $\varphi$ such that its first bit is 0, and similarly define $3SAT_1(\varphi) = 1$ if there is a satisfying assignment $y$ with $y_0 = 1$. The key observation is that both $3SAT_0$ and $3SAT_1$ are in $\text{NP}$, and so if $P = \text{NP}$ then we can compute them in polynomial time as well. Thus we can use this to find the first bit of the satisfying assignment. We can continue in this way to recover all the bits. ∗

Proof of Theorem 15.1. If $P = \text{NP}$ then for every polynomial-time algorithm $V$ and $a, b \in \mathbb{N}$, there is a polynomial-time algorithm $\text{STARTSWITH}_V$ that on input $x \in \{0, 1\}^*$ and $z \in \{0, 1\}^\ell$, outputs 1 if and only if there exists some $y \in \{0, 1\}^{anb}$ such that the first $\ell$ bits of $y$ are equal to $z$ and $V(xy) = 1$. Indeed, we leave it as an exercise to verify that the $\text{STARTSWITH}_V$ function is in $\text{NP}$ and hence can be solved in polynomial time if $P = \text{NP}$.

Now for any such polynomial-time $V$ and $a, b \in \mathbb{N}$, we can implement $\text{FIND}_V(x)$ as follows:

Algorithm $\text{FIND}_V$:

Input: $x \in \{0, 1\}^a$

Goal: Find $z \in \{0, 1\}^{anb}$ such that $V(xz) = 1$, if such $z$ exists.
Operation:

1. For $\ell = 0, \ldots, an^b - 1$ do the following:
   (a) Let $b_0 = \text{STARTSWITH}_V(xz_0 \cdots z_{\ell-1}0)$ and $b_1 = \text{STARTSWITH}_V(xz_0 \cdots z_{\ell-1}1)$
   (b) If $b_0 = 1$ then $z_\ell = 0$, otherwise $z_\ell = 1$.

2. Output $z_0, \ldots, z_{an^b-1}$.

To analyze the $FIND$ algorithm, note that it makes $2an^b$ invocations to $\text{STARTSWITH}_V$, and hence if the latter is polynomial-time, then so is $FIND_V$. Now suppose that $x$ is such that there exists some $y$ satisfying $V(xy) = 1$. We claim that at every step $\ell = 0, \ldots, an^b - 1$, we maintain the invariant that there exists $y \in \{0, 1\}^{an^b}$ whose first $\ell$ bits are $z$ s.t. $V(xy) = 1$. Note that this claim implies the theorem, since in particular it means that for $\ell = an^b - 1$, $z$ satisfies $V(xz) = 1$.

We prove the claim by induction. For $\ell = 0$, this holds vacuously. Now for every $\ell > 0$, if the call $\text{STARTSWITH}_V(xz_0 \cdots z_{\ell-1}0)$ returns 1, then we are guaranteed the invariant by definition of $\text{STARTSWITH}_V$. Now under our inductive hypothesis, there is $y_\ell, \ldots, y_{an^b-1}$ such that $P(xz_0, \ldots, z_{\ell-1}y_\ell, \ldots, y_{an^b-1}) = 1$. If the call to $\text{STARTSWITH}_V(xz_0 \cdots z_{\ell-1}0)$ returns 0 then it must be the case that $y_\ell = 1$, and hence when we set $z_\ell = 1$ we maintain the invariant.  

15.2 OPTIMIZATION

Theorem 15.1 allows us to find solutions for \textbf{NP} problems if $P = \textbf{NP}$, but it is not immediately clear that we can find the \textit{optimal} solution. For example, suppose that $P = \textbf{NP}$, and you are given a graph $G$. Can you find the \textit{longest} simple path in $G$ in polynomial time?

It turns out the answer is \textbf{Yes}. The idea is simple: if $P = \textbf{NP}$ then we can find out in polynomial time if an $n$-vertex graph $G$ contains a simple path of length $n$, and moreover, by Theorem 15.1, if $G$ does contain such a path, then we can find it. (Can you see why?) If $G$ does not contain a simple path of length $n$, then we will check if it contains a simple path of length $n - 1$, and continue in this way to find the largest $k$ such that $G$ contains a simple path of length $k$.

The above reasoning was not specifically tailored to finding paths in graphs. In fact, it can be vastly generalized to proving the following
Theorem 15.2 — Optimization from $\mathsf{P} = \mathsf{NP}$. Suppose that $\mathsf{P} = \mathsf{NP}$. Then for every polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ there is a polynomial-time algorithm $\mathit{OPT}$ such that on input $x \in \{0,1\}^*$, $\mathit{OPT}(x,1^m) = \max_{y \in \{0,1\}^m} f(x,y)$ (where we identify the output of $f(x)$ with a natural number via the binary representation).

Moreover under the same assumption, there is a polynomial-time algorithm $\mathit{FINDOPT}$ such that for every $x \in \{0,1\}^*$, $\mathit{FINDOPT}(x,1^m)$ outputs $y^* \in \{0,1\}^*$ such that $f(x,y^*) = \mathit{OPT}(x,y^*)$.

The statement of Theorem 15.2 is a bit cumbersome. To understand it, think how it would subsume the example above of a polynomial time algorithm for finding the maximum length path in a graph. In this case the function $f$ would be the map that on input a pair $x,y$ outputs 0 if the pair $(x,y)$ does not represent some graph and a simple path inside the graph respectively; otherwise $f(x,y)$ would equal the length of the path $y$ in the graph $x$. Since a path in an $n$ vertex graph can be represented by at most $n \log n$ bits, for every $x$ representing a graph of $n$ vertices, finding $\max_{y \in \{0,1\}^{n \log n}} f(x,y)$ corresponds to finding the length of the maximum simple path in the graph corresponding to $x$, and finding the string $y^*$ that achieves this maximum corresponds to actually finding the path.

Proof Idea: The proof follows by generalizing our ideas from the longest path example above. Let $f$ be as in the theorem statement. If $\mathsf{P} = \mathsf{NP}$ then for every for every string $x \in \{0,1\}^*$ and number $k$, we can test in in $\text{poly}(|x|,m)$ time whether there exists $y$ such that $f(x,y) \geq k$, or in other words test whether $\max_{y \in \{0,1\}^m} f(x,y) \geq k$. If $f(x,y)$ is an integer between 0 and $\text{poly}(|x| + |y|)$ (as is the case in the example of longest path) then we can just try out all possibilities for $k$ to find the maximum number $k$ for which $\max_y f(x,y) \geq k$. Otherwise, we can use binary search to hone down on the right value. Once we do so, we can use search-to-decision to actually find the string $y^*$ that achieves the maximum. ⋆

Proof of Theorem 15.2. For every $f$ as in the theorem statement, we can
define the Boolean function $F : \{0, 1\}^* \to \{0, 1\}$ as follows.

$$F(x, 1^m, k) = \begin{cases} 
1 & \exists y \in \{0, 1\}^m f(x, y) \geq k \\
0 & \text{otherwise}
\end{cases} \quad (15.1)$$

Since $f$ is computable in polynomial time, $F$ is in $\mathbf{NP}$, and so under our assumption that $P = \mathbf{NP}$, $F$ itself can be computed in polynomial time. Now, for every $x$ and $m$, we can compute the largest $k$ such that $F(x, 1^m, k) = 1$ by a binary search. Specifically, we will do this as follows:

1. We maintain two numbers $a, b$ such that we are guaranteed that $a \leq \max_{y \in \{0, 1\}^m} f(x, y) < b$.
2. Initially we set $a = 0$ and $b = 2^{T(n)}$ where $T(n)$ is the running time of $f$. (A function with $T(n)$ running time can’t output more than $T(n)$ bits and so can’t output a number larger than $2^{T(n)}$.)
3. At each point in time, we compute the midpoint $c = \lfloor (a + b)/2 \rfloor$ and let $y = F(1^n, c)$.
   (a) If $y = 1$ then we set $a = c$ and leave $b$ as it is.
   (b) If $y = 0$ then we set $b = c$ and leave $a$ as it is.

We then go back to step 3, until $b \leq a + 1$.

Since $|b - a|$ shrinks by a factor of 2, within $\log_2 2^{T(n)} = T(n)$ steps, we will get to the point at which $b \leq a + 1$, and then we can simply output $a$. Once we find the maximum value of $k$ such that $F(x, 1^m, k) = 1$, we can use the search to decision reduction of Theorem 15.1 to obtain the actual value $y^* \in \{0, 1\}^m$ such that $f(x, y^*) = k$.

Example 15.3 — Integer programming. One application for Theorem 15.2 is in solving optimization problems. For example, the task of linear programming is to find $y \in \mathbb{R}^n$ that maximizes some linear objective $\sum_{i=0}^{n-1} c_i y_i$ subject to the constraint that $y$ satisfies linear inequalities of the form $\sum_{i=0}^{n-1} a_i y_i \leq c$. As we discussed in Section 11.1.3, there is a known polynomial-time algorithm for linear programming. However, if we want to place additional constraints on $y$, such as requiring the coordinates of $y$ to be integer or 0/1 valued then the best-known algorithms run in exponential time in the worst case. However, if $P = \mathbf{NP}$ then Theorem 15.2 tells us that we would be able to solve all problems of this form in polynomial time. For every string $x$ that describes a set of constraints
and objective, we will define a function $f$ such that if $y$ satisfies the constraints of $x$ then $f(x, y)$ is the value of the objective, and otherwise we set $f(x, y) = -M$ where $M$ is some large number. We can then use Theorem 15.2 to compute the $y$ that maximizes $f(x, y)$ and that will give us the assignment for the variables that satisfies our constraints and maximizes the objective. (If the computation results in $y$ such that $f(x, y) = -M$ then we can double $M$ and try again; if the true maximum objective is achieved by some string $y^*$, then eventually $M$ will be large enough so that $-M$ would be smaller than the objective achieved by $y^*$, and hence when we run procedure of Theorem 15.2 we would get a value larger than $-M$.)

**Need for binary search.** In many examples, such as the case of finding longest path, we don’t need to use the binary search step in Theorem 15.2, and can simply enumerate over all possible values for $k$ until we find the correct one. One example where we do need to use this binary search step is in the case of the problem of finding a maximum length path in a weighted graph. This is the problem where $G$ is a weighted graph, and every edge of $G$ is given a weight which is a number between 0 and $2^k$. Theorem 15.2 shows that we can find the maximum-weight simple path in $G$ (i.e., simple path maximizing the sum of the weights of its edges) in time polynomial in the number of vertices and in $k$.

Beyond just this example there is a vast field of mathematical optimization that studies problems of the same form as in Theorem 15.2. In the context of optimization, $x$ typically denotes a set of constraints over some variables (that can be Boolean, integer, or real valued), $y$ encodes an assignment to these variables, and $f(x, y)$ is the value of some objective function that we want to maximize. Given that we don’t know efficient algorithms for NP complete problems, researchers in optimization research study special cases of functions $f$ (such as linear programming and semidefinite programming) where it is possible to optimize the value efficiently. Optimization is widely used in a great many scientific areas including machine learning, engineering, economics and operations research.

**15.2.1 Example: Supervised learning**

One classical optimization task is supervised learning. In supervised learning we are given a list of examples $x_0, x_1, ..., x_{m-1}$ (where we can think of each $x_i$ as a string in $\{0, 1\}^n$ for some $n$) and the labels for them $y_0, ..., y_{n-1}$ (which we will think of simply bits, i.e.,
y_i \in \{0, 1\}). For example, we can think of the \(x_i\)'s as images of either dogs or cats, for which \(y_i = 1\) in the former case and \(y_i = 0\) in the latter case. Our goal is to come up with a hypothesis or predictor \(h : \{0, 1\}^n \rightarrow \{0, 1\}\) such that if we are given a new example \(x\) that has an (unknown to us) label \(y\), then with high probability \(h\) will predict the label. That is, with high probability it will hold that \(h(x) = y\). The idea in supervised learning is to use the Occam’s Razor principle: the simplest hypothesis that explains the data is likely to be correct. There are several ways to model this, but one popular approach is to pick some fairly simple function \(H : \{0, 1\}^{k+n} \rightarrow \{0, 1\}\). We think of the first \(k\) inputs as the parameters and the last \(n\) inputs as the example data. (For example, we can think of the first \(k\) inputs of \(H\) as specifying the weights and connections for some neural network that will then be applied on the latter \(n\) inputs.) We can then phrase the supervised learning problem as finding, given a set of labeled examples \(S = \{(x_0, y_0), \ldots, (x_m, y_m)\}\), the set of parameters \(\theta_0, \ldots, \theta_{k-1} \in \{0, 1\}\) that minimizes the number of errors made by the predictor \(x \mapsto H(\theta, x)\).

In other words, we can define for every set \(S\) as above the function \(F_S : \{0, 1\}^k \rightarrow [m]\) such that \(F_S(\theta) = \sum_{(x,y) \in S} |H(\theta, x) - y|\). Now, finding the value \(\theta\) that minimizes \(F_S(\theta)\) is equivalent to solving the supervised learning problem with respect to \(H\). For every polynomial-time computable \(H : \{0, 1\}^{k+n} \rightarrow \{0, 1\}\), the task of minimizing \(F_S(\theta)\) can be “massaged” to fit the form of Theorem 15.2 and hence if \(P = NP\), then we can solve the supervised learning problem in great generality. In fact, this observation extends to essentially any learning model, and allows for finding the optimal predictors given the minimum number of examples. (This is in contrast to many current learning algorithms, which often rely on having access to an extremely large number of examples—far beyond the minimum needed, and in particular far beyond the number of examples humans use for the same tasks.)

15.2.2 Example: Breaking cryptosystems

We will discuss cryptography later in this course, but it turns out that if \(P = NP\) then almost every cryptosystem can be efficiently broken. One approach is to treat finding an encryption key as an instance of a supervised learning problem. If there is an encryption scheme that maps a “plaintext” message \(p\) and a key \(\theta\) to a “ciphertext” \(c\), then given examples of ciphertext/plaintext pairs of the form \((c_0, p_0), \ldots, (c_m, p_m)\), our goal is to find the key \(\theta\) such that \(E(\theta, p_i) = c_i\) where \(E\) is the encryption algorithm. While you might think getting such “labeled examples” is unrealistic, it turns out (as many amateur homebrew crypto designers learn the hard way) that
this is actually quite common in real-life scenarios, and that it is also possible to relax the assumption to having more minimal prior information about the plaintext (e.g., that it is English text). We defer a more formal treatment to Chapter 20.

15.3 FINDING MATHEMATICAL PROOFS

In the context of Gödel’s Theorem, we discussed the notion of a proof system (see Section 10.1). Generally speaking, a proof system can be thought of as an algorithm \( V : \{0, 1\}^* \to \{0, 1\} \) (known as the verifier) such that given a statement \( x \in \{0, 1\}^* \) and a candidate proof \( w \in \{0, 1\}^* \), \( V(x, w) = 1 \) if and only if \( w \) encodes a valid proof for the statement \( x \). Any type of proof system that is used in mathematics for geometry, number theory, analysis, etc., is an instance of this form. In fact, standard mathematical proof systems have an even simpler form where the proof \( w \) encodes a sequence of lines \( w^0, \ldots, w^n \) (each of which is itself a binary string) such that each line \( w^i \) is either an axiom or follows from some prior lines through an application of some inference rule. For example, Peano’s axioms encode a set of axioms and rules for the natural numbers, and one can use them to formalize proofs in number theory. Also, there are some even stronger axiomatic systems, the most popular one being Zermelo–Fraenkel with the Axiom of Choice or ZFC for short. Thus, although mathematicians typically write their papers in natural language, proofs of number theorists can typically be translated to ZFC or similar systems, and so in particular the existence of an \( n \)-page proof for a statement \( x \) implies that there exists a string \( w \) of length \( \text{poly}(n) \) (in fact often \( O(n) \) or \( O(n^2) \)) that encodes the proof in such a system. Moreover, because verifying a proof simply involves going over each line and checking that it does indeed follow from the prior lines, it is fairly easy to do that in \( O(|w|) \) or \( O(|w|^2) \) (where as usual \( |w| \) denotes the length of the proof \( w \)). This means that for every reasonable proof system \( V \), the following function \( \text{SHORTPROOF}_V : \{0, 1\}^* \to \{0, 1\} \) is in \( \text{NP} \), where for every input of the form \( x1^m \), \( \text{SHORTPROOF}_V(x, 1^m) = 1 \) if and only if there exists \( w \in \{0, 1\}^* \) with \( |w| \leq m \) s.t. \( V(xyw) = 1 \). That is, \( \text{SHORTPROOF}_V(x, 1^m) = 1 \) if there is a proof (in the system \( V \)) of length at most \( m \) bits that \( x \) is true. Thus, if \( P = \text{NP} \), then despite Gödel’s Incompleteness Theorems, we can still automate mathematics in the sense of finding proofs that are not too long for every statement that has one. (Frankly speaking, if the shortest proof for some statement requires a terabyte, then human mathematicians won’t ever find this proof either.) For this reason, Gödel himself felt that the question of whether \( \text{SHORTPROOF}_V \) has a polynomial time algorithm is of great interest. As Gödel wrote in a letter to John von Neumann in 1956 (before the concept of \( \text{NP} \) or even “polynomial time” was formally
The undecidability of Entscheidungsproblem refers to the uncomputability of the function that maps a statement in first order logic to 1 if and only if that statement has a proof. One can obviously easily construct a Turing machine, which for every formula $F$ in first order predicate logic and every natural number $n$, allows one to decide if there is a proof of $F$ of length $n$ (length = number of symbols). Let $\psi(F, n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_p \psi(F, n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi \geq k \cdot n$ [for some constant $k > 0$]. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number $n$ so large that when the machine does not deliver a result, it makes no sense to think more about the problem.

For many reasonable proof systems (including the one that Gödel referred to), $\text{SHORT PROOF}_V$ is in fact $\text{NP}$-complete, and so Gödel can be thought of as the first person to formulate the $P$ vs $\text{NP}$ question. Unfortunately, the letter was only discovered in 1988.

15.4 QUANTIFIER ELIMINATION (ADVANCED)

If $P = \text{NP}$ then we can solve all $\text{NP}$ search and optimization problems in polynomial time. But can we do more? It turns out that the answer is that Yes we can!

An $\text{NP}$ decision problem can be thought of as the task of deciding, given some string $x \in \{0, 1\}^*$ the truth of a statement of the form

$$\exists y \in \{0, 1\}^{p(|x|)} V(xy) = 1 \quad (15.2)$$

for some polynomial-time algorithm $V$ and polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$. That is, we are trying to determine, given some string $x$, whether there exists a string $y$ such that $x$ and $y$ satisfy some polynomial-time checkable condition $V$. For example, in the independent set problem, the string $x$ represents a graph $G$ and a number $k$, the string $y$ represents some subset $S$ of $G$’s vertices, and the condition that we check is whether $|S| \geq k$ and there is no edge $\{u, v\}$ in $G$ such that both $u \in S$ and $v \in S$.

We can consider more general statements such as checking, given a
string \( x \in \{0,1\}^* \), the truth of a statement of the form
\[
\exists y \in \{0,1\}^{p_0(|x|)} \forall z \in \{0,1\}^{p_1(|x|)} V(xyz) = 1 ,
\]
which in words corresponds to checking, given some string \( x \), whether there exists a string \( y \) such that for every string \( z \), the triple \((x, y, z)\) satisfy some polynomial-time checkable condition. We can also consider more levels of quantifiers such as checking the truth of the statement
\[
\exists y \in \{0,1\}^{p_0(|x|)} \forall z \in \{0,1\}^{p_1(|x|)} \exists w \in \{0,1\}^{p_2(|x|)} V(xyzw) = 1
\]
and so on and so forth.

For example, given an \( n \)-input NAND program \( P \), we might want to find the smallest NAND program \( P' \) that computes the same function as \( P \). The question of whether there is such a \( P' \) that can be described by a string of at most \( s \) bits can be phrased as
\[
\exists P' \in \{0,1\}^s \forall x \in \{0,1\}^n P'(x) = P(x)
\]
which has the form Eq. (15.3).\(^7\) Another example of a statement involving \( \ell \) levels of quantifiers would be to check, given a chess position \( x \), whether there is a strategy that guarantees that White wins within \( \ell \) steps. For example is \( \ell = 3 \) we would want to check if given the board position \( x \), there exists a move \( y \) for White such that for every move \( z \) for Black there exists a move \( w \) for White that ends in a a checkmate.

It turns out that if \( P = NP \) then we can solve these kinds of problems as well:

**Theorem 15.4 — Polynomial hierarchy collapse.** If \( P = NP \) then for every \( a \in \mathbb{N} \), polynomial \( p : \mathbb{N} \to \mathbb{N} \) and polynomial-time algorithm \( V \), there is a polynomial-time algorithm \( SOLVE_{V,a} \) that on input \( x \in \{0,1\}^n \) returns 1 if and only if
\[
\exists y \in \{0,1\}^m \forall y_i \in \{0,1\}^m \cdots Q y_{a-1} \in \{0,1\}^m V(x_0 y_1 \cdots y_{a-1}) = 1
\]
where \( m = p(n) \) and \( Q \) is either \( \exists \) or \( \forall \) depending on whether \( a \) is odd or even, respectively.\(^8\)

**Proof Idea:** To understand the idea behind the proof, consider the special case where we want to decide, given \( x \in \{0,1\}^n \), whether for every \( y \in \{0,1\}^n \) there exists \( z \in \{0,1\}^n \) such that \( V(xyz) = 1 \). Consider the function \( F \) such that \( F(xy) = 1 \) if there exists \( z \in \{0,1\}^n \) such that \( V(xyz) = 1 \). Since \( V \) runs in polynomial-time \( F \in NP \) and hence if \( P = NP \), then there is an algorithm \( V' \) that on input \( x, y \) outputs 1 if and only if there exists \( z \in \{0,1\}^n \) such that \( V(xyz) = 1 \).

\(^7\) Since NAND programs are equivalent to Boolean circuits, the search problem corresponding to Eq. (15.5) known as the circuit minimization problem and is widely studied in Engineering. You can skip ahead to Section 15.4.1 to see a particularly compelling application of this.

\(^8\) For the ease of notation, we assume that all the strings we quantify over have the same length \( m = p(n) \), but using simple padding one can show that this captures the general case of strings of different polynomial lengths.
Now we can see that the original statement we consider is true if and only if for every \( y \in \{0,1\}^n \), \( V'(xy) = 1 \), which means it is false if and only if the following condition (\(*\)) holds: there exists some \( y \in \{0,1\}^n \) such that \( V'(xy) = 0 \). But for every \( x \in \{0,1\}^n \), the question of whether the condition (\(*\)) is itself in \( \text{NP} \) (as we assumed \( V' \) can be computed in polynomial time) and hence under the assumption that \( P = \text{NP} \) we can determine in polynomial time whether the condition (\(*\)), and hence our original statement, is true.

\[ \text{Proof of Theorem 15.4.} \] We prove the theorem by induction. We assume that there is a polynomial-time algorithm \( \mathcal{L}_a \) that can solve the problem Eq. (15.6) for \( a - 1 \) and use that to solve the problem for \( a \). For \( a = 1 \), \( \text{SOLVE}_{V,a-1}(x) = 1 \) if and only if \( V(x) = 1 \) which is a polynomial-time computation since \( V \) runs in polynomial time. For every \( x, y_0 \), define the statement \( \varphi_{x,y_0} \) to be the following:

\[
\varphi_{x,y_0} = \forall y_1 \in \{0,1\}^n \exists y_2 \in \{0,1\}^m \cdots \exists y_{a-1} \in \{0,1\}^m V(xy_0 y_1 \cdots y_{a-1}) = 1 \tag{15.7}
\]

By the definition of \( \text{SOLVE}_{V,a} \), for every \( x \in \{0,1\}^n \), our goal is that \( \text{SOLVE}_{V,a}(x) = 1 \) if and only if there exists \( y_0 \in \{0,1\}^m \) such that \( \varphi_{x,y_0} \) is true.

The negation of \( \varphi_{x,y_0} \) is the statement

\[
\overline{\varphi}_{x,y_0} = \exists y_1 \in \{0,1\}^n \forall y_2 \in \{0,1\}^m \cdots \exists y_{a-1} \in \{0,1\}^m V(xy_0 y_1 \cdots y_{a-1}) = 0 \tag{15.8}
\]

where \( \overline{\varphi} \) is \( \exists \) if \( \varphi \) was \( \forall \) and \( \overline{\varphi} \) is \( \forall \) otherwise. (Please stop and verify that you understand why this is true, this is a generalization of the fact that if \( \Psi \) is some logical condition then the negation of \( \exists y \forall z \Psi(y,z) \) is \( \forall y \exists z \neg \Psi(y,z) \).)

The crucial observation is that \( \overline{\varphi}_{x,y_0} \) is exactly a statement of the form we consider with \( a - 1 \) quantifiers instead of \( a \), and hence by our inductive hypothesis there is some polynomial time algorithm \( \overline{S} \) that on input \( xy_0 \) outputs 1 if and only if \( \overline{\varphi}_{x,y_0} \) is true. If we let \( S \) be the algorithm that on input \( x, y_0 \) outputs \( 1 - \overline{S}(xy_0) \) then we see that \( S \) outputs 1 if and only if \( \varphi_{x,y_0} \) is true. Hence we can rephrase the original statement Eq. (15.6) as follows:

\[
\exists y_0 \in \{0,1\}^m S(xy_0) = 1 \tag{15.9}
\]

but since \( S \) is a polynomial-time algorithm, Eq. (15.9) is clearly a statement in \( \text{NP} \) and hence under our assumption that \( P = \text{NP} \) there is a polynomial time algorithm that on input \( x \in \{0,1\}^n \), will determine if Eq. (15.9) is true and so also if the original statement Eq. (15.6) is true.
The algorithm of Theorem 15.4 can also solve the search problem as well: find the value $y_0$ that certifies the truth of Eq. (15.6). We note that while this algorithm is in polynomial time, the exponent of this polynomial blows up quite fast. If the original NANDSAT algorithm required $\Omega(n^2)$ time, solving $a$ levels of quantifiers would require time $\Omega(n^{2^a})$.9

15.4.1 Application: self improving algorithm for 3SAT
Suppose that we found a polynomial-time algorithm $A$ for 3SAT that is “good but not great”. For example, maybe our algorithm runs in time $cn^2$ for some not too small constant $c$. However, it’s possible that the best possible SAT algorithm is actually much more efficient than that. Perhaps, as we guessed before, there is a circuit $C^*$ of at most $10^9n$ gates that computes 3SAT on $n$ variables, and we simply haven’t discovered it yet. We can use Theorem 15.4 to “bootstrap” our original “good but not great” 3SAT algorithm to discover the optimal one. The idea is that we can phrase the question of whether there exists a size $s$ circuit that computes 3SAT for all length $n$ inputs as follows: there exists a size $\leq s$ circuit $C$ such that for every formula $\varphi$ described by a string of length at most $n$, if $C(\varphi) = 1$ then there exists an assignment $x$ to the variables of $\varphi$ that satisfies it. One can see that this is a statement of the form Eq. (15.4) and hence if $P = NP$ we can solve it in polynomial time as well. We can therefore imagine investing huge computational resources in running $A$ one time to discover the circuit $C^*$ and then using $C^*$ for all further computation.

15.5 APPROXIMATING COUNTING PROBLEMS (ADVANCED, OPTIONAL)
Given a NAND program $P$, if $P = NP$ then we can find an input $x$ (if one exists) such that $P(x) = 1$. But what if there is more than one $x$ like that? Clearly we can’t efficiently output all such $x$’s; there might be exponentially many. But we can get an arbitrarily good multiplicative approximation (i.e., a $1 \pm \epsilon$ factor for arbitrarily small $\epsilon > 0$) for the number of such $x$’s, as well as output a (nearly) uniform member of this set. We defer the details to later in this course, when we learn about randomized computation. However, we state (without proof) the following theorem for now:

**Theorem 15.5** — Approximate counting if $P = NP$. Let $V : \{0,1\}^* \rightarrow \{0,1\}$ be some polynomial-time algorithm, and suppose that $P = NP$. Then there exists an algorithm $\text{COUNT}_V$ that on input $x,1^m,\epsilon$, runs in time polynomial in $|x|,m,1/\epsilon$ and outputs a number

---

9 We do not know whether such loss is inherent. As far as we can tell, it’s possible that the quantified boolean formula problem has a linear-time algorithm. We will, however, see later in this course that it satisfies a notion known as PSPACE-hardness that is even stronger than NP-hardness.
$K \in \{0, \ldots, 2^m\}$ such that

$$(1 - \epsilon) K \leq \left| \{ y \in \{0,1\}^m : V(xy) = 1 \} \right| \leq (1 + \epsilon) K \quad (15.10)$$

That is, $K$ gives an approximation up to a factor of $1 \pm \epsilon$ for the number of witnesses for $x$ with respect to the verifying algorithm $V$.

Once again, to understand this theorem it can be useful to see how it implies that if $P = NP$ then there is a polynomial-time algorithm that given a graph $G$ and a number $k$, can compute a number $K$ that is within a $1 \pm 0.01$ factor equal to the number of simple paths in $G$ of length $k$. (That is, $K$ is between 0.99 to 1.01 times the number of such paths.)

### 15.6 WHAT DOES ALL OF THIS IMPLY?

So, what will happen if we have a $10^6 n$ algorithm for $3SAT$? We have mentioned that NP-hard problems arise in many contexts, and indeed scientists, engineers, programmers and others routinely encounter such problems in their daily work. A better $3SAT$ algorithm will probably make their lives easier, but that is the wrong place to look for the most foundational consequences. Indeed, while the invention of electronic computers did of course make it easier to do calculations that people were already doing with mechanical devices and pen and paper, the main applications computers are used for today were not even imagined before their invention.

An exponentially faster algorithm for all NP problems would be no less radical an improvement (and indeed, in some sense would be more) than the computer itself, and it is as hard for us to imagine what it would imply as it was for Babbage to envision today’s world. For starters, such an algorithm would completely change the way we program computers. Since we could automatically find the “best” (in any measure we chose) program that achieves a certain task, we would not need to define how to achieve a task, but only specify tests as to what would be a good solution, and could also ensure that a program satisfies an exponential number of tests without actually running them.

The possibility that $P = NP$ is often described as “automating creativity”. There is something to that analogy, as we often think of a creative solution as one that is hard to discover but that, once the “spark” hits, is easy to verify. But there is also an element of hubris to that statement, implying that the most impressive consequence
of such an algorithmic breakthrough will be that computers would succeed in doing something that humans already do today. In fact, creativity already is to a large extent automated or minimized (e.g., just see how much popular media content is mass-produced), and as in most professions we should expect to see the need for humans diminish with time even if $P \neq NP$.

Nevertheless, artificial intelligence, like many other fields, will clearly be greatly impacted by an efficient 3SAT algorithm. For example, it is clearly much easier to find a better Chess-playing algorithm when, given any algorithm $P$, you can find the smallest algorithm $P'$ that plays Chess better than $P$. Moreover, as we mentioned above, much of machine learning (and statistical reasoning in general) is about finding “simple” concepts that explain the observed data, and if $NP = P$, we could search for such concepts automatically for any notion of “simplicity” we see fit. In fact, we could even “skip the middle man” and do an automatic search for the learning algorithm with smallest generalization error. Ultimately the field of Artificial Intelligence is about trying to “shortcut” billions of years of evolution to obtain artificial programs that match (or beat) the performance of natural ones, and a fast algorithm for $NP$ would provide the ultimate shortcut.\(^\text{10}\)

More generally, a faster algorithm for $NP$ problems would be immensely useful in any field where one is faced with computational or quantitative problems— which is basically all fields of science, math, and engineering. This will not only help with concrete problems such as designing a better bridge, or finding a better drug, but also with addressing basic mysteries such as trying to find scientific theories or “laws of nature”. In a fascinating talk, physicist Nima Arkani-Hamed discusses the effort of finding scientific theories in much the same language as one would describe solving an $NP$ problem, for which the solution is easy to verify or seems “inevitable”, once found, but that requires searching through a huge landscape of possibilities to reach, and that often can get “stuck” at local optima:

\[\text{“the laws of nature have this amazing feeling of inevitability... which is associated with local perfection.”}\]

\[\text{“The classical picture of the world is the top of a local mountain in the space of ideas. And you go up to the top and it looks amazing up there and absolutely incredible. And you learn that there is a taller mountain out there. Find it, Mount Quantum... they're not smoothly connected... you've got to make a jump to go from classical to quantum... This also tells you why we have such}\]
Finding an efficient algorithm for \( \text{NP} \) amounts to always being able to search through an exponential space and find not just the “local” mountain, but the tallest peak.

But perhaps more than any computational speedups, a fast algorithm for \( \text{NP} \) problems would bring about a new type of understanding. In many of the areas where \( \text{NP} \)-completeness arises, it is not as much a barrier for solving computational problems as it is a barrier for obtaining “closed-form formulas” or other types of more constructive descriptions of the behavior of natural, biological, social and other systems. A better algorithm for \( \text{NP} \), even if it is “merely” \( 2^{\sqrt{n}} \)-time, seems to require obtaining a new way to understand these types of systems, whether it is characterizing Nash equilibria, spin-glass configurations, entangled quantum states, or any of the other questions where \( \text{NP} \) is currently a barrier for analytical understanding. Such new insights would be very fruitful regardless of their computational utility.

### 15.7 CAN \( \text{P} \neq \text{NP} \) BE NEITHER TRUE NOR FALSE?

The Continuum Hypothesis is a conjecture made by Georg Cantor in 1878, positing the non-existence of a certain type of infinite cardinality.\(^{11}\) This was considered one of the most important open problems in set theory, and settling its truth or falseness was the first problem put forward by Hilbert in the 1900 address we mentioned before. However, using the theories developed by Gödel and Turing, in 1963 Paul Cohen proved that both the Continuum Hypothesis and its negation are consistent with the standard axioms of set theory (i.e., the Zermelo-Fraenkel axioms + the Axiom of choice, or “ZFC” for short).\(^{12}\)

Today, many (though not all) mathematicians interpret this result as saying that the Continuum Hypothesis is neither true nor false, but rather is an axiomatic choice that we are free to make one way or the other. Could the same hold for \( \text{P} \neq \text{NP} \)?

In short, the answer is No. For example, suppose that we are trying to decide between the “3SAT is easy” conjecture (there is an \( 10^9 n \) time algorithm for 3SAT) and the “3SAT is hard” conjecture (for every \( n \), any NAND program that solves \( n \) variable 3SAT takes \( 2^{10^{-6} n} \) lines). Then, since for \( n = 10^9 \), \( 2^{10^{-6} n} > 10^6 n \), this boils down to the finite question of deciding whether or not there is a \( 10^{13} \)-line NAND

---

\(^{11}\) One way to phrase it is that for every infinite subset \( S \) of the real numbers \( \mathbb{R} \), either there is a one-to-one and onto function \( f : S \to \mathbb{R} \) or there is a one-to-one and onto function \( f : S \to \mathbb{N} \).

\(^{12}\) Formally, what he proved is that if ZFC is consistent, then so is ZFC when we assume either the continuum hypothesis or its negation.
program deciding 3SAT on formulas with $10^8$ variables. If there is
such a program then there is a finite proof of its existence, namely the
approximately 1TB file describing the program, and for which the ver-
ification is the (finite in principle though infeasible in practice) process
of checking that it succeeds on all inputs. If there isn’t such a pro-
gram, then there is also a finite proof of that, though any such proof
would take longer since we would need to enumerate over all programs
as well. Ultimately, since it boils down to a finite statement about bits
and numbers; either the statement or its negation must follow from
the standard axioms of arithmetic in a finite number of arithmetic
steps. Thus, we cannot justify our ignorance in distinguishing between
the “3SAT easy” and “3SAT hard” cases by claiming that this might be
an inherently ill-defined question. Similar reasoning (with different
numbers) applies to other variants of the P vs NP question. We note
that in the case that 3SAT is hard, it may well be that there is no short
proof of this fact using the standard axioms, and this is a question
that people have been studying in various restricted forms of proof
systems.

15.8 IS P = NP “IN PRACTICE”?

The fact that a problem is NP-hard means that we believe there is no
efficient algorithm that solve it in the worst case. It does not, however,
mean that every single instance of the problem is hard. For exam-
ple, if all the clauses in a 3SAT instance $\varphi$ contain the same variable
$x_i$ (possibly in negated form), then by guessing a value to $x_i$ we can
reduce $\varphi$ to a 2SAT instance which can then be efficiently solved. Gen-
eralizations of this simple idea are used in “SAT solvers”, which are
algorithms that have solved certain specific interesting SAT formulas
with thousands of variables, despite the fact that we believe SAT to
be exponentially hard in the worst case. Similarly, a lot of problems
arising in economics and machine learning are NP-hard. And yet
vendors and customers manage to figure out market-clearing prices
(as economists like to point out, there is milk on the shelves) and mice
succeed in distinguishing cats from dogs. Hence people (and ma-
chines) seem to regularly succeed in solving interesting instances of
NP-hard problems, typically by using some combination of guessing
while making local improvements.

It is also true that there are many interesting instances of NP-hard
problems that we do not currently know how to solve. Across all ap-
lication areas, whether it is scientific computing, optimization, con-
trol or more, people often encounter hard instances of NP problems
on which our current algorithms fail. In fact, as we will see, all of our
digital security infrastructure relies on the fact that some concrete and
easy-to-generate instances of, say, 3SAT (or, equivalently, any other

13 This inefficiency is not necessarily inherent. Later in this course we may
discuss results in program-checking, interactive proofs, and average-case
complexity, that can be used for efficient verification of proofs of related state-
ments. In contrast, the inefficiency of verifying failure of all programs could
well be inherent.

14 Actually, the computational difficulty
of problems in economics such as find-
ing optimal (or any) equilibria is quite
subtle. Some variants of such problems
are NP-hard, while others have a certain
“intermediate” complexity.
NP-hard problem) are exponentially hard to solve.

Thus it would be wrong to say that \( NP \) is easy “in practice”, nor would it be correct to take \( NP \)-hardness as the “final word” on the complexity of a problem, particularly when we have more information about how any given instance is generated. Understanding both the “typical complexity” of \( NP \) problems, as well as the power and limitations of certain heuristics (such as various local-search based algorithms) is a very active area of research. We will see more on these topics later in this course.

15.9 WHAT IF \( P \neq NP \)?

So, \( P = NP \) would give us all kinds of fantastical outcomes. But we strongly suspect that \( P \neq NP \), and moreover that there is no much-better-than-brute-force algorithm for 3SAT. If indeed that is the case, is it all bad news?

One might think that impossibility results, telling you that you cannot do something, is the kind of cloud that does not have a silver lining. But in fact, as we already alluded to before, it does. A hard (in a sufficiently strong sense) problem in \( NP \) can be used to create a code that cannot be broken, a task that for thousands of years has been the dream of not just spies but of many scientists and mathematicians over the generations. But the complexity viewpoint turned out to yield much more than simple codes, achieving tasks that people had previously not even dared to dream of. These include the notion of public key cryptography, allowing two people to communicate securely without ever having exchanged a secret key; electronic cash, allowing private and secure transaction without a central authority; and secure multiparty computation, enabling parties to compute a joint function on private inputs without revealing any extra information about it. Also, as we will see, computational hardness can be used to replace the role of randomness in many settings.

Furthermore, while it is often convenient to pretend that computational problems are simply handed to us, and that our job as computer scientists is to find the most efficient algorithm for them, this is not how things work in most computing applications. Typically even formulating the problem to solve is a highly non-trivial task. When we discover that the problem we want to solve is \( NP \)-hard, this might be a useful sign that we used the wrong formulation for it.

Beyond all these, the quest to understand computational hardness – including the discoveries of lower bounds for restricted computational models, as well as new types of reductions (such as those arising from “probabilistically checkable proofs”) – has already had surprising positive applications to problems in algorithm design, as

---

15 Talk more about coping with NP hardness. Main two approaches are heuristics such as SAT solvers that succeed on some instances, and proxy measures such as mathematical relaxations that instead of solving problem \( X \) (e.g., an integer program) solve program \( X' \) (e.g., a linear program) that is related to that. Maybe give compressed sensing as an example, and least square minimization as a proxy for maximum a posteriori probability.
well as in coding for both communication and storage. This is not surprising since, as we mentioned before, from group theory to the theory of relativity, the pursuit of impossibility results has often been one of the most fruitful enterprises of mankind.

Lecture Recap

- The question of whether \( P = NP \) is one of the most important and fascinating questions of computer science and science at large, touching on all fields of the natural and social sciences, as well as mathematics and engineering.

- Our current evidence and understanding supports the “SAT hard” scenario that there is no much-better-than-brute-force algorithm for 3SAT or many other \( NP \)-hard problems.

- We are very far from proving this, however. Researchers have studied proving lower bounds on the number of gates to compute explicit functions in restricted forms of circuits, and have made some advances in this effort, along the way generating mathematical tools that have found other uses. However, we have made essentially no headway in proving lower bounds for general models of computation such as NAND and NAND++ programs. Indeed, we currently do not even know how to rule out the possibility that for every \( n \in \mathbb{N} \), SAT restricted to \( n \)-length inputs has a NAND program of \( 10n \) lines (even though there exist \( n \)-input functions that require \( 2^n/(10n) \) lines to compute).

- Understanding how to cope with this computational intractability, and even benefit from it, comprises much of the research in theoretical computer science.

15.10 EXERCISES

Disclaimer  Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.
15.11 BIBLIOGRAPHICAL NOTES

15.12 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

• Polynomial hierarchy hardness for circuit minimization and related problems, see for example this paper.

15.13 ACKNOWLEDGEMENTS
Space bounded computation

PLAN: Example of space bounded algorithms, importance of preserving space. The classes L and PSPACE, space hierarchy theorem, PSPACE=NPSPACE, constant space = regular languages.

16.1 LECTURE SUMMARY

16.2 EXERCISES

Disclaimer  Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

16.3 BIBLIOGRAPHICAL NOTES

16.4 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

16.5 ACKNOWLEDGEMENTS
IV

RANDOMIZED COMPUTATION
17
Probability Theory 101

“God doesn’t play dice with the universe”, Albert Einstein

“Einstein was doubly wrong … not only does God definitely play dice, but He sometimes confuses us by throwing them where they can’t be seen.”, Stephen Hawking

“‘The probability of winning a battle’ has no place in our theory because it does not belong to any [random experiment]. Probability cannot be applied to this problem any more than the physical concept of work can be applied to the ‘work’ done by an actor reciting his part.”, Richard Von Mises, 1928 (paraphrased)

“I am unable to see why ‘objectivity’ requires us to interpret every probability as a frequency in some random experiment; particularly when in most problems probabilities are frequencies only in an imaginary universe invented just for the purpose of allowing a frequency interpretation.”, E.T. Jaynes, 1976

---

Before we show how to use randomness in algorithms, let us do a quick review of some basic notions in probability theory. This is not meant to replace a course on probability theory, and if you have not seen this material before, I highly recommend you look at additional resources to get up to speed.\(^1\) Fortunately, we will not need many of the advanced notions of probability theory, but, as we will see, even the so-called “simple” setting of tossing \(n\) coins can lead to very subtle and interesting issues.

---

\(^1\) Harvard’s STAT 110 class (whose lectures are available on youtube) is a highly recommended introduction to probability. See also these lecture notes from MIT’s “Mathematics for Computer Science” course.
17.1 RANDOM COINS

The nature of randomness and probability is a topic of great philosophical, scientific and mathematical depth. Is there actual randomness in the world, or does it proceed in a deterministic clockwork fashion from some initial conditions set at the beginning of time? Does probability refer to our uncertainty of beliefs, or to the frequency of occurrences in repeated experiments? How can we define probability over infinite sets?

These are all important questions that have been studied and debated by scientists, mathematicians, statisticians and philosophers. Fortunately, we will not need to deal directly with these questions here. We will be mostly interested in the setting of tossing \( n \) random, unbiased and independent coins. Below we define the basic probabilistic objects of \textit{events} and \textit{random variables} when restricted to this setting. These can be defined for much more general probabilistic experiments or \textit{sample spaces}, and later on we will briefly discuss how this can be done. However, the \( n \)-coin case is sufficient for almost everything we’ll need in this course.

If instead of “heads” and “tails” we encode the sides of each coin by “zero” and “one”, we can encode the result of tossing \( n \) coins as a string in \( \{0,1\}^n \). Each particular outcome \( x \in \{0,1\}^n \) is obtained with probability \( 2^{-n} \). For example, if we toss three coins, then we obtain each of the 8 outcomes 000, 001, 010, 011, 100, 101, 110, 111 with probability \( 2^{-3} = 1/8 \) (see also Fig. 17.1). We can describe the experiment of tossing \( n \) coins as choosing a string \( x \) uniformly at random from \( \{0,1\}^n \), and hence we’ll use the shorthand \( x \sim \{0,1\}^n \) for \( x \) that is chosen according to this experiment.

![Figure 17.1: The probabilistic experiment of tossing three coins corresponds to making 2 \( \times \) 2 \( \times \) 2 = 8 choices, each with equal probability. In this example, the blue set corresponds to the event \( A = \{ x \in \{0,1\}^3 \mid x_0 = 0 \} \) where the first coin toss is equal to 0, and the pink set corresponds to the event \( B = \{ x \in \{0,1\}^3 \mid x_1 = 1 \} \) where the second coin toss is equal to 1 (with their intersection having a purplish color). As we can see, each of these events contains 4 elements (out of 8 total) and so has probability \( 1/2 \). The intersection of \( A \) and \( B \) contains two elements, and so the probability that both of these events occur is \( \frac{2}{8} = \frac{1}{4} \).

An \textit{event} is simply a subset \( A \) of \( \{0,1\}^n \). The \textit{probability} of \( A \), de-
noted by \( \Pr_{x \sim \{0,1\}^n} [A] \) (or \( \Pr[A] \) for short, when the sample space is understood from the context), is the probability that an \( x \) chosen uniformly at random will be contained in \( A \). Note that this is the same as \( |A|/2^n \) (where \( |A| \) as usual denotes the number of elements in the set \( A \)). For example, the probability that \( x \) has an even number of ones is \( \Pr[x \sim \{0,1\}^n \mid \sum_{i=0}^{n-1} x_i \equiv 0 \ (\text{mod} \ 2)] \). In the case \( n = 3 \), \( A = \{000, 011, 101, 110\} \), and hence \( \Pr[A] = \frac{4}{8} = \frac{1}{2} \). It turns out this is true for every \( n \):

**Lemma 17.1**

\[
\Pr_{x \sim \{0,1\}^n} \left[ \sum_{i=0}^{n-1} x_i \text{ is even} \right] = \frac{1}{2} \tag{17.1}
\]

To test your intuition on probability, try to stop here and prove the lemma on your own.

**Proof of Lemma 17.1.** Let \( A = \{ x \in \{0,1\}^n : \sum_{i=0}^{n-1} x_i = 0 \ (\text{mod} \ 2) \} \). Since every \( x \) is obtained with probability \( 2^{-n} \), to show this we need to show that \( |A| = 2^n/2 = 2^{n-1} \). For every \( x_0, \ldots, x_{n-2} \), if \( \sum_{i=0}^{n-2} x_i \) is even then \((x_0, \ldots, x_{n-1}, 0) \in A \) and \((x_0, \ldots, x_{n-1}, 1) \notin A \). Similarly, if \( \sum_{i=0}^{n-2} x_i \) is odd then \((x_0, \ldots, x_{n-1}, 1) \in A \) and \((x_0, \ldots, x_{n-1}, 0) \notin A \). Hence, for every one of the \( 2^{n-1} \) prefixes \((x_0, \ldots, x_{n-2})\), there is exactly a single continuation of \((x_0, \ldots, x_{n-2})\) that places it in \( A \).

We can also use the *intersection* \( (\cap) \) and *union* \( (\cup) \) operators to talk about the probability of both event \( A \) and event \( B \) happening, or the probability of event \( A \) or event \( B \) happening. For example, the probability \( p \) that \( x \) has an even number of ones and \( x_0 = 1 \) is the same as \( \Pr[A \cap B] \) where \( A = \{ x \in \{0,1\}^n : \sum_{i=0}^{n-1} x_i = 0 \ (\text{mod} \ 2) \} \) and \( B = \{ x \in \{0,1\}^n : x_0 = 1 \} \). This probability is equal to \( 1/4 \). (It is a great exercise for you to pause here and verify that you understand why this is the case.)

Because intersection corresponds to considering the logical AND of the conditions that two events happen, while union corresponds to considering the logical OR, we will sometimes use the \( \wedge \) and \( \lor \) operators instead of \( \cap \) and \( \cup \), and so write this probability \( p = \Pr[A \cap B] \) defined above also as

\[
\Pr_{x \sim \{0,1\}^n} \left[ \sum_{i} x_i \equiv 0 \ (\text{mod} \ 2) \land x_0 = 1 \right]. \tag{17.2}
\]

If \( A \subseteq \{0,1\}^n \) is an event, then \( \overline{A} = \{0,1\}^n \setminus A \) corresponds to the event that \( A \) does not happen. Since \( |\overline{A}| = 2^n - |A| \), we get that

\[
\Pr[\overline{A}] = \frac{|\overline{A}|}{2^n} = \frac{2^n-|A|}{2^n} = 1 - \frac{|A|}{2^n} = 1 - \Pr[A] \tag{17.3}
\]

If \( A \subseteq \{0,1\}^n \) is an event, then \( \overline{A} = \{0,1\}^n \setminus A \) corresponds to the event that \( A \) does not happen. Since \( |\overline{A}| = 2^n - |A| \), we get that

\[
\Pr[\overline{A}] = \frac{|\overline{A}|}{2^n} = \frac{2^n-|A|}{2^n} = 1 - \frac{|A|}{2^n} = 1 - \Pr[A] \tag{17.3}
\]
This makes sense: since \( A \) happens if and only if \( \overline{A} \) does not happen, the probability of \( \overline{A} \) should be one minus the probability of \( A \).

**Remember the sample space** While the above definition might seem very simple and almost trivial, the human mind seems not to have evolved for probabilistic reasoning, and it is surprising how often people can get even the simplest settings of probability wrong. One way to make sure you don’t get confused when trying to calculate probability statements is to always ask yourself the following two questions: (1) Do I understand what is the sample space that this probability is taken over?, and (2) Do I understand what is the definition of the event that we are analyzing?

For example, suppose that I were to randomize seating in my course, and then it turned out that students sitting in row 7 performed better on the final: how surprising should we find this? If we started out with the hypothesis that there is something special about the number 7 and chose it ahead of time, then the event that we are discussing is the event \( A \) that students sitting in number 7 had better performance on the final, and we might find it surprising. However, if we first looked at the results and then chose the row whose average performance is best, then the event we are discussing is the event \( B \) that there exists some row where the performance is higher than the overall average. \( B \) is a superset of \( A \), and its probability (even if there is no correlation between sitting and performance) can be quite significant.

### 17.1.1 Random variables

*Events* correspond to Yes/No questions, but often we want to analyze finer questions. For example, if we make a bet at the roulette wheel, we don’t want to just analyze whether we won or lost, but also *how much* we’ve gained. A (real valued) *random variable* is simply a way to associate a number with the result of a probabilistic experiment. Formally, a random variable is simply a function \( X : \{0,1\}^n \rightarrow \mathbb{R} \) that maps every outcome \( x \in \{0,1\}^n \) to a real number \( X(x) \).

For example, the function \( \text{sum} : \{0,1\}^n \rightarrow \mathbb{R} \) that maps \( x \) to the sum of its coordinates (i.e., to \( \sum_{i=0}^{n-1} x_i \)) is a random variable.

The expectation of a random variable \( X \), denoted by \( \mathbb{E}[X] \), is the average value that that this number takes, taken over all draws from the probabilistic experiment. In other words, the expectation of \( X \) is defined as follows:

\[
\mathbb{E}[X] = \sum_{x \in \{0,1\}^n} 2^{-n} X(x) . \tag{17.4}
\]
If $X$ and $Y$ are random variables, then we can define $X + Y$ as simply the random variable that maps a point $x \in \{0, 1\}^n$ to $X(x) + Y(x)$. One basic and very useful property of the expectation is that it is linear:

**Lemma 17.2 — Linearity of expectation.**

$$
\mathbb{E}[X + Y] = \mathbb{E}[X] + \mathbb{E}[Y]
$$

**Proof.**

$$
\mathbb{E}[X + Y] = \sum_{x \in \{0, 1\}^n} 2^{-n} (X(x) + Y(x)) = \\
\sum_{x \in \{0, 1\}^n} 2^{-n} X(x) + \sum_{x \in \{0, 1\}^n} 2^{-n} Y(x) = \\
\mathbb{E}[X] + \mathbb{E}[Y]
$$

Similarly, $\mathbb{E}[kX] = k \mathbb{E}[X]$ for every $k \in \mathbb{R}$. For example, using the linearity of expectation, it is very easy to show that the expectation of the sum of the $x_i$’s for $x \sim \{0, 1\}^n$ is equal to $n/2$. Indeed, if we write $X = \sum_{i=0}^{n-1} x_i$ then $X = X_0 + \cdots + X_{n-1}$ where $X_i$ is the random variable $x_i$. Since for every $i$, $\Pr[X_i = 0] = 1/2$ and $\Pr[X_i = 1] = 1/2$, we get that $\mathbb{E}[X_i] = (1/2) \cdot 0 + (1/2) \cdot 1 = 1/2$ and hence $\mathbb{E}[X] = \sum_{i=0}^{n-1} \mathbb{E}[X_i] = n \cdot (1/2) = n/2$.

If you have not seen discrete probability before, please go over this argument again until you are sure you follow it; it is a prototypical simple example of the type of reasoning we will employ again and again in this course.

If $A$ is an event, then $1_A$ is the random variable such that $1_A(x)$ equals 1 if $x \in A$, and $1_A(x) = 0$ otherwise. Note that $\Pr[A] = \mathbb{E}[1_A]$ (can you see why?). Using this and the linearity of expectation, we can show one of the most useful bounds in probability theory:

**Lemma 17.3 — Union bound.** For every two events $A, B$, $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$
Proof of Lemma 17.3. For every $x$, the variable $1_{A \cup B}(x) \leq 1_A(x) + 1_B(x)$. Hence, $\Pr[A \cup B] = \mathbb{E}[1_{A \cup B}] \leq \mathbb{E}[1_A + 1_B] = \mathbb{E}[1_A] + \mathbb{E}[1_B] = \Pr[A] + \Pr[B]$. ■

The way we often use this in theoretical computer science is to argue that, for example, if there is a list of 100 bad events that can happen, and each one of them happens with probability at most $1/10000$, then with probability at least $1 - 100/10000 = 0.99$, no bad event happens.

Figure 17.2: The union bound tells us that the probability of $A$ or $B$ happening is at most the sum of the individual probabilities. We can see it by noting that for every two sets $|A \cup B| \leq |A| + |B|$ (with equality only if $A$ and $B$ have no intersection).

17.1.2 Distributions over strings
While most of the time we think of random variables as having as output a real number, we sometimes consider random variables whose output is a string. That is, we can think of a map $Y : \{0, 1\}^n \rightarrow \{0, 1\}^*$ and consider the “random variable” $Y$ such that for every $y \in \{0, 1\}^*$, the probability that $Y$ outputs $y$ is equal to $\frac{1}{2^n} |\{x \in \{0, 1\}^n \mid Y(x) = y\}|$. To avoid confusion, we will typically refer to such string-valued random variables as distributions over strings. So, a distribution $Y$ over strings $\{0, 1\}^*$ can be thought of as a finite collection of strings $y_0, \ldots, y_{M-1} \in \{0, 1\}^*$ and probabilities $p_0, \ldots, p_{M-1}$ (which are non-negative numbers summing up to one), so that $\Pr[Y = y_i] = p_i$.

Two distributions $Y$ and $Y'$ are identical if they assign the same probability to every string. For example, consider the following two functions $Y, Y' : \{0, 1\}^2 \rightarrow \{0, 1\}^2$. For every $x \in \{0, 1\}^2$, we define $Y(x) = x$ and $Y'(x) = x_0 (x_0 \oplus x_1)$ where $\oplus$ is the XOR operations. Although these are two different functions, they induce the same distribution over $\{0, 1\}^2$ when invoked on a uniform input. The distribution
Y(x) for x ~ \{0, 1\}^2 is of course the uniform distribution over \{0, 1\}^2.
On the other hand Y' is simply the map 00 ↦ 00, 01 ↦ 01, 10 ↦ 11, 11 ↦ 10 which is a permutation over the map F : \{0, 1\}^2 → \{0, 1\}^2
declared as F(x_0x_1) = x_0x_1 and the map G : \{0, 1\}^2 → \{0, 1\}^2 defined
as G(x_0x_1) = x_0(x_0 \oplus x_1).

17.1.3 More general sample spaces.
While in this chapter we assume that the underlying probabilistic
experiment corresponds to tossing n independent coins, everything
we say easily generalizes to sampling x from a more general finite or
countable set S (and not-so-easily generalizes to uncountable sets S as
well). A probability distribution over a finite set S is simply a function
\mu : S → [0, 1] such that \sum_{x \in S} \mu(x) = 1. We think of this as the
experiment where we obtain every x ∈ S with probability \mu(x), and
sometimes denote this as x ~ \mu. An event A is a subset of S, and the
probability of A, which we denote by Pr_\mu[A], is \sum_{x \in A} \mu(x). A random
variable is a function X : S → R, where the probability that X = y is
equal to \sum_{x \in S \text{ s.t. } X(x) = y} \mu(x).

17.2 CORRELATIONS AND INDEPENDENCE
One of the most delicate but important concepts in probability is the
notion of independence (and the opposing notion of correlations). Subtle
correlations are often behind surprises and errors in probability and
statistical analysis, and several mistaken predictions have been blamed
on miscalculating the correlations between, say, housing prices in
Florida and Arizona, or voter preferences in Ohio and Michigan. See
also Joe Blitzstein’s aptly named talk “Conditioning is the Soul of
Statistics”.

Two events A and B are independent if the fact that A happens
makes B neither more nor less likely to happen. For example, if we
think of the experiment of tossing 3 random coins x ∈ \{0, 1\}^3, and we
let A be the event that x_0 = 1 and B the event that x_0 + x_1 + x_2 ≥ 2,
then if A happens it is more likely that B happens, and hence these
events are not independent. On the other hand, if we let C be the event
that x_1 = 1, then because the second coin toss is not affected by the
result of the first one, the events A and C are independent.

The formal definition is that events A and B are independent if
Pr[A \cap B] = Pr[A] · Pr[B]. If Pr[A \cap B] > Pr[A] · Pr[B] then we say
that A and B are positively correlated, while if Pr[A \cap B] < Pr[A] · Pr[B]
then we say that A and B are negatively correlated (see Fig. 17.1).

If we consider the above examples on the experiment of choosing

3 TODO: add exercise on simulating die tosses and choosing a random number
by coin tosses

4 Another thorny issue is of course the difference between correlation and
causation. Luckily, this is another point we don’t need to worry about in our
clean setting of tossing n coins.
x ∈ \{0, 1\}^3 then we can see that

\[
\Pr[x_0 = 1] = \frac{1}{2}
\]

\[
\Pr[x_0 + x_1 + x_2 \geq 2] = \Pr[\{011, 101, 110, 111\}] = \frac{4}{8} = \frac{1}{2}
\]

(17.7)

but

\[
\Pr[x_0 = 1 \land x_0 + x_1 + x_2 \geq 2] = \Pr[\{101, 110, 111\}] = \frac{3}{8} > \frac{1}{2} \cdot \frac{1}{2}
\]

(17.8)

and hence, as we already observed, the events \{x_0 = 1\} and \{x_0 + x_1 + x_2 \geq 2\} are not independent and in fact are positively correlated. On the other hand, \Pr[x_0 = 1 \land x_1 = 1] = \Pr[\{110, 111\}] = \frac{2}{8} = \frac{1}{2} \cdot \frac{1}{2}

and hence the events \{x_0 = 1\} and \{x_1 = 1\} are indeed independent.

**Disjointness vs independence** People sometimes confuse the notion of disjointness and independence, but these are actually quite different. Two events A and B are disjoint if A ∩ B = ∅, which means that if A happens then B definitely does not happen. They are independent if \Pr[A ∩ B] = \Pr[A] \cdot \Pr[B] which means that knowing that A happens gives us no information about whether B happened or not. If A and B have nonzero probability, then being disjoint implies that they are not independent, since in particular it means that they are negatively correlated.
Conditional probability: If \(A\) and \(B\) are events, and \(A\) happens with nonzero probability then we define the probability that \(B\) happens \emph{conditioned on} \(A\) to be \(\Pr[B|A] = \Pr[A \cap B]/\Pr[A]\). This corresponds to calculating the probability that \(B\) happens if we already know that \(A\) happened. Note that \(A\) and \(B\) are independent if and only if \(\Pr[B|A] = \Pr[B]\).

More than two events: We can generalize this definition to more than two events. We say that events \(A_1, \ldots, A_k\) are \emph{mutually independent} if knowing that any set of them occurred or didn’t occur does not change the probability that an event outside the set occurs. Formally, the condition is that for every subset \(I \subseteq [k],\)
\[
\Pr[\bigwedge_{i \in I} A_i] = \prod_{i \in I} \Pr[A_i].
\] (17.9)

For example, if \(x \sim \{0, 1\}^3\), then the events \(\{x_0 = 1\}, \{x_1 = 1\}\) and \(\{x_2 = 1\}\) are mutually independent. On the other hand, the events \(\{x_0 = 1\}, \{x_1 = 1\}\) and \(\{x_0 + x_1 = 0 \mod 2\}\) are \emph{not} mutually independent, even though every pair of these events is independent (can you see why? see also Fig. 17.4).

Figure 17.4: Consider the sample space \(\{0, 1\}^n\) and the events \(A, B, C, D, E\) corresponding to \(A: x_0 = 1, B: x_1 = 1, C: x_0 + x_1 + x_2 \geq 2, D: x_0 + x_1 + x_2 = 0 \mod 2\) and \(D: x_0 + x_1 = 0 \mod 2\). We can see that \(A\) and \(B\) are independent, \(C\) is positively correlated with \(A\) and positively correlated with \(B\), the three events \(A, B, D\) are mutually independent, and while every pair out of \(A, B, E\) is independent, the three events \(A, B, E\) are not mutually independent since their intersection has probability \(
\frac{2}{8} = \frac{1}{4}\)
instead of \(\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}\).

17.2.1 Independent random variables
We say that two random variables \(X : \{0, 1\}^n \to \mathbb{R}\) and \(Y : \{0, 1\}^n \to \mathbb{R}\) are independent if for every \(u, v \in \mathbb{R}\), the events \(\{X = u\}\) and \(\{Y = v\}\) are independent.\(^5\) In other words, \(X\) and \(Y\) are independent if \(\Pr[X = u \land Y = v] = \Pr[X = u] \Pr[Y = v]\) for every \(u, v \in \mathbb{R}\). For example, if \(X\) is a Bernoulli random variable with parameter \(p\), then \(\Pr[X = u \land Y = v] = \Pr[X = u] \Pr[Y = v] = p \cdot q\) if \(u = 0, v = 0\) or \(u = 1, v = 1\) and \(0\) otherwise.

\(^5\)We use \(\{X = u\}\) as shorthand for \(\{x \mid X(x) = u\}\).
two random variables depend on the result of tossing different coins then they are independent:

**Lemma 17.4** Suppose that $S = \{s_0, \ldots, s_{k-1}\}$ and $T = \{t_0, \ldots, t_{m-1}\}$ are disjoint subsets of $\{0, \ldots, n-1\}$ and let $X, Y : \{0, 1\}^n \rightarrow \mathbb{R}$ be random variables such that $X = F(x_{s_0}, \ldots, x_{s_{k-1}})$ and $Y = G(x_{t_0}, \ldots, x_{t_{m-1}})$ for some functions $F : \{0, 1\}^k \rightarrow \mathbb{R}$ and $G : \{0, 1\}^m \rightarrow \mathbb{R}$. Then $X$ and $Y$ are independent.

The notation in the lemma’s statement is a bit cumbersome, but at the end of the day, it simply says that if $X$ and $Y$ are random variables that depend on two disjoint sets $S$ and $T$ of coins (for example, $X$ might be the sum of the first $n/2$ coins, and $Y$ might be the largest consecutive stretch of zeroes in the second $n/2$ coins), then they are independent.

**Proof of Lemma 17.4.** Let $a, b \in \mathbb{R}$, and let $A = \{x \in \{0, 1\}^k : F(x) = a\}$ and $B = \{x \in \{0, 1\}^m : F(x) = b\}$. Since $S$ and $T$ are disjoint, we can reorder the indices so that $S = \{0, \ldots, k-1\}$ and $T = \{k, \ldots, k+m-1\}$ without affecting any of the probabilities. Hence we can write $\Pr[X = a \land X = b] = |C|/2^n$ where $C = \{x_0, \ldots, x_{n-1} : (x_0, \ldots, x_{k-1}) \in A \land (x_k, \ldots, x_{k+m-1}) \in B\}$. Another way to write this using string concatenation is that $C = \{xyz : x \in A, y \in B, z \in \{0, 1\}^{n-k-m}\}$, and hence $|C| = |A||B|2^{n-k-m}$, which means that

$$\frac{|C|}{2^n} = \frac{|A|}{2^k} \frac{|B|}{2^{n-k-m}} = \Pr[X = a] \Pr[Y = b]. \quad (17.10)$$

Note that if $X$ and $Y$ are independent random variables then (if we let $S_X, S_Y$ denote all the numbers that have positive probability of being the output of $X$ and $Y$, respectively) it holds that:

$$\mathbb{E}[XY] = \sum_{a \in S_X, b \in S_Y} \Pr[X = a \land Y = b] \cdot ab = (1) \sum_{a \in S_X, b \in S_Y} \Pr[X = a] \Pr[Y = b] \cdot ab = (2)$$

$$\left( \sum_{a \in S_X} \Pr[X = a] \cdot a \right) \left( \sum_{b \in S_Y} \Pr[Y = b] \cdot b \right) = (3)$$

$$\mathbb{E}[X] \mathbb{E}[Y] \quad (17.11)$$

where the first equality $(=1)$ follows from the independence of $X$ and $Y$, the second equality $(=2)$ follows by “opening the parentheses” of the righthand side, and the third inequality $(=3)$ follows from the definition of expectation. (This is not an "if and only if"; see Exercise 17.3.)
Another useful fact is that if \( X \) and \( Y \) are independent random variables, then so are \( F(X) \) and \( G(Y) \) for all functions \( F, G : \mathbb{R} \to \mathbb{R} \). This is intuitively true since learning \( F(X) \) can only provide us with less information than does learning \( X \) itself. Hence, if learning \( X \) does not teach us anything about \( Y \) (and so also about \( F(Y) \)) then neither will learning \( F(X) \). Indeed, to prove this we can write for every \( a, b \in \mathbb{R} \):

\[
\Pr[F(X) = a \land G(Y) = b] = \sum_{x \text{ s.t. } F(x) = a, y \text{ s.t. } G(y) = b} \Pr[X = x \land Y = y] = \\
\sum_{x \text{ s.t. } F(x) = a, y \text{ s.t. } G(y) = b} \Pr[X = x] \Pr[Y = y] = \\
\left( \sum_{x \text{ s.t. } F(x) = a} \Pr[X = x] \right) \cdot \left( \sum_{y \text{ s.t. } G(y) = b} \Pr[Y = y] \right) = \\
\Pr[F(X) = a] \Pr[G(Y) = b].
\]

(17.12)

**17.2.2 Collections of independent random variables.**

We can extend the notions of independence to more than two random variables: we say that the random variables \( X_0, \ldots, X_{n-1} \) are mutually independent if for every \( a_0, \ldots, a_{n-1} \in \mathbb{R} \),

\[
\Pr[X_0 = a_0 \land \cdots \land X_{n-1} = a_{n-1}] = \Pr[X_0 = a_0] \cdots \Pr[X_{n-1} = a_{n-1}].
\]

(17.13)

And similarly, we have that

**Lemma 17.5 — Expectation of product of independent random variables.** If \( X_0, \ldots, X_{n-1} \) are mutually independent then

\[
\mathbb{E}[\prod_{i=0}^{n-1} X_i] = \prod_{i=0}^{n-1} \mathbb{E}[X_i].
\]

(17.14)

**Lemma 17.6 — Functions preserve independence.** If \( X_0, \ldots, X_{n-1} \) are mutually independent, and \( Y_0, \ldots, Y_{n-1} \) are defined as \( Y_i = F_i(X_i) \) for some functions \( F_0, \ldots, F_{n-1} : \mathbb{R} \to \mathbb{R} \), then \( Y_0, \ldots, Y_{n-1} \) are mutually independent as well.

We leave proving Lemma 17.5 and Lemma 17.6 as Exercise 17.4 Exercise 17.5. It is good idea for you stop now and do these exercises to make sure you are comfortable with the notion of independence, as we will use it heavily later on in this course.
17.3 CONCENTRATION

The name “expectation” is somewhat misleading. For example, suppose that you and I place a bet on the outcome of 10 coin tosses, where if they all come out to be 1’s then I pay you 100,000 dollars and otherwise you pay me 10 dollars. If we let $X : \{0, 1\}^{10} \to \mathbb{R}$ be the random variable denoting your gain, then we see that

$$E[X] = 2^{-10} \cdot 100000 - (1 - 2^{-10})^{10} \sim 90.$$ (17.15)

But we don’t really “expect” the result of this experiment to be for you to gain 90 dollars. Rather, 99.9% of the time you will pay me 10 dollars, and you will hit the jackpot 0.01% of the times.

However, if we repeat this experiment again and again (with fresh and hence independent coins), then in the long run we do expect your average earning to be 90 dollars, which is the reason why casinos can make money in a predictable way even though every individual bet is random. For example, if we toss $n$ coins, then as $n$ grows, the number of coins that come up ones will be more and more concentrated around $n/2$ according to the famous “bell curve” (see Fig. 17.5).

Figure 17.5: The probabilities that we obtain a particular sum when we toss $n = 10, 20, 100, 1000$ coins converge quickly to the Gaussian/normal distribution.

Much of probability theory is concerned with so called concentration or tail bounds, which are upper bounds on the probability that a random variable $X$ deviates too much from its expectation. The first and simplest one of them is Markov’s inequality:

**Theorem 17.7 — Markov’s inequality.** If $X$ is a non-negative random
variable then $\Pr[X \geq k \mathbb{E}[X]] \leq 1/k$.

Markov’s Inequality is actually a very natural statement (see also Fig. 17.6). For example, if you know that the average (not the median!) household income in the US is 70,000 dollars, then in particular you can deduce that at most 25 percent of households make more than 280,000 dollars, since otherwise, even if the remaining 75 percent had zero income, the top 25 percent alone would cause the average income to be larger than 70,000. From this example you can already see that in many situations, Markov’s inequality will not be tight and the probability of deviating from expectation will be much smaller: see the Chebyshev and Chernoff inequalities below.

**Proof of Theorem 17.7.** Let $\mu = \mathbb{E}[X]$ and define $Y = 1_{X \geq k\mu}$. That is, $Y(x) = 1$ if $X(x) \geq k\mu$ and $Y(x) = 0$ otherwise. Note that by definition, for every $x$, $Y(x) \leq X/(k\mu)$. We need to show $\mathbb{E}[Y] \leq 1/k$. But this follows since $\mathbb{E}[Y] \leq \mathbb{E}[X/k(\mu)] = \mathbb{E}[X]/(k\mu) = \mu/(k\mu) = 1/k$.

**Figure 17.6:** Markov’s Inequality tells us that a non-negative random variable $X$ cannot be much larger than its expectation, with high probability. For example, if the expectation of $X$ is $\mu$, then the probability that $X > 4\mu$ must be at most $1/4$, as otherwise just the contribution from this part of the sample space will be too large.

**Going beyond Markov’s Inequality:** Markov’s inequality says that a (non-negative) random variable $X$ can’t go too crazy and be, say, a million times its expectation, with significant probability. But
ideally we would like to say that with high probability, $X$ should be very close to its expectation, e.g., in the range $[0.99\mu, 1.01\mu]$ where $\mu = \mathbb{E}[X]$. This is not generally true, but does turn out to hold when $X$ is obtained by combining (e.g., adding) many independent random variables. This phenomenon, variants of which are known as “law of large numbers”, “central limit theorem”, “invariance principles” and “Chernoff bounds”, is one of the most fundamental in probability and statistics, and is one that we heavily use in computer science as well.

17.3.1 Chebyshev’s Inequality

A standard way to measure the deviation of a random variable from its expectation is by using its standard deviation. For a random variable $X$, we define the variance of $X$ as $\text{Var}[X] = \mathbb{E}[(X - \mu)^2]$ where $\mu = \mathbb{E}[X]$; i.e., the variance is the average squared distance of $X$ from its expectation. The standard deviation of $X$ is defined as $\sigma[X] = \sqrt{\text{Var}[X]}$.

Theorem 17.8 — Chebyshev’s inequality. Suppose that $\mu = \mathbb{E}[X]$ and $\sigma^2 = \text{Var}[X]$. Then for every $k > 0$, $\Pr[|X - \mu| \geq k\sigma] \leq 1/k^2$.

Proof. The proof follows from Markov’s inequality. We define the random variable $Y = (X - \mu)^2$. Then $\mathbb{E}[Y] = \text{Var}[X] = \sigma^2$, and hence by Markov the probability that $Y > k^2\sigma^2$ is at most $1/k^2$. But clearly $(X - \mu)^2 \geq k^2\sigma^2$ if and only if $|X - \mu| \geq k\sigma$. $\blacksquare$

One example of how to use Chebyshev’s inequality is the setting when $X = X_1 + \cdots + X_n$ where $X_i$’s are independent and identically distributed (i.i.d for short) variables with values in $[0, 1]$ where each has expectation 1/2. Since $\mathbb{E}[X] = \sum_i \mathbb{E}[X_i] = n/2$, we would like to say that $X$ is very likely to be in, say, the interval $[0.499n, 0.501n]$. Using Markov’s inequality directly will not help us, since it will only tell us that $X$ is very likely to be at most $100n$ (which we already knew, since it always lies between 0 and $n$). However, since $X_1, \ldots, X_n$ are independent,

$$\text{Var}[X_1 + \cdots + X_n] = \text{Var}[X_1] + \cdots + \text{Var}[X_n].$$

(17.16)

(We leave showing this to the reader as Exercise 17.6.)

For every random variable $X_i$ in $[0, 1]$, $\text{Var}[X_i] \leq 1$ (if the variable is always in $[0, 1]$, it can’t be more than 1 away from its expectation), and hence Eq. (17.16) implies that $\text{Var}[X] \leq n$ and hence $\sigma[X] \leq \sqrt{n}$. 
For large $n$, $\sqrt{n} \ll 0.001n$, and in particular if $\sqrt{n} \leq 0.001n/k$, we can use Chebyshev’s inequality to bound the probability that $X$ is not in $[0.499n, 0.501n]$ by $1/k^2$.

### 17.3.2 The Chernoff bound

Chebyshev’s inequality already shows a connection between independence and concentration, but in many cases we can hope for a quantitatively much stronger result. If, as in the example above, $X = X_1 + \ldots + X_n$, where the $X_i$’s are bounded i.i.d random variables of mean $1/2$, then as $n$ grows, the distribution of $X$ would be roughly the normal or Gaussian distribution— that is, distributed according to the bell curve (see Fig. 17.5 and Fig. 17.7). This distribution has the property of being very concentrated in the sense that the probability of deviating $k$ standard deviations from the mean is not merely $1/k^2$ as is guaranteed by Chebyshev, but rather is roughly $e^{-k^2}$. That is, we have an exponential decay of the probability of deviation.

---

$^6$ Specifically, for a normal random variable $X$ of expectation $\mu$ and standard deviation $\sigma$, the probability that $|X - \mu| \geq k\sigma$ is at most $2e^{-k^2/2}$.

---

**Figure 17.7**: In the normal distribution or the Bell curve, the probability of deviating $k$ standard deviations from the expectation shrinks exponentially in $k^2$, and specifically with probability at least $1 - 2e^{-k^2/2}$, a random variable $X$ of expectation $\mu$ and standard deviation $\sigma$ satisfies $\mu - k\sigma \leq X \leq \mu + k\sigma$. This figure gives more precise bounds for $k = 1, 2, 3, 4, 5, 6$. (Image credit: Imran Baghirov)

The following extremely useful theorem shows that such exponential decay occurs every time we have a sum of independent and bounded variables. This theorem is known under many names in different communities, though it is mostly called the Chernoff bound in the computer science literature:
Theorem 17.9 — Chernoff/Hoeffding bound. If $X_1, \ldots, X_n$ are i.i.d random variables such that $X_i \in [0, 1]$ and $\mathbb{E}[X_i] = p$ for every $i$, then for every $\epsilon > 0$

$$\Pr\left[\left|\sum_{i=0}^{n-1} X_i - np\right| > \epsilon n\right] \leq 2 \cdot e^{-2\epsilon^2 n}.$$

We omit the proof, which appears in many texts, and uses Markov’s inequality on i.i.d random variables $Y_0, \ldots, Y_n$ that are of the form $Y_i = e^{\lambda X_i}$, for some carefully chosen parameter $\lambda$. See Exercise 17.9 for a proof of the simple (but highly useful and representative) case where each $X_i$ is $\{0, 1\}$ valued and $p = 1/2$. (See also Exercise 17.10 for a generalization.)

17.4 LECTURE SUMMARY

- A basic probabilistic experiment corresponds to tossing $n$ coins or choosing $x$ uniformly at random from $\{0, 1\}^n$.

- Random variables assign a real number to every result of a coin toss. The expectation of a random variable $X$ is its average value, and there are several concentration results showing that under certain conditions, random variables deviate significantly from their expectation only with small probability.

17.5 EXERCISES

Exercise 17.1 Suppose that we toss three independent fair coins $a, b, c \in \{0, 1\}$. What is the probability that the XOR of $a, b, c$ is equal to 1? What is the probability that the AND of these three values is equal to 1? Are these two events independent?

Exercise 17.2 Give an example of random variables $X, Y : \{0, 1\}^3 \to \mathbb{R}$ such that $\mathbb{E}[XY] \neq \mathbb{E}[X] \mathbb{E}[Y]$.

Exercise 17.3 Give an example of random variables $X, Y : \{0, 1\}^3 \to \mathbb{R}$ such that $X$ and $Y$ are not independent but $\mathbb{E}[XY] = \mathbb{E}[X] \mathbb{E}[Y]$.

\textit{Disclaimer} Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

\textsuperscript{7} TODO: maybe add an example application of Chernoff. Perhaps a probabilistic method proof using Chernoff+Union bound.
Exercise 17.4 — **Product of expectations.** Prove Lemma 17.5

Exercise 17.5 — **Transformations preserve independence.** Prove Lemma 17.6

Exercise 17.6 — **Variance of independent random variables.** Prove that if $X_0, \ldots, X_{n-1}$ are independent random variables then $\text{Var}[X_0 + \cdots + X_{n-1}] = \sum_{i=0}^{n-1} \text{Var}[X_i]$.

Exercise 17.7 — **Entropy (challenge).** Recall the definition of a distribution $\mu$ over some finite set $S$. Shannon defined the entropy of a distribution $\mu$, denoted by $H(\mu)$, to be $\sum_{x \in S} \mu(x) \log(1/\mu(x))$. The idea is that if $\mu$ is a distribution of entropy $k$, then encoding members of $\mu$ will require $k$ bits, in an amortized sense. In this exercise we justify this definition. Let $\mu$ be such that $H(\mu) = k$.

1. Prove that for every one to one function $F : S \to \{0, 1\}^*$, $E_{x \sim \mu} |F(x)| \geq k$.
2. Prove that for every $\epsilon$, there is some $n$ and a one-to-one function $F : S^n \to \{0, 1\}^*$, such that $E_{x \sim \mu^n} |F(x)| \leq n(k + \epsilon)$, where $x \sim \mu$ denotes the experiments of choosing $x_0, \ldots, x_{n-1}$ each independently from $S$ using the distribution $\mu$.

Exercise 17.8 — **Entropy approximation to binomial.** Let $H(p) = p \log(1/p) + (1 - p) \log(1/(1 - p))$. Prove that for every $p \in (0, 1)$ and $\epsilon > 0$, if $n$ is large enough then

$$2^{|H(p - \epsilon)n|} \left( \begin{array}{c} n \\ pn \end{array} \right) \leq 2^{|H(p + \epsilon)n|}$$

where $\left( \begin{array}{c} n \\ k \end{array} \right)$ is the binomial coefficient $\frac{n!}{k!(n-k)!}$ which is equal to the number of $k$-size subsets of $\{0, \ldots, n-1\}$.

Exercise 17.9 — **Chernoff using Stirling.** 1. Prove that $\Pr_{x \sim \{0, 1\}^n}[\sum x_i = k] = \left( \begin{array}{c} n \\ k \end{array} \right) 2^{-n}$.
2. Use this and Exercise 17.8 to prove the Chernoff bound for the case that $X_0, \ldots, X_n$ are i.i.d. random variables over $\{0, 1\}$ each equaling 0 and 1 with probability 1/2.

Exercise 17.10 — **Poor man’s Chernoff.** Let $X_0, \ldots, X_n$ be i.i.d random variables with $E X_i = p$ and $\Pr[0 \leq X_i \leq 1] = 1$. Define $Y_i = X_i - p$.

1. Prove that for every $j_1, \ldots, j_n \in \mathbb{N}$, if there exists one $i$ such that $j_i$ is odd then $E[\prod_{i=1}^{n-1} Y_i^{j_i}] = 0$.
2. Prove that for every $k$, $E[(\sum_{i=1}^{n-1} Y_i)^k] \leq (10kn)^k/2^{10}$.
3. Prove that for every $\epsilon > 0$, $\Pr[|\sum Y_i| \geq \epsilon n] \geq 2^{-\epsilon^2 n/(10000 \log 1/\epsilon)}$.

Exercise 17.11 — **Simulating distributions using coins.** Our model for probability involves tossing $n$ coins, but sometimes algorithms require sampling from other distributions, such as selecting a uniform

---

*Hint:* While you don’t need this to solve this exercise, this is the function that maps $p$ to the entropy (as defined in Exercise 17.7) of the $p$-biased coin distribution over $\{0, 1\}$, which is the function $\mu : \{0, 1\} \to [0, 1]$ s.t. $\mu(0) = 1 - p$ and $\mu(1) = p$.

*Hint:* Use Stirling’s formula for approximating the factorial function.

*Hint:* Bound the number of tuples $\alpha_0, \ldots, \alpha_{-1}$ such that every $\alpha_i$ is even and $\sum i \alpha_i$.

*Hint:* Set $k = 2[\epsilon^2 n/1000]$ and then show that if the event $|\sum Y_i| \geq \epsilon n$ happens then the random variable $(\sum Y_i)^k$ is a factor of $\epsilon^{-k}$ larger than its expectation.
number in \{0, \ldots, M - 1\} for some \( M \). Fortunately, we can simulate this with an exponentially small probability of error: prove that for every \( M \), if \( n > k \lfloor \log M \rfloor \), then there is a function \( F : \{0,1\}^n \rightarrow \{0, \ldots, M - 1\} \cup \{\bot\} \) such that (1) The probability that \( F(x) = \bot \) is at most \( 2^{-k} \) and (2) the distribution of \( F(x) \) conditioned on \( F(x) \neq \bot \) is equal to the uniform distribution over \( \{0, \ldots, M - 1\} \).\(^{12}\)

**Exercise 17.12 — Sampling.** Suppose that a country has 300,000,000 citizens, 52 percent of which prefer the color “green” and 48 percent of which prefer the color “orange”. Suppose we sample \( n \) random citizens and ask them their favorite color (assume they will answer truthfully). What is the smallest value \( n \) among the following choices so that the probability that the majority of the sample answers “green” is at most 0.05? a. 1,000 b. 10,000 c. 100,000 d. 1,000,000

**Exercise 17.13** Would the answer to **Exercise 17.12** change if the country had 300,000,000,000 citizens?

**Exercise 17.14 — Sampling (2).** Under the same assumptions as **Exercise 17.12**, what is the smallest value \( n \) among the following choices so that the probability that the majority of the sample answers “green” is at most \( 2^{-100} \)? a. 1,000 b. 10,000 c. 100,000 d. 1,000,000 e. It is impossible to get such low probability since there are fewer than \( 2^{100} \) citizens.

\(^{12}\) Hint: Think of \( x \in \{0,1\}^n \) as choosing \( k \) numbers \( y_1, \ldots, y_k \in \{0, \ldots, 2^\lfloor \log M \rfloor - 1\} \). Output the first such number that is in \( \{0, \ldots, M - 1\} \).

13 TODO: add some exercise about the probabilistic method

17.6 BIBLIOGRAPHICAL NOTES

17.7 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

17.8 ACKNOWLEDGEMENTS
18
Probabilistic computation

“in 1946 .. (I asked myself) what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method … might not be to lay it our say one hundred times and simple observe and count”, Stanislaw Ulam, 1983

“The salient features of our method are that it is probabilistic … and with a controllable miniscule probability of error.”, Michael Rabin, 1977

In early computer systems, much effort was taken to drive out randomness and noise. Hardware components were prone to non-deterministic behavior from a number of causes, whether it was vacuum tubes overheating or actual physical bugs causing short circuits (see Fig. 18.1). This motivated John von Neumann, one of the early computing pioneers, to write a paper on how to error correct computation, introducing the notion of redundancy.

So it is quite surprising that randomness turned out not just a hindrance but also a resource for computation, enabling to achieve tasks much more efficiently than previously known. One of the first applications involved the very same John von Neumann. While he was sick in bed and playing cards, Stan Ulam came up with the observation that calculating statistics of a system could be done much faster by running several randomized simulations. He mentioned this idea to von Neumann, who became very excited about it; indeed, it turned out to be crucial for the neutron transport calculations that were needed for development of the Atom bomb and later on the hydrogen bomb. Because this project was highly classified, Ulam, von Neumann and their collaborators came up with the codeword “Monte Carlo”

Learning Objectives:
- See examples of randomized algorithms
- Get more comfort with analyzing probabilistic processes and tail bounds
- Success amplification using tail bounds
for this approach (based on the famous casinos where Ulam’s uncle gambled). The name stuck, and randomized algorithms are known as Monte Carlo algorithms to this day.¹

In this chapter, we will see some examples of randomized algorithms that use randomness to compute a quantity in a faster or simpler way than was known otherwise. We will describe the algorithms in an informal / “pseudo-code” way, rather than as NAND or NAND++ programs. In Chapter 19 we will discuss how to augment the NAND and NAND++ models to incorporate the ability to “toss coins”.

18.1 FINDING APPROXIMATELY GOOD MAXIMUM CUTS.

We start with the following example. Recall the maximum cut problem of finding, given a graph $G = (V, E)$, the cut that maximizes the number of edges. This problem is NP-hard, which means that we do not know of any efficient algorithm that can solve it, but randomization enables a simple algorithm that can cut at least half of the edges:

**Theorem 18.1 — Approximating max cut.** There is an efficient probabilistic algorithm that on input an $n$-vertex $m$-edge graph $G$, outputs a cut $(S, \overline{S})$ that cuts at least $m/2$ of the edges of $G$ in expectation.
Proof Idea: We simply choose a random cut: we choose a subset $S$ of vertices by choosing every vertex $v$ to be a member of $S$ with probability $1/2$ independently. It’s not hard to see that each edge is cut with probability $1/2$ and so the expected number of cut edges is $m/2$. ⋆

Proof of Theorem 18.1. The algorithm is extremely simple:

Algorithm Random Cut:

Input: Graph $G = (V, E)$ with $n$ vertices and $m$ edges. Denote $V = \{v_0, v_1, ..., v_{n-1}\}$.

Operation:

1. Pick $x$ uniformly at random in $\{0, 1\}^n$.
2. Let $S \subseteq V$ be the set $\{v_i : x_i = 1, i \in [n]\}$ that includes all vertices corresponding to coordinates of $x$ where $x_i = 1$.
3. Output the cut $(S, \overline{S})$.

We claim that the expected number of edges cut by the algorithm is $m/2$. Indeed, for every edge $e \in E$, let $X_e$ be the random variable such that $X_e(x) = 1$ if the edge $e$ is cut by $x$, and $X_e(x) = 0$ otherwise. For every such edge $e = \{i, j\}$, $X_e(x) = 1$ if and only if $x_i \neq x_j$. Since the pair $(x_i, x_j)$ obtains each of the values $00, 01, 10, 11$ with probability $1/4$, the probability that $x_i \neq x_j$ is $1/2$ and hence $\mathbb{E}[X_e] = 1/2$. If we let $X$ be the random variable corresponding to the total number of edges cut by $S$, then $X = \sum_{e \in E} X_e$ and hence by linearity of expectation

$$\mathbb{E}[X] = \sum_{e \in E} \mathbb{E}[X_e] = m(1/2) = m/2.$$  \hfill (18.1)

18.1.1 Amplification

Theorem 18.1 gives us an algorithm that cuts $m/2$ edges in expectation. But, as we saw before, expectation does not immediately imply concentration, and so a priori, it may be the case that when we run the algorithm, most of the time we don’t get a cut matching the expectation. Luckily, we can amplify the probability of success by repeating the process several times and outputting the best cut we find. We start by arguing that the probability the algorithm above succeeds in cutting at least $m/2$ edges is not too tiny.

Lemma 18.2 The probability that a random cut in an $m$ edge graph cuts at least $m/2$ edges is at least $1/(2m)$.

Proof Idea: To see the idea behind the proof, think of the case that $m = 1000$. In this case one can show that we will cut at least 500
edges with probability at least 0.001 (and so in particular larger than \(1/(2m) = 1/2000\)). Specifically, if we assume otherwise, then this means that with probability more than 0.999 the algorithm cuts 499 or fewer edges. But since we can never cut more than the total of 1000 edges, given this assumption, the highest value the expected number of edges cut is if we cut exactly 499 edges with probability 0.999 and cut 1000 edges with probability 0.001. Yet even in this case the expected number of edges will be 0.999 \(\cdot 499 + 0.001 \cdot 1000 = 499.99 < 500\), which contradicts the fact that we’ve calculated the expectation to be at least 500 in Theorem 18.1. 

\[ \star \]

**Proof of Lemma 18.2.** Let \(p\) be the probability that we cut at least \(m/2\) edges and suppose, towards a contradiction, that \(p < 1/(2m)\). Since the number of edges cut is an integer, and \(m/2\) is a multiple of 0.5, by definition of \(p\), with probability \(1 - p\) we cut at most \(m/2 - 0.5\) edges. Moreover, since we can never cut more than \(m\) edges, under our assumption that \(p < m/2\), we can bound the expected number of edges cut by

\[
\Pr \left[ \text{number of edges cut by } (S, \overline{S}) \geq \frac{|E|}{2} \right] \geq 1 - 2^{-1000} .
\]

Proof. The algorithm will work as follows:
Algorithm AMPLIFY RANDOM CUT:

**Input:** Graph $G = (V, E)$ with $n$ vertices and $m$ edges. Denote $V = \{v_0, v_1, ..., v_{n-1}\}$. Number $k > 0$.

**Operation:**

1. Repeat the following $200km$ times:
   
   (a) Pick $x$ uniformly at random in $\{0, 1\}^n$.
   
   (b) Let $S \subseteq V$ be the set $\{v_i : x_i = 1 , i \in [n]\}$ that includes all vertices corresponding to coordinates of $x$ where $x_i = 1$.
   
   (c) If $(S, \overline{S})$ cuts at least $m/2$ then halt and output $(S, \overline{S})$.

2. Output “failed”

We leave completing the analysis as an exercise to the reader (see Exercise 18.1).

18.1.2 Two-sided amplification

The analysis above relied on the fact that the maximum cut has one sided error. By this we mean that if we get a cut of size at least $m/2$ then we know we have succeeded. This is common for randomized algorithms, but is not the only case. In particular, consider the task of computing some Boolean function $F : \{0, 1\}^* \rightarrow \{0, 1\}$. A randomized algorithm $A$ for computing $F$, given input $x$, might toss coins and succeed in outputting $F(x)$ with probability, say, 0.9. We say that $A$ has two sided errors if there is positive probability that $A(x)$ outputs 1 when $F(x) = 0$, and positive probability that $A(x)$ outputs 0 when $F(x) = 1$.

In such a case, to simplify $A$’s success, we cannot simply repeat it $k$ times and output 1 if a single one of those repetitions resulted in 1, nor can we output 0 if a single one of the repetitions resulted in 0. But we can output the majority value of these repetitions. By the Chernoff bound (Theorem 17.9), with probability exponentially close to 1 (i.e., $1 - 2^{\Omega(k)}$), the fraction of the repetitions where $A$ will output $F(x)$ will be at least, say 0.89, and in such cases we will of course output the correct answer.

The above translates into the following theorem.

**Theorem 18.4 — Two-sided amplification.** If $F : \{0, 1\}^* \rightarrow \{0, 1\}$ is a function such that there is a polynomial-time algorithm $A$ satisfying

$$\Pr[A(x) = F(x)] \geq 0.51 \quad (18.5)$$

for every $x \in \{0, 1\}^*$, then there is a polynomial time algorithm $B$
This question does have some significance to practice, since hardware that generates high quality randomness at speed is nontrivial to construct.

Theorem 18.4

We omit the proof of Theorem 18.4, since we will prove a more general result later on in Theorem 19.3.

18.1.3 What does this mean?

We have shown a probabilistic algorithm that on any $m$ edge graph $G$, will output a cut of at least $m/2$ edges with probability at least $1 - 2^{-1000}$. Does it mean that we can consider this problem as “easy”? Should we be somewhat wary of using a probabilistic algorithm, since it can sometimes fail?

First of all, it is important to emphasize that this is still a worst case guarantee. That is, we are not assuming anything about the input graph: the probability is only due to the internal randomness of the algorithm. While a probabilistic algorithm might not seem as nice as a deterministic algorithm that is guaranteed to give an output, to get a sense of what a failure probability of $2^{-1000}$ means, note that:

- The chance of winning the Massachusetts Mega Million lottery is one over $(75)^5 \cdot 15$ which is roughly $2^{-35}$. So $2^{-1000}$ corresponds to winning the lottery about 300 times in a row, at which point you might not care so much about your algorithm failing.

- The chance for a U.S. resident to be struck by lightning is about $1/700000$, which corresponds to about $2^{-45}$ chance that you’ll be struck by lightning the very second that you’re reading this sentence (after which again you might not care so much about the algorithm’s performance).

- Since the earth is about 5 billion years old, we can estimate the chance that an asteroid of the magnitude that caused the dinosaurs’ extinction will hit us this very second to be about $2^{-60}$. It is quite likely that even a deterministic algorithm will fail if this happens.

So, in practical terms, a probabilistic algorithm is just as good as a deterministic one. But it is still a theoretically fascinating question whether randomized algorithms actually yield more power, or whether is it the case that for any computational problem that can be solved by probabilistic algorithm, there is a deterministic algorithm with nearly the same performance.\(^2\) For example, we will see in Exercise 18.2 that there is in fact a deterministic algorithm that can cut at least $m/2$ edges in an $m$-edge graph. We will discuss this question in generality in Chapter 19. For now, let us see a couple of examples where randomization leads to algorithms that are better in some sense than the known deterministic algorithms.

\(^2\) This question does have some significance to practice, since hardware that generates high quality randomness at speed is nontrivial to construct.
18.1.4 Solving SAT through randomization

The 3SAT problem is NP hard, and so it is unlikely that it has a polynomial (or even subexponential) time algorithm. But this does not mean that we can’t do at least somewhat better than the trivial $2^n$ algorithm for $n$-variable 3SAT. The best known worst-case algorithms for 3SAT are randomized, and are related to the following simple algorithm, variants of which are also used in practice:

Algorithm WalkSAT:

Input: An $n$ variable 3CNF formula $\varphi$.

Parameters: $T, S \in \mathbb{N}$

Operation:

1. Repeat the following $T$ steps:
   (a) Choose a random assignment $x \in \{0,1\}^n$ and repeat the following for $S$ steps:
      i. If $x$ satisfies $\varphi$ then output $x$.
      ii. Otherwise, choose a random clause $(\ell_i \lor \ell_j \lor \ell_k)$ that $x$ does not satisfy, choose a random literal in $\ell_i, \ell_j, \ell_k$ and modify $x$ to satisfy this literal.

2. If all the $T \cdot S$ repetitions above did not result in a satisfying assignment then output Unsatisfiable.

The running time of this algorithm is $S \cdot T \cdot \text{poly}(n)$, and so the key question is how small we can make $S$ and $T$ so that the probability that WalkSAT outputs Unsatisfiable on a satisfiable formula $\varphi$ is small. It is known that we can do so with $ST = \tilde{O}((4/3)^n) = \tilde{O}(1.333...)^n$ (see Exercise 18.4 for a weaker result), but we’ll show below a simpler analysis yielding $ST = \tilde{O}(\sqrt{3}^n) = \tilde{O}(1.74^n)$, which is still much better than the trivial $2^n$ bound.

### Theorem 18.5 — WalkSAT simple analysis.

If we set $T = 100 \cdot \sqrt{3}^n$ and $S = n/2$, then the probability we output Unsatisfiable for a satisfiable $\varphi$ is at most $1/2$.

**Proof.** Suppose that $\varphi$ is a satisfiable formula and let $x^*$ be a satisfying assignment for it. For every $x \in \{0,1\}^n$, denote by $\Delta(x, x^*)$ the number of coordinates that differ between $x$ and $x^*$. The heart of the proof is the following claim:

**Claim I:** For every $x, x^*$ as above, in every local improvement step, the value $\Delta(x, x^*)$ is decreased by one with probability at least $1/3$.

**Proof of Claim I:** Since $x^*$ is a satisfying assignment, if $C$ is a clause that $x$ does not satisfy, then at least one of the variables involve in $C$
must get different values in \( x \) and \( x^* \). Thus when we change \( x \) by one of the three literals in the clause, we have probability at least \( 1/3 \) of decreasing the distance.

The second claim is that our starting point is not that bad:

**Claim 2:** With probability at least \( 1/2 \) over a random \( x \in \{0,1\}^n \), \( \Delta(x,x^*) \leq n/2 \).

**Proof of Claim II:** Consider the map \( FLIP : \{0,1\}^n \to \{0,1\}^n \) that simply “flips” all the bits of its input from 0 to 1 and vice versa. That is, \( FLIP(x_0,\ldots,x_{n-1}) = (1-x_0,\ldots,1-x_{n-1}) \). Clearly \( FLIP \) is one to one. Moreover, if \( x \) is of distance \( k \) to \( x^* \), then \( FLIP(x) \) is distance \( n-k \) to \( x^* \). Now let \( B \) be the “bad event” in which \( x \) is of distance \( \geq n/2 \) from \( x^* \). Then the set \( A = FLIP(B) = \{ FLIP(x) : x \in \{0,1\}^n \} \) satisfies \( |A| = |B| \) and that if \( x \in A \) then \( x \) is of distance \( < n/2 \) from \( x^* \). Since \( A \) and \( B \) are disjoint events, \( \Pr[A] + \Pr[B] \leq 1 \). Since they have the same cardinality, they have the same probability and so we get that \( 2 \Pr[B] \leq 1 \) or \( \Pr[B] \leq 1/2 \). (See also Fig. 18.2).

Claims I and II imply that each of the \( T \) iterations of the outer loop succeeds with probability at least \( 0.5 \cdot \sqrt{3}^{-n/2} \). Indeed, by Claim II, the original guess \( x \) will satisfy \( \Delta(x,x^*) \leq n/2 \), and by Claim I, even conditioned on all the history so far, for each of the \( S = n/2 \) steps we have probability \( \geq 1/3 \) of being “lucky” and decreasing the distance at one. The chance we will be lucky in all \( n/2 \) steps is hence at least \( (1/3)^{n/2} = \sqrt{3}^{-n/2} \).

Since any single iteration of the outer loop succeeds with probability at least \( \frac{1}{2} \cdot \sqrt{3}^{-n} \), the probability that we never do so in \( T = 100\sqrt{3}^n \) repetitions is at most \( (1 - \frac{1}{2\sqrt{3}})^{100\sqrt{3}^n} \leq (1/e)^{50} \).

### 18.1.5 Bipartite matching.

The **matching** problem is one of the canonical optimization problems, arising in all kinds of applications: matching residents with hospitals, kidney donors with patients, flights with crews, and many others. One prototypical variant is **bipartite perfect matching**. In this problem, we are given a bipartite graph \( G = (L \cup R, E) \) which has 2\( n \) vertices partitioned into \( n \)-sized sets \( L \) and \( R \), where all edges have one endpoint in \( L \) and the other in \( R \). The goal is to determine whether there is a **perfect matching**, a subset \( M \subseteq E \) of \( n \) disjoint edges. That is, \( M \) matches every vertex in \( L \) to a unique vertex in \( R \).

The bipartite matching problem turns out to have a polynomial-time algorithm, since we can reduce finding a matching in \( G \) to finding a maximum flow (or equivalently, minimum cut) in a related graph \( G' \) (see Fig. 18.3). However, we will see a different probabilistic algorithm to determine whether a graph contains such a matching.

Let us label \( G \)'s vertices as \( L = \{\ell_0,\ldots,\ell_{n-1}\} \) and \( R = \{r_0,\ldots,r_{n-1}\} \).
Figure 18.2: For every $x^* \in \{0,1\}^n$, we can sort all strings in $\{0,1\}^n$ according to their distance from $x^*$ (top to bottom in the above figure), where we let $A = \{x \in \{0,1\}^n \mid \text{dist}(x,x^* \leq n/2)\}$ be the "top half" of strings. If we define $FLIP : \{0,1\}^n \rightarrow \{0,1\}$ to be the map that "flips" the bits of a given string $x$ then it maps every $x \in A$ to an output $FLIP(x) \in A$ in a one-to-one way, and so it demonstrates that $|A| \leq |A|$ which implies that $\Pr[A] \geq \Pr[A]$ and hence $\Pr[A] \geq 1/2$.

Figure 18.3: The bipartite matching problem in the graph $G = (L \cup R, E)$ can be reduced to the minimum $s, t \text{ cut problem}$ in the graph $G'$ obtained by adding vertices $s, t$ to $G$, connecting $s$ with $L$ and connecting $t$ with $R$. 
A matching $M$ corresponds to a permutation $\pi \in S_n$ (i.e., one-to-one and onto function $\pi: [n] \to [n]$) where for every $i \in [n]$, we define $\pi(i)$ to be the unique $j$ such that $M$ contains the edge $\{\ell_i, r_j\}$. Define an $n \times n$ matrix $A = A(G)$ where $A_{i,j} = 1$ if and only if the edge $\{\ell_i, r_j\}$ is present and $A_{i,j} = 0$ otherwise. The correspondence between matchings and permutations implies the following claim:

**Lemma 18.6 — Matching polynomial.** Define $P = P(G)$ to be the polynomial mapping $\mathbb{R}^{n^2}$ to $\mathbb{R}$ where

$$P(x_{0,0}, \ldots, x_{n-1,n-1}) = \sum_{\pi \in S_n} \left( \prod_{i=0}^{n-1} \text{sign}(\pi) A_{i,\pi(i)} \right) \prod_{i=0}^{n-1} x_{i,\pi(i)} \quad (18.7)$$

Then $G$ has a perfect matching if and only if $P$ is not identically zero. That is, $G$ has a perfect matching if and only if there exists some assignment $x = (x_{i,j})_{i,j\in[n]} \in \mathbb{R}^{n^2}$ such that $P(x) \neq 0.4$

**Proof.** If $G$ has a perfect matching $M^*$, then let $\pi^*$ be the permutation corresponding to $M$ and let $x^* \in \mathbb{Z}^{n^2}$ defined as follows: $x_{i,j} = 1$ if $j = \pi(i)$ and $x_{i,j} = 0$. Note that for every $\pi \neq \pi^*$, $\prod_{i=0}^{n-1} x_{i,\pi(i)} = 0$ but $\prod_{i=0}^{n-1} x_{i,\pi^*(i)} = 1$. Hence $P(x^*)$ will equal $\prod_{i=0}^{n-1} A_{i,\pi^*(i)}$. But since $M^*$ is a perfect matching in $G$, $\prod_{i=0}^{n-1} A_{i,\pi^*(i)} = 1$.

On the other hand, suppose that $P$ is not identically zero. By Eq. (18.7), this means that at least one of the terms $\prod_{i=0}^{n-1} A_{i,\pi(i)}$ is not equal to zero. But then this permutation $\pi$ must be a perfect matching in $G$. ■

As we’ve seen before, for every $x \in \mathbb{R}^{n^2}$, we can compute $P(x)$ by simply computing the determinant of the matrix $A(x)$, which is obtained by replacing $A_{i,j}$ with $A_{i,j}x_{i,j}$. This reduces testing perfect matching to the zero testing problem for polynomials: given some polynomial $P(\cdot)$, test whether $P$ is identically zero or not. The intuition behind our randomized algorithm for zero testing is the following:

*If a polynomial is not identically zero, then it can’t have “too many” roots.*

This intuition sort of makes sense. For one variable polynomials, we know that a nonzero linear function has at most one root, a quadratic function (e.g., a parabola) has at most two roots, and generally a degree $d$ equation has at most $d$ roots. While in more than one variable there can be an infinite number of roots (e.g., the polynomial $x_0 + x_0$ vanishes on the line $y = -x$) it is still the case that the set of roots is very “small” compared to the set of all inputs. For example,
the root of a bivariate polynomial form a curve, the roots of a three-variable polynomial form a surface, and more generally the roots of an $n$-variable polynomial are a space of dimension $n - 1$.

This intuition leads to the following simple randomized algorithm:

To decide if $P$ is identically zero, choose a “random” input $x$ and check if $P(x) \neq 0$.

This makes sense: if there are only “few” roots, then we expect that with high probability the random input $x$ is not going to be one of those roots. However, to transform this into an actual algorithm, we need to make both the intuition and the notion of a “random” input precise. Choosing a random real number is quite problematic, especially when you have only a finite number of coins at your disposal, and so we start by reducing the task to a finite setting. We will use the following result:

**Theorem 18.7 — Schwartz–Zippel lemma.** For every integer $q$, and polynomial $P : \mathbb{R}^n \to \mathbb{R}$ with integer coefficients. If $P$ has degree at most $d$ and is not identically zero, then it has at most $dq^{n-1}$ roots in the set $[q]^n = \{(x_0, \ldots, x_{n-1}) : x_i \in \{0, \ldots, q - 1\}\}$.

We omit the (not too complicated) proof of Theorem 18.7. We remark that it holds not just over the real numbers but over any field as well. Since the matching polynomial $P$ of Lemma 18.6 has degree at most $n$, Theorem 18.7 leads directly to a simple algorithm for testing if
it is nonzero:

Algorithm Perfect-Matching:

Input: Bipartite graph $G$ on $2n$ vertices

$\{\ell_0, \ldots, \ell_{n-1}, r_0, \ldots, r_{n-1}\}$.

Operation:

1. For every $i, j \in [n]$, choose $x_{i,j}$ independently at random from $[2n] = \{0, \ldots, 2n - 1\}$.
2. Compute the determinant of the matrix $A(\mathbf{x})$ whose $(i, j)$th entry equals $x_{i,j}$ if the edge $\{\ell_i, r_j\}$ is present and 0 otherwise.
3. Output no perfect matching if this determinant is zero, and output perfect matching otherwise.

This algorithm can be improved further (e.g., see Exercise 18.5). While it is not necessarily faster than the cut-based algorithms for perfect matching, it does have some advantages. In particular, it is more amenable for parallelization. (However, it also has the significant disadvantage that it does not produce a matching but only states that one exists.) The Schwartz–Zippel Lemma, and the associated zero testing algorithm for polynomials, is widely used across computer science, including in several settings where we have no known deterministic algorithm matching their performance.

Lecture Recap

- Using concentration results, we can amplify in polynomial time the success probability of a probabilistic algorithm from a mere $1/p(n)$ to $1 - 2^{-\alpha(n)}$ for every polynomials $p$ and $q$.
- There are several randomized algorithms that are better in various senses (e.g., simpler, faster, or other advantages) than the best known deterministic algorithm for the same problem.

18.2 EXERCISES
Exercise 18.1 — Amplification for max cut. Prove Lemma 18.3.

Exercise 18.2 — Deterministic max cut algorithm.  

Exercise 18.3 — Simulating distributions using coins. Our model for probability involves tossing $n$ coins, but sometimes algorithm requires sampling from other distributions, such as selecting a uniform number in $\{0, \ldots, M - 1\}$ for some $M$. Fortunately, we can simulate this with an exponentially small probability of error: prove that for every $M$, if $n > k \log M$, then there is a function $F : \{0, 1\}^n \to \{0, \ldots, M - 1\} \cup \{\perp\}$ such that (1) The probability that $F(x) = \perp$ is at most $2^{-k}$ and (2) the distribution of $F(x)$ conditioned on $F(x) \neq \perp$ is equal to the uniform distribution over $\{0, \ldots, M - 1\}$.  

Exercise 18.4 — Better walksat analysis.  

1. Prove that for every $\epsilon > 0$, if $n$ is large enough then for every $x^* \in \{0, 1\}^n$ $\Pr_{x \sim \{0, 1\}^n}[\Delta(x, x^*) \leq n/3] \leq 2^{-(1-H(1/3)-\epsilon)n}$ where $H(p) = p \log(1/p) + (1 - p) \log(1/(1 - p))$ is the same function as in Exercise 17.8.  

2. Prove that $2^{1-H(1/4)+(1/4) \log 3} = (3/2)$.  

3. Use the above to prove that for every $\delta > 0$ and large enough $n$, if we set $T = 1000 \cdot (3/2 + \delta)^n$ and $S = n/4$ in the WalkSAT algorithm then for every satisfiable 3CNF $\varphi$, the probability that we output unsatisfiable is at most $1/2$.

Exercise 18.5 — Faster bipartite matching (challenge).

$\text{monte carlo history: } \url{http://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-88-9068}$

18.4 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

18.5 ACKNOWLEDGEMENTS
19
Modeling randomized computation

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin." John von Neumann, 1951.

So far we have described randomized algorithms in an informal way, assuming that an operation such as “pick a string $x \in \{0,1\}^n$” can be done efficiently. We have neglected to address two questions:

1. How do we actually efficiently obtain random strings in the physical world?

2. What is the mathematical model for randomized computations, and is it more powerful than deterministic computation?

The first question is of both practical and theoretical importance, but for now let’s just say that there are various physical sources of “random” or “unpredictable” data. A user’s mouse movements and typing pattern, (non solid state) hard drive and network latency, thermal noise, and radioactive decay have all been used as sources for randomness. For example, many Intel chips come with a random number generator built in. One can even build mechanical coin tossing machines (see Fig. 19.1).¹

In this chapter we focus on the second question: formally modeling probabilistic computation and studying its power. Modeling randomized computation is actually quite easy. We can add the following operations to our NAND, NAND++ and NAND« programming languages:

foo = RAND()

where foo is a variable. The result of applying this operation is that foo is assigned a random bit in $\{0, 1\}$. (Every time the RAND operation is invoked it returns a fresh independent random bit.) We

¹ The output of processes such as above can be thought of as a binary string sampled from some distribution $\mu$ that might have significant unpredictability (or entropy) but is not necessarily the uniform distribution over $\{0, 1\}^n$. Indeed, as this paper shows, even (real-world) coin tosses do not have exactly the distribution of a uniformly random string. Therefore, to use the resulting measurements for randomized algorithms, one typically needs to apply a “distillation” or randomness extraction process to the raw measurements to transform them to the uniform distribution.
call the resulting languages RNAND, RNAND++, and RNAND« respectively.

We can use this to define the notion of a function being computed by a randomized $T(n)$ time algorithm for every nice time bound $T : \mathbb{N} \to \mathbb{N}$, as well as the notion of a finite function being computed by a size $S$ randomized NAND program (or, equivalently, a randomized circuit with $S$ gates that correspond to either the NAND or coin-tossing operations). However, for simplicity we will not define randomized computation in full generality, but simply focus on the class of functions that are computable by randomized algorithms running in polynomial time, which by historical convention is known as BPP:

**Definition 19.1 — BPP.** Let $F : \{0, 1\}^* \to \{0, 1\}$. We say that $F \in \text{BPP}$ if there exist constants $a, b \in \mathbb{N}$ and an RNAND« program $P$ such that for every $x \in \{0, 1\}^*$, on input $x$, the program $P$ halts within at most $a|x|^b$ steps and

$$\Pr[P(x) = F(x)] \geq \frac{2}{3}$$

(19.1)

where this probability is taken over the result of the RAND operations of $P$.

The same polynomial-overhead simulation of NAND« programs by NAND++ programs we saw in Theorem 12.5 extends to randomized programs as well. Hence the class BPP is the same regardless of whether it is defined via RNAND++ or RNAND« programs.
19.0.1 An alternative view: random coins as an “extra input”

While we presented randomized computation as adding an extra “coin tossing” operation to our programs, we can also model this as being given an additional extra input. That is, we can think of a randomized algorithm $A$ as a deterministic algorithm $A'$ that takes two inputs $x$ and $r$ where the second input $r$ is chosen at random from $\{0, 1\}^m$ for some $m \in \mathbb{N}$ (see Fig. 19.2). The equivalence to the Definition 19.1 is shown in the following theorem:

![Figure 19.2](image)

**Theorem 19.2 — Alternative characterization of BPP.** Let $F : \{0, 1\}^* \to \{0, 1\}$. Then $F \in \text{BPP}$ if and only if there exists $a, b \in \mathbb{N}$ and $G : \{0, 1\}^* \to \{0, 1\}$ such that $G$ is in $\text{P}$ and for every $x \in \{0, 1\}^*$,

$$
\Pr_{r \sim \{0, 1\}^a|x:b} [G(xr) = F(x)] \geq \frac{2}{3}.
$$

(19.2)

**Proof Idea:** The idea behind the proof is that, as illustrated in Fig. 19.2, we can simply replace sampling a random coin with reading a bit from the extra “random input” $r$ and vice versa. To prove this rigorously we need to work through some slightly cumbersome formal notation. This might be one of those proofs that is easier to work out on your own than to read. ⋆

**Proof of Theorem 19.2.** We start by showing the “only if” direction. Let $F \in \text{BPP}$ and let $P$ be an RNAND$\omega$ program that computes $F$ as per Definition 19.1, and let $a, b \in \mathbb{N}$ be such that on every input of length $n$, the program $P$ halts within at most $an^b$ steps. We will construct a polynomial-time NAND$\omega$ program $P'$ that computes a function $G$ satisfying the conditions of Eq. (19.2).
The program $P'$ is very simple:

Program $P'$: (Deterministic NAND$n$ program)

Inputs:
- $x \in \{0,1\}^n$
- $r \in \{0,1\}^{an^b}$

Goal: Output $y$ that has the same distribution as the output of the RNAND$n$ program $P$ on input $x$.

Operation:
1. Copy the string $r$ to an array $\text{Coins}$. That is $\text{Coins}[i] = r_i$ for all $i \in [an^b]$.
2. Let $\text{coincounter}$ be a variable and set it to 0.
3. Simulate an execution the RNAND$n$ program $P$, replacing any line of the form $\text{foo} = \text{RAND}()$ with the two lines: $\text{foo} = \text{Coins}[\text{coincounter}]$ and $\text{coincounter} = \text{coincounter} + 1$.

The program $P'$ is a deterministic polynomial time NAND$n$ program, and so computes some function $G \in \mathbb{P}$. By its construction, the distribution of $P'(xr)$ for random $r \in \{0,1\}^{2an^b}$ is identical to the distribution of $P(x)$ (where in the latter case the sample space is the results of the RAND() calls), and hence in particular it will hold that $\Pr_{r \in \{0,1\}^{an^b}}[P'(xr) = F(x)] \geq 2/3$.

For the other direction, given a function $G \in \mathbb{P}$ satisfying the condition Eq. (19.2) and a NAND$n$ program $P'$ that computes $G$ in polynomial time, we can construct an RNAND$n$ program $P$ that computes $F$ in polynomial time. On input $x \in \{0,1\}^n$, the program $P$ will simply use the RNAND() instruction $an^b$ times to fill an array $R[0], ..., R[an^b-1]$ and then execute the original program $P'$ on input $xr$ where $r_i$ is the $i$-th element of the array $R$. Once again, it is clear that if $P'$ runs in polynomial time then so will $P$, and for every input $x$ and $r \in \{0,1\}^{an^b}$, the output of $P$ on input $x$ and where the coin tosses outcome is $r$ is equal to $P'(xr)$.

Definitions of BPP and NP

The characterization of BPP Theorem 19.2 is reminiscent of the characterization of NP in Definition 14.1, with the randomness in the case of BPP playing the role of the solution in the case of NP. However, there are important differences between the two:

- The definition of NP is “one sided”: $F(x) = 1$ if there exists a solution $w$ such that $G(xw) = 1$ and
Modeling Randomized Computation

The relation between \( \text{NP} \) and \( \text{BPP} \) is not immediately clear. It is not known whether \( \text{BPP} \subseteq \text{NP} \), \( \text{NP} \subseteq \text{BPP} \), or these two classes are incomparable. It is however known (with a non-trivial proof) that if \( \text{P} = \text{NP} \) then \( \text{BPP} = \text{P} \) (see Theorem 19.9).

Most importantly, the definition of \( \text{NP} \) is “ineffective,” since it does not yield a way of actually finding whether there exists a solution among the exponentially many possibilities. By contrast, the definition of \( \text{BPP} \) gives us a way to compute the function in practice by simply choosing the second input at random.

“Random tapes” Theorem 19.2 motivates sometimes considering the randomness of an RNAND++ (or RNAND\(\bowtie\)) program as an extra input. As such, if \( A \) is a randomized algorithm that on inputs of length \( n \) makes at most \( p(n) \) coin tosses, we will often use the notation \( A(x; r) \) (where \( x \in \{0,1\}^n \) and \( r \in \{0,1\}^{p(n)} \)) to refer to the result of executing \( x \) when the coin tosses of \( A \) correspond to the coordinates of \( r \). This second, or “auxiliary,” input is sometimes referred to as a “random tape.” This terminology originates from the model of randomized Turing machines.

19.0.2 Amplification

The number \( 2/3 \) might seem arbitrary, but as we’ve seen in Chapter 18 it can be amplified to our liking:

**Theorem 19.3 — Amplification.** Let \( P \) be an RNAND\(\bowtie\) program, \( F : \{0,1\}^* \rightarrow \{0,1\} \), and \( T : \mathbb{N} \rightarrow \mathbb{N} \) be a nice time bound such that for every \( x \in \{0,1\}^* \), on input \( x \) the program \( P \) runs in at most \( T(|x|) \) steps and moreover \( \Pr[P(x) = F(x)] \geq \frac{1}{2} + \epsilon \) for some \( \epsilon > 0 \). Then for every \( k \), there is a program \( P' \) taking at most \( O(k \cdot T(n)/\epsilon^2) \) steps such that on input \( x \in \{0,1\}^* \), \( \Pr[P'(x) = F(x)] > 1 - 2^{-k} \).

**Proof Idea:** The proof is the same as we’ve seen before in the maximum cut and other examples. We use the Chernoff bound to argue that if we run the program \( O(k/\epsilon^2) \) times, each time using fresh and independent random coins, then the probability that the majority of the answers will not be correct will be less than \( 2^{-k} \). Amplification can be thought of as a “polling” of the choices for randomness for the algorithm (see Fig. 19.3).
**Proof of Theorem 19.3.** We can run $P$ on input $x$ for $t = 10k/\epsilon^2$ times, using fresh randomness in each execution, and compute the outputs $y_0, \ldots, y_{t-1}$. We output the value $y$ that appeared the largest number of times. Let $X_i$ be the random variable that is equal to 1 if $y_i = F(x)$ and equal to 0 otherwise. The random variables $X_0, \ldots, X_{t-1}$ are i.i.d. and satisfy $\mathbb{E}[X_i] = \Pr[X_i = 1] \geq 1/2 + \epsilon$, and hence by linearity of expectation $\mathbb{E}[(\sum_{i=0}^{t-1} X_i)] \geq t(1/2 + \epsilon)$. For the plurality value to be incorrect, it must hold that $\sum_{i=0}^{t-1} X_i \leq t/2$, which means that $\sum_{i=0}^{t-1} X_i$ is at least $\epsilon t$ far from its expectation. Hence by the Chernoff bound (Theorem 17.9), the probability that the plurality value is not correct is at most $2^{\Omega(\log(1/\delta)/\epsilon^2)}$, which is smaller than $2^{-\delta}$ for our choice of $t$. ■

![Figure 19.3](image)

**Figure 19.3:** If $F \in \text{BPP}$ then there is randomized polynomial-time algorithm $P$ with the following property: In the case $F(x) = 0$ two thirds of the “population” of random choices satisfy $P(x; r) = 0$ and in the case $F(x) = 1$ two thirds of the population satisfy $P(x; r) = 1$. We can think of amplification as a form of “polling” of the choices of randomness. By the Chernoff bound, if we poll a sample of $O(\log(1/\delta)/\epsilon^2)$ random choices $r$, then with probability at least $1 - \delta$, the fraction of $r$’s in the sample satisfying $P(x; r) = 1$ will give us an estimate of the fraction of the population within an $\epsilon$ margin of error. This is the same calculation used by pollsters to determine the needed sample size in their polls.

There is nothing special about NAND$\neg$ in Theorem 19.3. The same proof can be used to amplify randomized NAND or NAND++ programs as well.

## 19.1 BPP AND NP COMPLETENESS

Since “noisy processes” abound in nature, randomized algorithms can be realized physically, and so it is reasonable to propose BPP rather than $P$ as our mathematical model for “feasible” or “tractable” computation. One might wonder if this makes all the previous chapters irrelevant, and in particular if the theory of NP completeness still applies to probabilistic algorithms. Fortunately, the answer is Yes:
Before seeing the proof, note that Theorem 19.4 implies that if there was a randomized polynomial time algorithm for any $\mathbf{NP}$-complete problem such as $3\mathbf{SAT}$, $\mathbf{ISET}$ etc., then there would be such an algorithm for every problem in $\mathbf{NP}$. Thus, regardless of whether our model of computation is deterministic or randomized algorithms, $\mathbf{NP}$-complete problems retain their status as the “hardest problems in $\mathbf{NP}$.”

**Proof Idea:** The idea is to simply run the reduction as usual, and plug it into the randomized algorithm instead of a deterministic one. It would be an excellent exercise, and a way to reinforce the definitions of $\mathbf{NP}$-hardness and randomized algorithms, for you to work out the proof for yourself. However for the sake of completeness, we include this proof below.

**Proof of Theorem 19.4.** Suppose that $F$ is $\mathbf{NP}$-hard and $F \in \mathbf{BPP}$. We will now show that this implies that $\mathbf{NP} \subseteq \mathbf{BPP}$. Let $G \in \mathbf{NP}$. By the definition of $\mathbf{NP}$-hardness, it follows that $G \leq_p F$, or that in other words there exists a polynomial-time computable function $R : \{0,1\}^* \rightarrow \{0,1\}^*$ such that $G(x) = F(R(x))$ for every $x \in \{0,1\}^*$. Now if $F$ is in $\mathbf{BPP}$ then there is a polynomial-time $\mathsf{R\!N\!A\!N\!D}^+$ program $P$ such that

$$\Pr[P(y) = F(y)] \geq 2/3$$

for every $y \in \{0,1\}^*$ (where the probability is taken over the random coin tosses of $\mathcal{P}$). Hence we can get a polynomial-time $\mathsf{R\!N\!A\!N\!D}^+$ program $P'$ to compute $G$ by setting $P'(x) = P(R(x))$. By Eq. (19.3) $\Pr[P'(x) = F(R(x))] \geq 2/3$ and since $F(R(x)) = G(x)$ this implies that $\Pr[P'(x) = G(x)] \geq 2/3$, which proves that $G \in \mathbf{BPP}$. \[\square\]

Most of the results we’ve seen about $\mathbf{NP}$ hardness, including the search to decision reduction of Theorem 15.1, the decision to optimization reduction of Theorem 15.2, and the quantifier elimination result of Theorem 15.4, all carry over in the same way if we replace $\mathbf{P}$ with $\mathbf{BPP}$ as our model of efficient computation. Thus if $\mathbf{NP} \subseteq \mathbf{BPP}$ then we get essentially all of the strange and wonderful consequences of $\mathbf{P} = \mathbf{NP}$. Unsurprisingly, we cannot rule out this possibility. In fact, unlike $\mathbf{P} = \mathbf{EXP}$, which is ruled out by the time hierarchy theorem, we don’t even know how to rule out the possibility that $\mathbf{BPP} = \mathbf{EXP}$! Thus a priori it’s possible (though seems highly unlikely) that randomness is a magical tool that allows us to speed up arbitrary exponential time computation.\(^3\) Nevertheless, as we discuss below, it is believed that randomization’s power is much weaker and $\mathbf{BPP}$ lies in much more “pedestrian” territory.

\(^3\) At the time of this writing, the largest “natural” complexity class which we can’t rule out being contained in $\mathbf{BPP}$ is the class $\mathbf{NEXP}$, which we did not define in this course, but corresponds to non deterministic exponential time. See this paper for a discussion of this question.
19.2 THE POWER OF RANDOMIZATION

A major question is whether randomization can add power to computation. Mathematically, we can phrase this as the following question: does $\text{BPP} = \text{P}$? Given what we’ve seen so far about the relations of other complexity classes such as $\text{P}$ and $\text{NP}$, or $\text{NP}$ and $\text{EXP}$, one might guess that:

1. We do not know the answer to this question.
2. But we suspect that $\text{BPP}$ is different than $\text{P}$.

One would be correct about the former, but wrong about the latter. As we will see, we do in fact have reasons to believe that $\text{BPP} = \text{P}$. This can be thought of as supporting the extended Church Turing hypothesis that deterministic polynomial-time NAND++ program (or, equivalently, polynomial-time Turing machines) capture what can be feasibly computed in the physical world.

We now survey some of the relations that are known between $\text{BPP}$ and other complexity classes we have encountered. (See also Fig. 19.4.)

![Figure 19.4: Some possibilities for the relations between $\text{BPP}$ and other complexity classes. Most researchers believe that $\text{BPP} = \text{P}$ and that these classes are not powerful enough to solve $\text{NP}$-complete problems, let alone all problems in $\text{EXP}$. However, we have not even been able yet to rule out the possibility that randomness is a “silver bullet” that allows exponential speedup on all problems, and hence $\text{BPP} = \text{EXP}$. As we’ve already seen, we also can’t rule out that $\text{P} = \text{NP}$. Interestingly, in the latter case, $\text{P} = \text{BPP}$.](image)

19.2.1 Solving $\text{BPP}$ in exponential time

It is not hard to see that if $F$ is in $\text{BPP}$ then it can be computed in exponential time.

**Theorem 19.5** — Simulating randomized algorithms in exponential time.

$\text{BPP} \subseteq \text{EXP}$

The proof of Theorem 19.5 readily follows by enumerating over all the (exponentially many) choices for the random coins. We omit the formal proof, as doing it by yourself is an excellent way to get comfortable with Definition 19.1.
19.2.2 Simulating randomized algorithms by circuits or straightline programs.

We have seen in Theorem 12.8 that if \( F \) is in \( P \), then there is a polynomial \( p : \mathbb{N} \rightarrow \mathbb{N} \) such that for every \( n \), the restriction \( F_n \) of \( F \) to inputs \( \{0,1\}^n \) is in \( SIZE(p(n)) \). (In other words, that \( P \subseteq P/\text{poly} \)). A priori it is not at all clear that the same holds for a function in \( \text{BPP} \), but this does turn out to be the case.

![Figure 19.5: The possible guarantees for a randomized algorithm \( A \) computing some function \( F \). In the tables above, the columns correspond to different inputs and the rows to different choices of the random tape. A cell at position \( r, x \) is colored green if \( A(x,r) = F(x) \) (i.e., the algorithm outputs the correct answer) and red otherwise. The standard \( \text{BPP} \) guarantee corresponds to the middle figure, where for every input \( x \), at least two thirds of the choices \( r \) for a random tape will result in \( A \) computing the correct value. That is, every column is colored green in at least two thirds of its coordinates.

In the left figure we have an “average case” guarantee where the algorithm is only guaranteed to output the correct answer with probability two thirds over a random input (i.e., at most one third of the total entries of the table are colored red, but there could be an all red column). The right figure corresponds to the “offline \( \text{BPP} \)” case, with probability at least two thirds over the random choice \( r \), \( r \) will be good for every input. That is, at least two thirds of the rows are all green. Theorem 19.6 is proven by amplifying the success of a \( \text{BPP} \) algorithm until we have the “offline \( \text{BPP} \)” guarantee, and then hardwiring the choice of the randomness \( r \) to obtain a nonuniform deterministic algorithm.

**Theorem 19.6 — Randomness does not help for non uniform computation.** \( \text{BPP} \subseteq P/\text{poly} \). That is, for every \( F \in \text{BPP} \), there exist some \( a, b \in \mathbb{N} \) such that for every \( n > 0 \), \( F_n \in \text{SIZE}(an^b) \) where \( F_n \) is the restriction of \( F \) to inputs in \( \{0,1\}^n \).

**Proof Idea:** The idea behind the proof is that we can first amplify by repetition the probability of success from \( 2/3 \) to \( 1 - 0.1 \cdot 2^{-n} \). This will allow us to show that there exists a single fixed choice of “favorable coins” that would cause the algorithm to output the right answer on all of the possible \( 2^n \) inputs. We can then use the standard “unraveling the loop” technique to transform an RNAND++ program to an RNAND program, and “hardwire” the favorable choice of random coins to transform the RNAND program into a plain old deterministic
NAND program.  

**Proof of Theorem 19.6.** Suppose that $F \in \text{BPP}$. Let $P$ be a polynomial-time RNAND++ program that computes $F$ as per Definition 19.1. Using Theorem 19.3, we can amplify the success probability of $P$ to obtain an RNAND++ program $P'$ that is at most a factor of $O(n)$ slower (and hence still polynomial time) such that for every $x \in \{0,1\}^n$

$$\Pr_{r \sim \{0,1\}^m}[P'(x; r) = F(x)] \geq 1 - 0.1 \cdot 2^{-m},$$  \hspace{1cm} (19.4)

where $m$ is the number of coin tosses that $P'$ uses on inputs of length $n$. We use the notation $P'(x; r)$ to denote the execution of $P'$ on input $x$ and when the result of the coin tosses corresponds to the string $r$.

For every $x \in \{0,1\}^n$, define the “bad” event $B_x$ to hold if $P'(x) \neq F(x)$, where the sample space for this event consists of the coins of $P'$. Then by Eq. (19.4), $\Pr[B_x] \leq 0.1 \cdot 2^{-n}$ for every $x \in \{0,1\}^n$. Since there are $2^n$ many such $x$'s, by the union bound we see that the probability that the union of the events $\{B_x\}_{x \in \{0,1\}^n}$ is at most 0.1. This means that if we choose $r \sim \{0,1\}^m$, then with probability at least 0.9 it will be the case that for every $x \in \{0,1\}^n$, $F(x) = P'(x; r)$. (Indeed, otherwise the event $B_x$ would hold for some $x$.) In particular, because of the mere fact that the the probability of $\bigcup_{x \in \{0,1\}^n} B_x$ is smaller than 1, this means that there exists a particular $r^* \in \{0,1\}^m$ such that

$$P'(x; r^*) = F(x)$$  \hspace{1cm} (19.5)

for every $x \in \{0,1\}^n$.

Now let us use the standard “unravelling the loop” the technique and transform $P'$ into a NAND program $Q$ of polynomial in $n$ size, such that $Q(xr) = P'(x; r)$ for every $x \in \{0,1\}^n$ and $r \in \{0,1\}^m$. Then by “hardwiring” the values $r_0^*, \ldots, r_{m-1}^*$ in place of the last $m$ inputs of $Q$, we obtain a new NAND program $Q_{r^*}$ that satisfies by Eq. (19.5) that $Q_{r^*}(x) = F(x)$ for every $x \in \{0,1\}^n$. This demonstrates that $F_n$ has a polynomial sized NAND program, hence completing the proof of Theorem 19.6.

---

**Randomness and non uniformity** The proof of Theorem 19.6 actually yields more than its statement. We can use the same “unravelling the loop” arguments we’ve used before to show that the restriction to $\{0,1\}^n$ of every function in BPP is also computable by a polynomial-size RNAND program (i.e., NAND program with the RAND operation). Like in the P vs SIZE(poly(n)) case, there are also functions outside BPP whose restrictions...
One amusing anecdote is a recent case where scammers managed to predict the imperfect “pseudorandom generator” used by slot machines to cheat casinos. Unfortunately we don’t know the details of how they did it, since the case was sealed.

The proof of Theorem 19.6 can be summarized as follows: we can replace a \(\text{poly}(n)\)-time algorithm that tosses coins as it runs with an algorithm that uses a single set of coin tosses \(r^* \in \{0, 1\}^{\text{poly}(n)}\) which will be good enough for all inputs of size \(n\). Another way to say it is that for the purposes of computing functions, we do not need “online” access to random coins and can generate a set of coins “offline” ahead of time, before we see the actual input.

But this does not really help us with answering the question of whether \(\text{BPP} = \text{P}\), since we still need to find a way to generate these “offline” coins in the first place. To derandomize an RNAND++ program we will need to come up with a single deterministic algorithm that will work for all input lengths. That is, unlike in the case of RNAND programs, we cannot choose for every input length \(n\) some string \(r^* \in \{0, 1\}^{\text{poly}(n)}\) to use as our random coins.

Can we derandomize randomized algorithms, or does randomness add an inherent extra power for computation? This is a fundamentally interesting question but is also of practical significance. Ever since people started to use randomized algorithms during the Manhattan project, they have been trying to remove the need for randomness and replace it with numbers that are selected through some deterministic process. Throughout the years this approach has often been used successfully, though there have been a number of failures as well.

A common approach people used over the years was to replace the random coins of the algorithm by a “randomish looking” string that they generated through some arithmetic progress. For example, one can use the digits of \(\pi\) for the random tape. Using these type of methods corresponds to what von Neumann referred to as a “state of sin”. (Though this is a sin that he himself frequently committed, as generating true randomness in sufficient quantity was and still is often too expensive.) The reason that this is considered a “sin” is that such a procedure will not work in general. For example, it is easy to modify any probabilistic algorithm \(A\) such as the ones we have seen in

\[\text{BPP/poly} \neq \text{P/poly}\]
Chapter 18, to an algorithm $A'$ that is guaranteed to fail if the random tape happens to equal the digits of $\pi$. This means that the procedure “replace the random tape by the digits of $\pi$” does not yield a general way to transform a probabilistic algorithm to a deterministic one that will solve the same problem. Of course, this procedure does not always fail, but we have no good way to determine when it fails and when it succeeds. This reasoning is not specific to $\pi$ and holds for every deterministically produced string, whether it obtained by $\pi$, $\pi$, the Fibonacci series, or anything else.

An algorithm that checks if its random tape is equal to $\pi$ and then fails seems to be quite silly, but this is but the “tip of the iceberg” for a very serious issue. Time and again people have learned the hard way that one needs to be very careful about producing random bits using deterministic means. As we will see when we discuss cryptography, many spectacular security failures and break-ins were the result of using “insufficiently random” coins.

### 19.3.1 Pseudorandom generators

So, we can’t use any single string to “derandomize” a probabilistic algorithm. It turns out however, that we can use a collection of strings to do so. Another way to think about it is that rather than trying to eliminate the need for randomness, we start by focusing on reducing the amount of randomness needed. (Though we will see that if we reduce the randomness sufficiently, we can eventually get rid of it altogether.)

We make the following definition:

**Definition 19.7 — Pseudorandom generator.** A function $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^m$ is a $(T, \epsilon)$-pseudorandom generator if for every NAND program $P$ with $m$ inputs and one output of at most $T$ lines,

$$\left| \Pr_{s \sim \{0, 1\}^\ell} [P(G(s)) = 1] - \Pr_{r \sim \{0, 1\}^m} [P(r) = 1] \right| < \epsilon \quad (19.6)$$

This is a definition that’s worth reading more than once, and spending some time to digest it. Note that it takes several parameters:

- $T$ is the limit on the number of lines of the program $P$ that the generator needs to “fool”. The larger $T$ is, the stronger the generator.
- $\epsilon$ is how close is the output of the pseudorandom generator to the true uniform distribution over $\{0, 1\}^m$. The smaller $\epsilon$ is, the stronger the generator.
- $\ell$ is the input length and $m$ is the output length.
A pseudorandom generator $G$ maps a short string $s \in \{0,1\}^\ell$ into a long string $r \in \{0,1\}^m$ such that an small program $P$ cannot distinguish between the case that it is provided a random input $r \sim \{0,1\}^m$ and the case that it is provided a “pseudorandom” input of the form $r = G(s)$ where $s \sim \{0,1\}^\ell$. The short string $s$ is sometimes called the seed of the pseudorandom generator, as it is a small object that can be thought as yielding a large “tree of randomness”.

If $\ell \geq m$ then it is trivial to come up with such a generator: on input $s \in \{0,1\}^\ell$, we can output $s_0, \ldots, s_{m-1}$. In this case $\Pr_{s \sim \{0,1\}^\ell}[P(G(s)) = 1]$ will simply equal $\Pr_{r \in \{0,1\}^m}[P(r) = 1]$, no matter how many lines $P$ has. So, the smaller $\ell$ is and the larger $m$ is, the stronger the generator, and to get anything non-trivial, we need $m > \ell$.

Furthermore note that although our eventual goal is to fool probabilistic randomized algorithms that take an unbounded number of inputs, Definition 19.7 refers to finite and deterministic NAND programs.

We can think of a pseudorandom generator as a “randomness amplifier.” It takes an input $s$ of $\ell$ bits chosen at random and expands these $\ell$ bits into an output $r$ of $m > \ell$ pseudorandom bits. If $\epsilon$ is small enough then the pseudorandom bits will “look random” to any NAND program that is not too big. Still, there are two questions we haven’t answered:

- What reason do we have to believe that pseudorandom generators with non-trivial parameters exist?
- Even if they do exist, why would such generators be useful to derandomize
randomized algorithms? After all, Definition 19.7 does not involve RNAND++ or RNAND« programs, but rather deterministic NAND programs with no randomness and no loops.

We will now (partially) answer both questions. For the first question, let us come clean and confess we do not know how to prove that interesting pseudorandom generators exist. By interesting we mean pseudorandom generators that satisfy that \( \epsilon \) is some small constant (say \( \epsilon < 1/3 \)), \( m > \ell \), and the function \( G \) itself can be computed in \( \text{poly}(m) \) time. Nevertheless, Lemma 19.10 (whose statement and proof is deferred to the end of this chapter) shows that if we only drop the last condition (polynomial-time computability), then there do in fact exist pseudorandom generators where \( m \) is exponentially larger than \( \ell \).

At this point you might want to skip ahead and look at the statement of Lemma 19.10. However, since its proof is somewhat subtle, I recommend you defer reading it until you’ve finished reading the rest of this chapter.

19.3.2 From existence to constructivity

The fact that there exists a pseudorandom generator does not mean that there is one that can be efficiently computed. However, it turns out that we can turn complexity “on its head” and use the assumed non existence of fast algorithms for problems such as 3SAT to obtain pseudorandom generators that can then be used to transform randomized algorithms into deterministic ones. This is known as the Hardness vs Randomness paradigm. A number of results along those lines, most of whom are outside the scope of this course, have led researchers to believe the following conjecture:

**Optimal PRG conjecture:** There is a polynomial-time computable function \( PRG : \{0,1\}^* \to \{0,1\} \) that yields an exponentially secure pseudorandom generator.

Specifically, there exists a constant \( \delta > 0 \) such that for every \( \ell \) and \( m < 2^{\ell \delta} \), if we define \( G : \{0,1\}^\ell \to \{0,1\}^m \) as \( G(s) = PRG(s, i) \) for every \( s \in \{0,1\}^\ell \) and \( i \in [m] \), then \( G \) is a \( (2^{\ell \delta}, 2^{-\ell \delta}) \) pseudorandom generator.

The “optimal PRG conjecture” is worth while reading more than once. What it posits is that we can obtain \((T, \epsilon)\) pseudorandom generator \( G \) such
that every output bit of $G$ can be computed in time polynomial in the length $\ell$ of the input, where $T$ is exponentially large in $\ell$ and $\epsilon$ is exponentially small in $\ell$. (Note that we could not hope for the entire output to be computable in $\ell$, as just writing the output down will take too long.)

To understand why we call such a pseudorandom generator “optimal,” it is a great exercise to convince yourself that there exists no $(T, \epsilon)$ pseudorandom generator unless $T$ is smaller than (say) $2^{2^\ell}$ and $\epsilon$ is at least (say) $2^{-2^\ell}$. For the former case note that if we allow a NAND program with much more than $2^\ell$ lines then this NAND program could “hardwire” inside it all the outputs of $G$ on all its $2^\ell$ inputs, and use that to distinguish between a string of the form $G(s)$ and a uniformly chosen string in $\{0,1\}^n$. For the latter case note that by trying to “guess” the input $s$, we can achieve a $2^{-\ell}$ advantage in distinguishing a pseudorandom and uniform input. But working out these details is a highly recommended exercise.

We emphasize again that the optimal PRG conjecture is, as its name implies, a conjecture, and we still do not know how to prove it. In particular, it is stronger than the conjecture that $P \neq \text{NP}$. But we do have some evidence for its truth. There is a spectrum of different types of pseudorandom generators, and there are weaker assumption than the optimal PRG conjecture that suffice to prove that $\text{BPP} = P$. In particular this is known to hold under the assumption that there exists a function $F \in \text{TIME}(2^{O(n)})$ and $\epsilon > 0$ such that for every sufficiently large $n$, $F_n$ is not in $\text{SIZE}(2^{\epsilon n})$. The name “Optimal PRG conjecture” is non standard. This conjecture is sometimes known in the literature as the existence of exponentially strong pseudorandom functions.\footnote{For more on the many interesting results and connections in the study of pseudorandomness, see this monograph of Salil Vadhan.}

### 19.3.3 Usefulness of pseudorandom generators

We now show that optimal pseudorandom generators are indeed very useful, by proving the following theorem:

**Theorem 19.8 — Derandomization of BPP.** Suppose that the optimal PRG conjecture is true. Then $\text{BPP} = P$.

**Proof Idea:** The optimal PRG conjecture tells us that we can achieve exponential expansion of $\ell$ truly random coins into as many as $2^{\ell}$ “pseudorandom coins.” Looked at from the other direction, it allows us to reduce the need for randomness by taking an algorithm that uses $m$ coins and converting it into an algorithm that only uses $O(\log m)$ coins. Now an algorithm of the latter type by can be made fully deterministic by enumerating over all the $2^{O(\log m)}$ (which is polynomial in
possibilities for its random choices.

We now proceed with the proof details.

**Proof of Theorem 19.8.** Let $F \in \text{BPP}$ and let $P$ be a NAND++ program and $a, b, c, d$ constants such that for every $x \in \{0, 1\}^n$, $P(x)$ runs in at most $c \cdot n^d$ steps and $\Pr_{r \sim \{0,1\}^m}[P(x; r) = F(x)] \geq 2/3$. By “unrolling the loop” and hardwiring the input $x$, we can obtain for every input $x \in \{0, 1\}^n$ a NAND program $Q_x$ of at most, say, $T = 10c \cdot n^d$ lines, that takes $m$ bits of input and such that $Q(r) = P(x; r)$.

Now suppose that $G : \{0, 1\}^\ell \to \{0, 1\}$ is a $(T, 0.1)$ pseudorandom generator. Then we could deterministically estimate the probability $p(x) = \Pr_{r \sim \{0,1\}^m}[Q_x(r) = 1]$ up to 0.1 accuracy in time $O(T \cdot 2^\ell \cdot m \cdot \text{cost}(G))$ where $\text{cost}(G)$ is the time that it takes to compute a single output bit of $G$.

The reason is that we know that $\tilde{p}(x) = \Pr_{s \sim \{0, 1\}^\ell}[Q_x(G(s)) = 1]$ will give us such an estimate for $p(x)$, and we can compute the probability $\tilde{p}(x)$ by simply trying all $2^\ell$ possibilities for $s$. Now, under the optimal PRG conjecture we can set $T = 2^\delta \ell$ or equivalently $\ell = \frac{1}{\delta} \log T$, and our total computation time is polynomial in $2^\ell = T^{1/\delta}$.

Since $T \leq 10c \cdot n^d$, this running time will be polynomial in $n$.

This completes the proof, since we are guaranteed that $\Pr_{r \sim \{0,1\}^m}[Q_x(r) = F(x)] \geq 2/3$, and hence estimating the probability $p(x)$ to within 0.1 accuracy is sufficient to compute $F(x)$. ■

**19.4 \text{P} = \text{NP AND \text{BPP} VS \text{P}}**

Two computational complexity questions that we cannot settle are:

- Is $\text{P} = \text{NP}$? Where we believe the answer is negative.
- Is $\text{BPP} = \text{P}$? Where we believe the answer is positive.

However we can say that the “conventional wisdom” is correct on at least one of these questions. Namely, if we’re wrong on the first count, then we’ll be right on the second one:

**Theorem 19.9 — Sipser–Gács Theorem.** If $\text{P} = \text{NP}$ then $\text{BPP} = \text{P}$. Before reading the proof, it is instructive to think why this result is not “obvious.” If $\text{P} = \text{NP}$ then given any randomized algorithm $A$ and input $x$, we will be able to figure out in polynomial time if there is a string $r \in \{0, 1\}^m$ of random coins for $A$ such that $A(xr) = 1$. The problem is that even if $\Pr_{r \sim \{0,1\}^m}[A(xr) = F(x)] \geq 0.9999$, it can still be the
case that even when \( F(x) = 0 \) there exists a string \( r \) such that \( A(xr) = 1 \).

The proof is rather subtle. It is much more important that you understand the statement of the theorem than that you follow all the details of the proof.

**Proof Idea:** The construction follows the "quantifier elimination" idea which we have seen in Theorem 15.4. We will show that for every \( F \in \text{BPP} \), we can reduce the question of some input \( x \) satisfies \( F(x) = 1 \) to the question of whether a formula of the form \( \exists u \in \{0,1\}^m \forall v \in \{0,1\}^k P(u, v) \) is true, where \( m, k \) are polynomial in the length of \( x \) and \( P \) is polynomial-time computable. By Theorem 15.4, if \( P =\text{NP} \) then we can decide in polynomial time whether such a formula is true or false.

The idea behind this construction is that using amplification we can obtain a randomized algorithm \( A \) for computing \( \exists u \in \{0,1\}^m \forall v \in \{0,1\}^k P(u, v) \) output 1 is extremely tiny, and if \( P(x) = 1 \) then it is very large. Now in the case \( P(x) = 1 \), one can show that this means that there exists a small number \( k \) of "shifts" \( s_0, ..., s_{k-1} \) such that the union of the sets \( S \oplus s_i \) covers \( \{0,1\}^m \), while in the case \( P(x) = 0 \) this union will always be of size at most \( k|S| \) which is much smaller than \( 2^m \). We can express the condition that there exists \( s_0, ..., s_{k-1} \) such that \( \bigcup_{i \in [k]} (S \oplus s_i) = \{0,1\}^m \) as a statement with a constant number of quantifiers.

**Proof of Theorem 19.9.** Let \( F \in \text{BPP} \). Using Theorem 19.3, there exists a polynomial-time algorithm \( A \) for computing \( F \) using \( m \) coins such that for every \( x \in \{0,1\}^n \), if \( F(x) = 0 \) then the set \( S \subseteq \{0,1\}^m \) of coins that make \( A \) output 1 is extremely tiny, and if \( F(x) = 1 \) then it is very large. Now in the case \( F(x) = 1 \), one can show that this means that there exists a small number \( k \) of “shifts” \( s_0, ..., s_k \) such that the union of the sets \( S \oplus s_i \) covers \( \{0,1\}^m \), while in the case \( F(x) = 0 \) this union will always be of size at most \( k|S| \) which is much smaller than \( 2^m \). We can express the condition that there exists \( s_0, ..., s_k \) such that \( \bigcup_{i \in [k]} (S \oplus s_i) = \{0,1\}^m \) as a statement with a constant number of quantifiers.

Let \( x \in \{0,1\}^n \), and let \( S_x \subseteq \{0,1\}^m \) be the set \( \{ r \in \{0,1\}^m : A(xr) = 1 \} \). By our assumption, if \( F(x) = 0 \) then \( |S_x| \leq \frac{1}{100m^2} 2^m \) and if \( F(x) = 1 \) then \( |S_x| \geq \left(1 - \frac{1}{100m^2}\right) 2^m \).

For a set \( S \subseteq \{0,1\}^m \) and a string \( s \in \{0,1\}^m \), we define the set \( S \oplus s \) to be \( \{ r \oplus s : r \in S \} \) where \( \oplus \) denotes the XOR operation. That is, \( S \oplus s \) is the set \( S \) “shifted” by \( s \). Note that \( |S \oplus s| = |S| \). (Please make sure that you see why this is true.)

The heart of the proof is the following two claims:

**CLAIM I:** For every subset \( S \subseteq \{0,1\}^m \), if \( |S| \leq \frac{1}{100m} 2^m \), then for every \( s_0, ..., s_{100m-1} \in \{0,1\}^m \), \( \bigcup_{i \in [100m]} (S \oplus s_i) \subseteq \{0,1\}^m \).
If $F \in \text{BPP}$ then through amplification we can ensure that there is an algorithm $A$ to compute $F$ on $n$-length inputs and using $m$ coins such that $\Pr_{r \in \{0,1\}^m}[A(xr) \neq F(xr)] \ll 1/poly(m)$. Hence if $F(x) = 1$ then almost all of the $2^m$ choices for $r$ will cause $A(xr)$ to output 1, while if $F(x) = 0$ then $A(xr) = 0$ for almost all $r$'s. To prove the Sipser–Gács Theorem we consider several “shifts” of the set $\mathcal{S} \subseteq \{0,1\}^m$ of the coins $r$ such that $A(xr) = 1$. If $F(x) = 1$ then we can find a set of $k$ shifts $s_0, \ldots, s_{k-1}$ for which $\cup_{i \in [k]}(\mathcal{S} \oplus s_i) = \{0,1\}^m$. If $F(x) = 0$ then for every such set $|\cup_{i \in [k]}(S_i)| \leq k|\mathcal{S}| \ll 2^m$. We can phrase the question of whether there is such a set of shift using a constant number of quantifiers, and so can solve it in polynomial time if $P = \text{NP}$.

**CLAIM II:** For every subset $\mathcal{S} \subseteq \{0,1\}^m$, if $|\mathcal{S}| \geq \frac{1}{2}2^m$ then there exist a set of string $s_0, \ldots, s_{100m-1}$ such that $\cup_{i \in [100m]}(\mathcal{S} \oplus s_i) \subseteq \{0,1\}^m$.

CLAIM I and CLAIM II together imply the theorem. Indeed, they mean that under our assumptions, for every $x \in \{0,1\}^n$, $F(x) = 1$ if and only if

$$\exists s_0,\ldots,s_{100m-1} \in \{0,1\}^m \cup_{i \in [100m]}(S_x \oplus s_i) = \{0,1\}^m$$

(19.8)

which we can re-write as

$$\exists s_0,\ldots,s_{100m-1} \in \{0,1\}^m \forall w \in \{0,1\}^m \left( w \in (S_x \oplus s_0) \lor w \in (S_x \oplus s_1) \lor \cdots \lor w \in (S_x \oplus s_{100m-1}) \right)$$

(19.9)

or equivalently

$$\exists s_0,\ldots,s_{100m-1} \in \{0,1\}^m \forall w \in \{0,1\}^m \left( A(x(w \oplus s_0)) = 1 \lor A(x(w \oplus s_1)) = 1 \lor \cdots \lor A(x(w \oplus s_{100m-1})) = 1 \right)$$

(19.10)

which (since $A$ is computable in polynomial time) is exactly the type of statement shown in Theorem 15.4 to be decidable in polynomial time if $P = \text{NP}$.

We see that all that is left is to prove CLAIM I and CLAIM II.
CLAIM I follows immediately from the fact that

\[
\bigcup_{i \in [100m-1]} |S_x \oplus s_i| \leq \sum_{i=0}^{100m-1} |S_x \oplus s_i| = \sum_{i=0}^{100m-1} |S_x| = 100m|S_x|.
\]  
(19.11)

To prove CLAIM II, we will use a technique known as the probabilistic method (see the proof of Lemma 19.10 for a more extensive discussion). Note that this is a completely different use of probability than in the theorem statement, we just use the methods of probability to prove an existential statement.

Proof of CLAIM II: Let \( S \subseteq \{0,1\}^m \) with \(|S| \geq 0.5 \cdot 2^m \) be as in the claim’s statement. Consider the following probabilistic experiment: we choose 100m random shifts \( s_0, \ldots, s_{100m-1} \) independently at random in \{0,1\}^m, and consider the event GOOD that

\[
\bigcup_{i \in [100m]} (S \oplus s_i) = \{0,1\}^m.
\]

To prove CLAIM II it is enough to show that \( \Pr[GOOD] > 0 \), since that means that in particular there must exist shifts \( s_0, \ldots, s_{100m-1} \) that satisfy this condition.

For every \( z \in \{0,1\}^m \), define the event \( BAD_z \) to hold if \( z \notin \bigcup_{i \in [100m-1]} (S \oplus s_i) \). The event GOOD holds if \( BAD_z \) fails for every \( z \in \{0,1\}^m \), and so our goal is to prove that \( \Pr[\bigcup_{z \in \{0,1\}^m} BAD_z] < 1 \). By the union bound, to show this, it is enough to show that \( \Pr[BAD_z] < 2^{-m} \) for every \( z \in \{0,1\}^m \). Define the event \( BAD_z^i \) to hold if \( z \notin S \oplus s_i \). Since every shift \( s_i \) is chosen independently, for every fixed \( z \) the events \( BAD_z^0, \ldots, BAD_z^{100m-1} \) are mutually independent, and hence

\[
\Pr[BAD_z] = \Pr[\bigcap_{i \in [100m-1]} BAD_z^i] = \prod_{i=0}^{100m-1} \Pr[BAD_z^i].
\]  
(19.12)

So this means that the result will follow by showing that \( \Pr[BAD_z^i] \leq \frac{1}{2} \) for every \( z \in \{0,1\}^m \) and \( i \in [100m] \) (as that would allow to bound the righthand side of Eq. (19.12) by \( 2^{-100m} \)). In other words, we need to show that for every \( z \in \{0,1\}^m \) with \(|S| \geq \frac{1}{2} 2^m \),

\[
\Pr_{s \in \{0,1\}^m} [z \in S \oplus s] \geq \frac{1}{2}.
\]  
(19.13)

To show this, we observe that \( z \in S \oplus s \) if and only if \( s \in S \oplus z \) (can you see why). Hence we can rewrite the probability on the lefthand side of Eq. (19.13) as \( \Pr_{z \in \{0,1\}^m} [s \in S \oplus z] \) which simply equals \( |S \oplus z|/2^m = |S|/2^m \geq 1/2! \) This concludes the proof of CLAIM I and hence of Theorem 19.9.

\[\square\]
19.5 NON-CONSTRUCTIVE EXISTENCE OF PSEUDORANDOM GENERATORS (ADVANCED, OPTIONAL)

We now show that, if we don’t insist on constructivity of pseudorandom generators, then we can show that there exists pseudorandom generators with output that exponentially larger in the input length.

**Lemma 19.10 — Existence of inefficient pseudorandom generators.** There is some absolute constant $C$ such that for every $\epsilon, T$, if $\ell > C \log T + \log(1/\epsilon)$ and $m \leq T$, then there is an $(T, \epsilon)$ pseudorandom generator $G : \{0,1\}^\ell \to \{0,1\}^m$.

**Proof Idea:** The proof uses an extremely useful technique known as the “probabilistic method” which is not too hard mathematically but can be confusing at first. The idea is to give a “non constructiv” proof of existence of the pseudorandom generator $G$ by showing that if $G$ was chosen at random, then the probability that it would be a valid $(T, \epsilon)$ pseudorandom generator is positive. In particular this means that there exists a single $G$ that is a valid $(T, \epsilon)$ pseudorandom generator. The probabilistic method is just a proof technique to demonstrate the existence of such a function. Ultimately, our goal is to show the existence of a deterministic function $G$ that satisfies the condition.

The above discussion might be rather abstract at this point, but would become clearer after seeing the proof.

**Proof of Lemma 19.10.** Let $\epsilon, T, \ell, m$ be as in the lemma’s statement. We need to show that there exists a function $G : \{0,1\}^\ell \to \{0,1\}^m$ that “fools” every $T$ line program $P$ in the sense of Eq. (19.6). We will show that this follows from the following claim:

**Claim I:** For every fixed NAND program $P$, if we pick $G : \{0,1\}^\ell \to \{0,1\}^m$ at random then the probability that Eq. (19.6) is violated is at most $2^{-2^\ell}$.

Before proving Claim I, let us see why it implies Lemma 19.10. We can identify a function $G : \{0,1\}^\ell \to \{0,1\}^m$ with its “truth table” or simply the list of evaluations on all its possible $2^\ell$ inputs. Since each output is an $m$ bit string, we can also think of $G$ as a string in $\{0,1\}^{m \cdot 2^\ell}$. We define $\mathcal{F}_\ell^m$ to be the set of all functions from $\{0,1\}^\ell$ to $\{0,1\}^m$. As discussed above we can identify $\mathcal{F}_\ell^m$ with $\{0,1\}^{m \cdot 2^\ell}$ and choosing a random function $G \sim \mathcal{F}_\ell^m$ corresponds to choosing a random $m \cdot 2^\ell$-long bit string.

For every NAND program $P$ let $B_P$ be the event that, if we choose $G$ at random from $\mathcal{F}_\ell^m$ then Eq. (19.6) is violated with respect to the program $P$. It is important to understand what is the sample space that the event $B_P$ is defined over, namely this event depends on the choice of $G$ and so $B_P$ is a subset of $\mathcal{F}_\ell^m$. An equivalent way to define

---

7 There is a whole (highly recommended) book by Alon and Spencer devoted to this method.
the event $B_p$ is that it is the subset of all functions mapping $\{0, 1\}^\ell$ to $\{0, 1\}^m$ that violate Eq. (19.6), or in other words:

$$B_p = \left\{ G \in \mathcal{F}^m_\ell \mid \left| \frac{1}{\pi} \sum_{s \in \{0, 1\}^\ell} P(G(s)) - \frac{1}{\pi} \sum_{r \in \{0, 1\}^m} P(r) \right| > \epsilon \right\}$$

(We’ve replaced here the probability statements in Eq. (19.6) with the equivalent sums so as to reduce confusion as to what is the sample space that $B_p$ is defined over.)

To understand this proof it is crucial that you pause here and see how the definition of $B_p$ above corresponds to Eq. (19.14). This may well take re-reading the above text once or twice, but it is a good exercise at parsing probabilistic statements and learning how to identify the sample space that these statements correspond to.

Now, we’ve shown in Theorem 5.5 that up to renaming variables (which makes no difference to program’s functionality) there are $2^{O(T \log T)}$ NAND programs of at most $T$ lines. Since $T \log T < T^2$ for sufficiently large $T$, this means that if Claim I is true, then by the union bound it holds that the probability of the union of $B_p$ over all NAND programs of at most $T$ lines is at most $2^{O(T \log T)2^{-T^2}} < 0.1$ for sufficiently large $T$. What is important for us about the number 0.1 is that it is smaller than 1. In particular this means that there exists a single $G^* \in \mathcal{F}^m_\ell$ such that $G^*$ does not violate Eq. (19.6) with respect to any NAND program of at most $T$ lines, but that precisely means that $G^*$ is a $(T, \epsilon)$ pseudorandom generator.

Hence to conclude the proof of Lemma 19.10, it suffices to prove Claim I. Choosing a random $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^m$ amounts to choosing $\frac{1}{\pi} \sum_{s \in \{0, 1\}^\ell} P(G(s)) = \frac{1}{\pi} \sum_{r \in \{0, 1\}^m} P(r)$ (identifying $\{0, 1\}^\ell$ and $[\pi]$ via the binary representation). This means that proving the claim amounts to showing that for every fixed function $P : \{0, 1\}^m \rightarrow \{0, 1\}$, if $L > 2^{C(\log T + \log \epsilon)}$ (which by setting $C > 4$, we can ensure is larger than $10T^2/\epsilon^2$) then the probability that

$$\left| \frac{1}{T} \sum_{i=0}^{L-1} P(y_i) - \Pr_{s \sim \{0, 1\}^m}[P(s) = 1] \right| > \epsilon$$

is at most $2^{-T^2}$.

Eq. (19.15) follows directly from the Chernoff bound. Indeed, if we let for every $i \in [L]$ the random variable $X_i$ denote $P(y_i)$, then since $y_0, \ldots, y_{L-1}$ is chosen independently at random, these are independently and identically distributed random variables with mean $E_{P \sim \{0, 1\}}[P(y)] = \Pr_{P \sim \{0, 1\}}[P(y) = 1]$ and hence the probability that they deviate from their expectation by $\epsilon$ is at most $2 \cdot 2^{-\epsilon^2 L/2}$.  

Lecture Recap

- We can model randomized algorithms by either adding a special “coin toss” operation or assuming an extra randomly chosen input.
- The class BPP contains the set of Boolean functions that can be computed by polynomial time randomized algorithms.
- We know that $P \subseteq BPP \subseteq \text{EXP}$.
- We also know that $BPP \subseteq P_{\text{poly}}$.
- The relation between $BPP$ and $NP$ is not known, but we do know that if $P = NP$ then $BPP = P$.
- Pseudorandom generators are objects that take a short random “seed” and expand it to a much longer output that “appears random” for efficient algorithms. We conjecture that exponentially strong pseudorandom generators exist. Under this conjecture, $BPP = P$.

19.6 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

19.7 BIBLIOGRAPHICAL NOTES

19.8 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

19.9 ACKNOWLEDGEMENTS
ADVANCED TOPICS
Cryptography - the art or science of “secret writing” - has been around for several millennia, and for almost all of that time Edgar Allan Poe’s quote above held true. Indeed, the history of cryptography is littered with the figurative corpses of cryptosystems believed secure and then broken, and sometimes with the actual corpses of those who have mistakenly placed their faith in these cryptosystems.

Yet, something changed in the last few decades, which is the “rev-
olution” alluded to (and to a large extent initiated by) Diffie and Hellman’s 1976 paper quoted above. New cryptosystems have been found that have not been broken despite being subjected to immense efforts involving both human ingenuity and computational power on a scale that completely dwarves the “code breakers” of Poe’s time. Even more amazingly, these cryptosystem are not only seemingly unbreakable, but they also achieve this under much harsher conditions. Not only do today’s attackers have more computational power but they also have more data to work with. In Poe’s age, an attacker would be lucky if they got access to more than a few encryptions of known messages. These days attackers might have massive amounts of data- terabytes or more - at their disposal. In fact, with public key encryption, an attacker can generate as many ciphertexts as they wish.

The key to this success has been a clearer understanding of both how to define security for cryptographic tools and how to relate this security to concrete computational problems. Cryptography is a vast and continuously changing topic, but we will touch on some of these issues in this chapter.

### 20.1 CLASSICAL CRYPTOSYSTEMS

A great many cryptosystems have been devised and broken throughout the ages. Let us recount just some of these stories. In 1587, Mary the queen of Scots, and the heir to the throne of England, wanted to arrange the assassination of her cousin, queen Elisabeth I of England, so that she could ascend to the throne and finally escape the house arrest under which she had been for the last 18 years. As part of this complicated plot, she sent a coded letter to Sir Anthony Babington.

![Snippet from encrypted communication between queen Mary and Sir Babington](image)

Mary used what’s known as a substitution cipher where each letter is transformed into a different obscure symbol (see Fig. 20.1). At a first look, such a letter might seem rather inscrutable– a meaningless sequence of strange symbols. However, after some thought, one might recognize that these symbols repeat several times and moreover that different symbols repeat with different frequencies. Now it doesn’t take a large leap of faith to assume that perhaps each symbol corre-
sponds to a different letter and the more frequent symbols correspond to letters that occur in the alphabet with higher frequency. From this observation, there is a short gap to completely breaking the cipher, which was in fact done by queen Elisabeth’s spies who used the decoded letters to learn of all the co-conspirators and to convict queen Mary of treason, a crime for which she was executed. Trusting in superficial security measures (such as using “inscrutable” symbols) is a trap that users of cryptography have been falling into again and again over the years. (As in many things, this is the subject of a great XKCD cartoon, see Fig. 20.2.)

The Vigenère cipher is named after Blaise de Vigenère who described it in a book in 1586 (though it was invented earlier by Bellaso). The idea is to use a collection of substitution cyphers - if there are $n$ different ciphers then the first letter of the plaintext is encoded with the first cipher, the second with the second cipher, the $n^{th}$ with the $n^{th}$ cipher, and then the $n + 1^{st}$ letter is again encoded with the first cipher. The key is usually a word or a phrase of $n$ letters, and the $i^{th}$ substitution cipher is obtained by shifting each letter $k_i$ positions in the alphabet. This “flattens” the frequencies and makes it much harder to do frequency analysis, which is why this cipher was considered “unbreakable” for 300+ years and got the nickname “le chiffre indéchiffreable” (“the unbreakable cipher”). Nevertheless, Charles Babbage cracked the Vigenère cipher in 1854 (though he did not publish it). In 1863 Friedrich Kasiski broke the cipher and published the result. The idea is that once you guess the length of the cipher, you can
reduce the task to breaking a simple substitution cipher which can be done via frequency analysis (can you see why?). Confederate generals used Vigenère regularly during the civil war, and their messages were routinely cryptanalzyed by Union officers.

Figure 20.3: Confederate Cipher Disk for implementing the Vigenère cipher

Figure 20.4: Confederate encryption of the message “Gen’l Pemberton: You can expect no help from this side of the river. Let Gen’l Johnston know, if possible, when you can attack the same point on the enemy’s lines. Inform me also and I will endeavor to make a diversion. I have sent some caps. I subjoin a despatch from General Johnston.”

The Enigma cipher had was a mechanical cipher (looking like a typewriter, see Fig. 20.5) where each letter typed would get mapped into a different letter depending on the (rather complicated) key and current state of the machine which had several rotors that rotated at different paces. An identically wired machine at the other end could be used to decrypt. Just as many ciphers in history, this has also been believed by the Germans to be “impossible to break” and even quite late in the war they refused to believe it was broken despite mounting evidence to that effect. (In fact, some German generals refused to believe it was broken even after the war.) Breaking Enigma was an heroic
effort which was initiated by the Poles and then completed by the British at Bletchley Park, with Alan Turing (of the Turing machines) playing a key role. As part of this effort the Brits built arguably the world’s first large scale mechanical computation devices (though they looked more similar to washing machines than to iPhones). They were also helped along the way by some quirks and errors of the German operators. For example, the fact that their messages ended with “Heil Hitler” turned out to be quite useful.

Here is a nice exercise: compute (up to an order of magnitude) the probability that a 50-letter long message composed of random letters will end up not containing the letter “L”.

Figure 20.5: In the Enigma mechanical cipher the secret key would be the settings of the rotors and internal wires. As the operator types up their message, the encrypted version appeared in the display area above, and the internal state of the cipher was updated (and so typing the same letter twice would generally result in two different letters output). Decrypting follows the same process: if the sender and receiver are using the same key then typing the ciphertext would result in the plaintext appearing in the display.

Here is one entertaining anecdote: the Enigma machine would never map a letter to itself. In March 1941, Mavis Batey, a cryptanalyst at Bletchley Park received a very long message that she tried to decrypt. She then noticed a curious property—the message did not contain the letter “L”. She realized that the probability that no “L”s appeared in the message is too small for this to happen by chance. Hence she surmised that the original message must have been composed only of L’s. That is, it must have been the case that the operator, perhaps to test the machine, have simply sent out a message where he repeatedly pressed the letter “L”. This observation helped her decode the next message, which helped inform of a planned Italian attack and secure a resounding British victory in what became known as “the Battle of Cape Matapan”. Mavis also helped break another Enigma machine. Using the information she provided, the Brits were able to feed the Germans with the false information that the main allied invasion would take place in Pas de Calais rather than on Normandy.

In the words of General Eisenhower, the intelligence from Bletchley park was of “priceless value”. It made a huge difference for the Allied
war effort, thereby shortening World War II and saving millions of lives. See also this interview with Sir Harry Hinsley.

20.2 DEFINING ENCRYPTION

Many of the troubles that cryptosystem designers faced over history (and still face!) can be attributed to not properly defining or understanding what are the goals they want to achieve in the first place. Let us focus on the setting of private key encryption. A sender (traditionally called “Alice”) wants to send a message (known also as a plaintext) \( x \in \{0,1\}^* \) to a receiver (traditionally called “Bob”). They would like their message to be kept secret from an adversary who listens in or “eavesdrops” on the communication channel (and is traditionally called “Eve”).

Alice and Bob share a secret key \( k \in \{0,1\}^* \). Alice uses the key \( k \) to “scramble” or encrypt the plaintext \( x \) into a ciphertext \( y \), and Bob uses the key \( k \) to “unscramble” or decrypt the ciphertext \( y \) back into the plaintext \( x \). This motivates the following definition which attempts to capture what it means for an encryption scheme to be valid or “make sense”, regardless of whether or not it is secure:

**Definition 20.1 — Valid encryption scheme.** Let \( L : \mathbb{N} \rightarrow \mathbb{N} \) be some function. A pair of polynomial-time computable functions \((E, D)\) mapping strings to strings is a valid private key encryption scheme (or encryption scheme for short) with plaintext length function \( L(\cdot) \) if for every \( k \in \{0,1\}^n \) and \( x \in \{0,1\}^{L(n)} \),

\[
D(k, E(k,x)) = x .
\] (20.1)

We also require that our encryption schemes are ciphertext length regular in the sense that all ciphertexts corresponding to keys of the same length are of the same length: there is some function \( C : \mathbb{N} \rightarrow \mathbb{N} \) such that for every \( k \in \{0,1\}^n \) and \( x \in \{0,1\}^{L(n)} \),

\[
|E(k,x)| = C(n) .
\]

We will often write the first input (i.e., the key) to the encryption and decryption as a subscript and so can write Eq. (20.1) also as \( D_k(E_k(x)) = x \).

20.3 DEFINING SECURITY OF ENCRYPTION

Definition 20.1 says nothing about the security of \( E \) and \( D \), and even allows the trivial encryption scheme that ignores the key altogether and sets \( E_k(x) = x \) for every \( x \). Defining security is not a trivial matter.

---

2 If you don’t know what “private key” means, you can ignore this adjective for now. For thousands of years, “private key encryption” was synonymous with encryption. Only in the 1970’s was the concept of public key encryption invented, see Definition 20.8.

3 If you don’t know what “private key” means, you can ignore this adjective for now. For thousands of years, “private key encryption” was synonymous with encryption. Only in the 1970’s was the concept of public key encryption invented, see Definition 20.8.
Throughout history, many attacks on cryptosystems were rooted in the cryptosystem designers’ reliance on “security through obscurity”— trusting that the fact their methods are not known to their enemy will protect them from being broken. This is a faulty assumption - if you reuse a method again and again (even with a different key each time) then eventually your adversaries will figure out what you are doing. And if Alice and Bob meet frequently in a secure location to decide on a new method, they might as well take the opportunity to exchange their secrets. These considerations led Auguste Kerckhoffs in 1883 to state the following principle:

*A cryptosystem should be secure even if everything about the system, except the key, is public knowledge.*

Why is it OK to assume the key is secret and not the algorithm? Because we can always choose a fresh key. But of course that won’t help us much if our key is “1234” or “passw0rd!”. In fact, if you use any deterministic algorithm to choose the key then eventually your adversary will figure this out. Therefore for security we must choose the key at random and can restate Kerckhoffs’s principle as follows:

*There is no secrecy without randomness*

This is such a crucial point that is worth repeating:

*There is no secrecy without randomness*

At the heart of every cryptographic scheme there is a secret key, and the secret key is always chosen at random. A corollary of that is that to understand cryptography, you need to know probability theory.

*R Randomness in the real world* Choosing the secrets for cryptography requires generating randomness, which is often done by measuring some “unpredictable” or “high entropy” data, and then applying...
hash functions to the result to “extract” a uniformly random string. Great care must be taken in doing this, and randomness generators often turn out to be the Achilles heel of secure systems.

In 2006 a programmer removed a line of code from the procedure to generate entropy in OpenSSL package distributed by Debian since it caused a warning in some automatic verification code. As a result for two years (until this was discovered) all the randomness generated by this procedure used only the process ID as an “unpredictable” source. This means that all communication done by users in that period is fairly easily breakable (and in particular, if some entities recorded that communication they could break it also retroactively). See XKCD’s take on that incident.

In 2012 two separate teams of researchers scanned a large number of RSA keys on the web and found out that about 4 percent of them are easy to break. The main issue were devices such as routers, internet-connected printers and such. These devices sometimes run variants of Linux- a desktop operating system but without a hard drive, mouse or keyboard, they don’t have access to many of the entropy sources that desktop have. Coupled with some good old fashioned ignorance of cryptography and software bugs, this led to many keys that are downright trivial to break, see this blog post and this web page for more details.

Since randomness is so crucial to security, breaking the procedure to generate randomness can lead to a complete break of the system that uses this randomness. Indeed, the Snowden documents, combined with observations of Shumow and Ferguson, strongly suggest that the NSA has deliberately inserted a trapdoor in one of the pseudorandom generators published by the National Institute of Standards and Technologies (NIST). Fortunately, this generator wasn’t widely adapted but apparently the NSA did pay 10 million dollars to RSA security so the latter would make this generator their default option in their products.

20.4 PERFECT SECRECY

If you think about encryption scheme security for a while, you might come up with the following principle for defining security: “An encryption scheme is secure if it is not possible to recover the key $k$ from $E_k(x)$”. However, a moment’s thought shows that the key is not really what we’re trying to protect. After all, the whole point of an encryption is to protect the confidentiality of the plaintext $x$. So, we can try to define that “an encryption scheme is secure if it is not possible to recover the
plaintext $x$ from $E_k(x)$”. Yet it is not clear what this means either. Suppose that an encryption scheme reveals the first 10 bits of the plaintext $x$. It might still not be possible to recover $x$ completely, but on an intuitive level, this seems like it would be extremely unwise to use such an encryption scheme in practice. Indeed, often even partial information about the plaintext is enough for the adversary to achieve its goals.

The above thinking led Shannon in 1945 to formalize the notion of perfect secrecy, which is that an encryption reveals absolutely nothing about the message. There are several equivalent ways to define it, but perhaps the cleanest one is the following:

**Definition 20.2 — Perfect secrecy.** A valid encryption scheme $(E, D)$ with length $L(\cdot)$ is perfectly secret if for every $n \in \mathbb{N}$ and plaintexts $x, x' \in \{0, 1\}^{L(n)}$, the following two distributions $Y$ and $Y'$ over $\{0, 1\}^*$ are identical:

- $Y$ is obtained by sampling $k \sim \{0, 1\}^n$ and outputting $E_k(x)$.
- $Y'$ is obtained by sampling $k \sim \{0, 1\}^n$ and outputting $E_k(x')$.

This definition might take more than one reading to parse. Try to think of how this condition would correspond to your intuitive notion of “learning no information” about $x$ from observing $E_k(x)$, and to Shannon’s quote in the beginning of this chapter. In particular, suppose that you knew ahead of time that Alice sent either an encryption of $\sigma_0$ or an encryption of $\sigma_1$. Would you learn anything new from observing the encryption of the message that Alice actually sent? It may help you to look at Fig. 20.6.

**20.4.1 Example: Perfect secrecy in the battlefield**

To understand Definition 20.2, suppose that Alice sends only one of two possible messages: “attack” or “retreat”, which we denote by $x_0$ and $x_1$ respectively, and that she sends each one of those messages with probability $1/2$. Let us put ourselves in the shoes of Eve, the eavesdropping adversary. A priori we would have guessed that Alice sent either $x_0$ or $x_1$ with probability $1/2$. Now we observe $y = E_k(x)$ where $k$ is a uniformly chosen key in $\{0, 1\}^n$. How does this new information cause us to update our beliefs on whether Alice sent the plaintext $x_0$ or the plaintext $x_1$?

Before reading the next paragraph, you might want to try the analysis yourself. You may find it useful to
Figure 20.6: For any key length $n$, we can visualize an encryption scheme $(E, D)$ as a graph with a vertex for every one of the $2^{2^n}$ possible plaintexts and for every one of the ciphertexts in $\{0, 1\}^*$ of the form $E_k(x)$ for $k \in \{0, 1\}^n$ and $x \in \{0, 1\}^{2^n}$. For every plaintext $x$ and key $k$, we add an edge labeled $k$ between $x$ and $E_k(x)$. By the validity condition, if we pick any fixed key $k$, the map $x \mapsto E_k(x)$ must be one-to-one. The condition of perfect secrecy simply corresponds to requiring that every two plaintexts $x$ and $x'$ have exactly the same set of neighbors (or multi-set, if there are parallel edges).
The equation Eq. (20.2) is a special case of Bayes’ rule which, although a simple restatement of the formula for conditional probability, is an extremely important and widely used tool in statistics and data analysis.

Let us define $p_0(y)$ to be the probability (taken over $k \sim \{0,1\}^n$) that $y = E_k(x_0)$ and similarly $p_1(y)$ to be $Pr_{k\sim\{0,1\}^n}[y = E_k(x_1)]$. Note that, since Alice chooses the message to send at random, our a priori probability for observing $y$ is $\frac{1}{2}p_0(y) + \frac{1}{2}p_1(y)$. However, as per Definition 20.2, the perfect secrecy condition guarantees that $p_0(y) = p_1(y)$! Let us denote the number $p_0(y) = p_1(y)$ by $p$. By the formula for conditional probability, the probability that Alice sent the message $x_0$ conditioned on our observation $y$ is simply

$$Pr[i = 0|y = E_k(x_i)] = \frac{Pr[i = 0 \land y = E_k(x_i)]}{Pr[y = E_k(x)]}. \quad (20.2)$$

Since the probability that $i = 0$ and $y$ is the ciphertext $E_k(0)$ is equal to $\frac{1}{2} \cdot p_0(y)$, and the a priori probability of observing $y$ is $\frac{1}{2}p_0(y) + \frac{1}{2}p_1(y)$, we can rewrite Eq. (20.2) as

$$Pr[i = 0|y = E_k(x_i)] = \frac{\frac{1}{2}p_0(y)}{\frac{1}{2}p_0(y) + \frac{1}{2}p_1(y)} = \frac{p}{p + p} = \frac{1}{2} \quad (20.3)$$

using the fact that $p_0(y) = p_1(y) = p$. This means that observing the ciphertext $y$ did not help us at all! We still would not be able to guess whether Alice sent “attack” or “retreat” with better than 50/50 odds!

This example can be vastly generalized to show that perfect secrecy is indeed “perfect” in the sense that observing a ciphertext gives Eve no additional information about the plaintext beyond her a priori knowledge.

20.4.2 Constructing perfectly secret encryption

Perfect secrecy is an extremely strong condition, and implies that an eavesdropper does not learn any information from observing the ciphertext. You might think that an encryption scheme satisfying such a strong condition will be impossible, or at least extremely complicated, to achieve. However it turns out we can in fact obtain perfectly secret encryption scheme fairly easily. Such a scheme for two-bit messages is illustrated in Fig. 20.7

In fact, this can be generalized to any number of bits:

**Theorem 20.3 — One Time Pad (Vernam 1917, Shannon 1949).** There is a perfectly secret valid encryption scheme $(E, D)$ with $L(n) = n$.

**Proof Idea:** Our scheme is the one-time pad also known as the “Vernam Cipher”, see Fig. 20.8. The encryption is exceedingly simple: to encrypt a message $x \in \{0,1\}^n$ with a key $k \in \{0,1\}^n$ we simply output
Figure 20.7: A perfectly secret encryption scheme for two-bit keys and messages. The blue vertices represent plaintexts and the red vertices represent ciphertexts, each edge mapping a plaintext $x$ to a ciphertext $y = E_k(x)$ is labeled with the corresponding key $k$. Since there are four possible keys, the degree of the graph is four and it is in fact a complete bipartite graph. The encryption scheme is valid in the sense that for every $k \in \{0, 1\}^2$, the map $x \mapsto E_k(x)$ is one-to-one, which in other words means that the set of edges labeled with $k$ is a matching.

$x \oplus k$ where $\oplus$ is the bitwise XOR operation that outputs the string corresponding to XORing each coordinate of $x$ and $k$. *

Proof of Theorem 20.3. For two binary strings $a$ and $b$ of the same length $n$, we define $a \oplus b$ to be the string $c \in \{0, 1\}^n$ such that $c_i = a_i + b_i \mod 2$ for every $i \in [n]$. The encryption scheme $(E, D)$ is defined as follows: $E_k(x) = x \oplus k$ and $D_k(y) = y \oplus k$.

By the associative law of addition (which works also modulo two), $D_k(E_k(x)) = (x \oplus k) \oplus k = x \oplus (k \oplus k) = x \oplus 0^n = x$, using the fact that for every bit $\sigma \in \{0, 1\}$, $\sigma + \sigma \mod 2 = 0$ and $\sigma + 0 = \sigma \mod 2$. Hence $(E, D)$ form a valid encryption.

To analyze the perfect secrecy property, we claim that for every $x \in \{0, 1\}^n$, the distribution $Y_{x} = E_k(x)$ where $k \sim \{0, 1\}^n$ is simply the uniform distribution over $\{0, 1\}^n$, and hence in particular the distributions $Y_x$ and $Y_{x'}$ are identical for every $x, x' \in \{0, 1\}^n$. Indeed, for every particular $y \in \{0, 1\}^n$, the value $y$ is output by $Y_x$ if and only if $y = x \oplus k$ which holds if and only if $k = x \oplus y$. Since $k$ is chosen uniformly at random in $\{0, 1\}^n$, the probability that $k$ happens to equal $k \oplus y$ is exactly $2^{-n}$, which means that every string $y$ is output by $Y_x$ with probability $2^{-n}$. ■

Figure 20.8: In the one time pad encryption scheme we encrypt a plaintext $x \in \{0, 1\}^n$ with a key $k \in \{0, 1\}^n$ by the ciphertext $x \oplus k$ where $\oplus$ denotes the bitwise XOR operation.
The argument above is quite simple but is worth reading again. To understand why the one-time pad is perfectly secret, it is useful to envision it as a bipartite graph as we’ve done in Fig. 20.7. (In fact the encryption scheme of Fig. 20.7 is precisely the one-time pad for $n = 2$.) For every $n$, the one-time pad encryption scheme corresponds to a bipartite graph with $2^n$ vertices on the “left side” corresponding to the plaintexts in $\{0, 1\}^n$ and $2^n$ vertices on the “right side” corresponding to the ciphertexts $\{0, 1\}^n$. For every $x \in \{0, 1\}^n$ and $k \in \{0, 1\}^n$, we connect $x$ to the vertex $y = E_k(x)$ with an edge that we label with $k$. One can see that this is the complete bipartite graph, where every vertex on the left is connected to all vertices on the right. In particular this means that for every left vertex $x$, the distribution on the ciphertexts obtained by taking a random $k \in \{0, 1\}^n$ and going to the neighbor of $x$ on the edge labeled $k$ is the uniform distribution over $\{0, 1\}^n$. This ensures the perfect secrecy condition.

20.5 NECESSITY OF LONG KEYS

So, does Theorem 20.3 give the final word on cryptography, and means that we can all communicate with perfect secrecy and live happily ever after? No it doesn’t. While the one-time pad is efficient, and gives perfect secrecy, it has one glaring disadvantage: to communicate $n$ bits you need to store a key of length $n$. In contrast, practically used cryptosystems such as AES-128 have a short key of 128 bits (i.e., 16 bytes) that can be used to protect terabytes or more of communication! Imagine that we all needed to use the one time pad. If that was the case, then if you had to communicate with $m$ people, you would have to maintain (securely!) $m$ huge files that are each as long as the length of the maximum total communication you expect with that person. Imagine that every time you opened an account with Amazon, Google, or any other service, they would need to send you in the mail (ideally with a secure courier) a DVD full of random numbers, and every time you suspected a virus, you’d need to ask all these services for a fresh DVD. This doesn’t sound so appealing.

This is not just a theoretical issue. The Soviets have used the one-time pad for their confidential communication since before the 1940’s. In fact, even before Shannon’s work, the U.S. intelligence already knew in 1941 that the one-time pad is in principle “unbreakable” (see page 32 in the Venona document). However, it turned out that the hassle of manufacturing so many keys for all the communication took its toll on the Soviets and they ended up reusing the same keys for more than one message. They did try to use them for completely dif-
ferent receivers in the (false) hope that this wouldn’t be detected. The Venona Project of the U.S. Army was founded in February 1943 by Gene Grabeel (see Fig. 20.9), a former home economics teacher from Madison Heights, Virginia and Lt. Leonard Zubko. In October 1943, they had their breakthrough when it was discovered that the Russians were reusing their keys. In the 37 years of its existence, the project has resulted in a treasure chest of intelligence, exposing hundreds of KGB agents and Russian spies in the U.S. and other countries, including Julius Rosenberg, Harry Gold, Klaus Fuchs, Alger Hiss, Harry Dexter White and many others.

Unfortunately it turns out that such long keys are necessary for perfect secrecy:

**Theorem 20.4 — Perfect secrecy requires long keys.** For every perfectly secret encryption scheme \((E,D)\) the length function \(L\) satisfies \(L(n) \leq n\).

**Proof Idea:** The idea behind the proof is illustrated in Fig. 20.10. If the number of keys is smaller than the number of messages then the neighborhoods of all vertices in the corresponding graphs cannot be identical.

---

*Credit to this discovery is shared by Lt. Richard Hallock, Carrie Berry, Frank Lewis, and Lt. Karl Elmquist, and there are others that have made important contribution to this project. See pages 27 and 28 in the document.*
Figure 20.10: An encryption scheme where the number of keys is smaller than the number of plaintexts corresponds to a bipartite graph where the degree is smaller than the number of vertices on the left side. Together with the validity condition this implies that there will be two left vertices \( x, x' \) with non-identical neighborhoods, and hence the scheme does not satisfy perfect secrecy.

Proof of Theorem 20.4. Let \( E, D \) be a valid encryption scheme with messages of length \( L \) and key of length \( n < L \). We will show that \( (E, D) \) is not perfectly secret by providing two plaintexts \( x_0, x_1 \in \{0,1\}^L \) such that the distributions \( Y_{x_0} \) and \( Y_{x_1} \) are not identical, where \( Y_x \) is the distribution obtained by picking \( k \sim \{0,1\}^n \) and outputting \( E_k(x) \).

We choose \( x_0 = 0^L \). Let \( S_0 \subseteq \{0,1\}^* \) be the set of all ciphertexts that have nonzero probability of being output in \( Y_{x_0} \). That is, \( S = \{y \mid \exists k \sim \{0,1\}^n \ y = E_k(x_0)\} \). Since there are only \( 2^n \) keys, we know that \( |S_0| \leq 2^n \).

We will show the following claim:

Claim I: There exists some \( x_1 \in \{0,1\}^L \) and \( k \in \{0,1\}^n \) such that \( E_k(x_1) \notin S_0 \).

Claim I implies that the string \( E_k(x_1) \) has positive probability of being output by \( Y_{x_1} \) and zero probability of being output by \( Y_{x_0} \) and hence in particular \( Y_{x_0} \) and \( Y_{x_1} \) are not identical. To prove Claim I, just choose a fixed \( k \in \{0,1\}^n \). By the validity condition, the map \( x \mapsto E_k(x) \) is a one to one map of \( \{0,1\}^L \) to \( \{0,1\}^* \), and hence in particular the image of this map: the set \( I = \{y \mid \exists x \in \{0,1\}^L \ y = E_k(x)\} \) has size at least (in fact exactly) \( 2^L \). Since \( |S_0| = 2^n < 2^L \), this means that \( |I| > |S_0| \) and so in particular there exists some string \( y \) in \( I \setminus S_0 \).
But by the definition of $I$ this means that there is some $x \in \{0, 1\}^L$ such that $E_k(x) \not\in S_0$ which concludes the proof of Claim I and hence of Theorem 20.4.

20.6 COMPUTATIONAL SECRECY

To sum up the previous episodes, we now know that:

• It is possible to obtain a perfectly secret encryption scheme with key length the same as the plaintext.

and

• It is not possible to obtain such a scheme with key that is even a single bit shorter than the plaintext.

How does this mesh with the fact that, as we’ve already seen, people routinely use cryptosystems with a 16 byte (i.e., 128 bit) key but many terabytes of plaintext? The proof of Theorem 20.4 does give in fact a way to break all these cryptosystems, but an examination of this proof shows that it only yields an algorithm with time exponential in the length of the key. This motivates the following relaxation of perfect secrecy to a condition known as “computational secrecy”. Intuitively, an encryption scheme is computationally secret if no polynomial time algorithm can break it. The formal definition is below:

**Definition 20.5 — Computational secrecy.** Let $(E, D)$ be a valid encryption scheme where for keys of length $n$, the plaintexts are of length $L(n)$ and the ciphertexts are of length $m(n)$. We say that $(E, D)$ is computationally secret if for every polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$, and large enough $n$, if $P$ is an $m(n)$-input and single output NAND program of at most $p(n)$ lines, and $x_0, x_1 \in \{0, 1\}^{L(n)}$ then

$$\left| \mathbb{E}_{k \sim \{0, 1\}^n} [P(E_k(x_0))] - \mathbb{E}_{k \sim \{0, 1\}^n} [P(E_k(x_1))] \right| < \frac{1}{p(n)} \quad (20.4)$$

**Definition 20.5** requires a second or third read and some practice to truly understand. One excellent exercise to make sure you follow it is to see that if we allow $P$ to be an arbitrary function mapping $\{0, 1\}^{m(n)}$ to $\{0, 1\}$, and we replace the condition in Eq. (20.4) that the lefthand side is smaller than $\frac{1}{p(L(n))}$ with the condition that it is equal to 0 then we get the perfect secrecy condition of Definition 20.2. Indeed if the distributions $E_k(x_0)$ and $E_k(x_1)$ are identical then applying any function $P$ to them we get the
same expectation. On the other hand, if the two distributions above give a different probability for some element \( y^* \in \{0,1\}^{m(n)} \), then the function \( P'(y) \) that outputs 1 iff \( y = y^* \) will have a different expectation under the former distribution than under the latter.

**Definition 20.5** raises two natural questions:

- Is it strong enough to ensure that a computationally secret encryption scheme protects the secrecy of messages that are encrypted with it?

- It is weak enough that, unlike perfect secrecy, it is possible to obtain a computationally secret encryption scheme where the key is much smaller than the message?

To the best of our knowledge, the answer to both questions is Yes. Regarding the first question, it is not hard to show that if, for example, Alice uses a computationally secret encryption algorithm to encrypt either “attack” or “retreat” (each chosen with probability \( 1/2 \)), then as long as she’s restricted to polynomial-time algorithms, an adversary Eve will not be able to guess the message with probability better than, say, 0.51, even after observing its encrypted form. (We omit the proof, but it is an excellent exercise for you to work it out on your own.)

To answer the second question we will show that under the same assumption we used for derandomizing \( \text{BPP} \), we can obtain a computationally secret cryptosystem where the key is almost exponentially smaller than the plaintext.

### 20.6.1 Stream ciphers or the “derandomized one-time pad”

It turns out that if pseudorandom generators exist as in the optimal PRG conjecture, then there exists a computationally secret encryption scheme with keys that are much shorter than the plaintext. The construction below is known as a stream cipher, though perhaps a better name is the “derandomized one-time pad”. It is widely used in practice with keys on the order of a few tens or hundreds of bits protecting many terabytes or even petabytes of communication.

**Theorem 20.6 — Derandomized one-time pad.** Suppose that the optimal PRG conjecture is true. Then for every constant \( a \in \mathbb{N} \) there is a computationally secret encryption scheme \((E, D)\) with plaintext length \( L(n) \) at least \( n^a \).

**Proof Idea:** The proof is illustrated in Fig. 20.11. We simply take the one-time pad on \( L \) bit plaintexts, but replace the key with \( G(k) \) where
Figure 20.11: In a stream cipher or “derandomized one-time pad” we use a pseudo-random generator \( G : \{0,1\}^n \rightarrow \{0,1\}^L \) to obtain an encryption scheme with a key length of \( n \) and plaintexts of length \( L \). We encrypt the plaintext \( x \in \{0,1\}^L \) with key \( k \in \{0,1\}^n \) by the ciphertext \( x \oplus G(k) \).

\( k \) is a string in \( \{0,1\}^n \) and \( G : \{0,1\}^n \rightarrow \{0,1\}^L \) is a pseudorandom generator. *

**Proof of Theorem 20.6.** Since an exponential function of the form \( 2^{\delta n} \) grows faster than any polynomial of the form \( n^a \), under the optimal PRG conjecture we can obtain a polynomial-time computable \( (2^{\delta n}, 2^{-\delta n}) \) pseudorandom generator \( G : \{0,1\}^n \rightarrow \{0,1\}^L \) for \( L = n^a \). We now define our encryption scheme as follows: given key \( k \in \{0,1\}^n \) and plaintext \( x \in \{0,1\}^L \), the encryption \( E_k(x) \) is simply \( x \oplus G(k) \). To decrypt a string \( y \in \{0,1\}^n \) we output \( y \oplus G(k) \). This is a valid encryption since \( G \) is computable in polynomial time and \( (x \oplus G(k)) \oplus G(k) = x \oplus (G(k) \oplus G(k)) = x \) for every \( x \in \{0,1\}^L \).

Computational secrecy follows from the condition of a pseudorandom generator. Suppose, towards a contradiction, that there is a polynomial \( p \), NAND program \( \mathcal{Q} \) of at most \( p(L) \) lines and \( x, x' \in \{0,1\}^{L(n)} \) such that

\[
\left| \mathbb{E}_{k \sim \{0,1\}^n} [Q(E_k(x))] - \mathbb{E}_{k \sim \{0,1\}^n} [Q(E_k(x'))] \right| > \frac{1}{p(L)} \tag{20.5}
\]

which by the definition of our encryption scheme means that

\[
\left| \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')] \right| > \frac{1}{p(L)} \tag{20.6}
\]

Now since (as we saw in the security analysis of the one-time pad), the distribution \( r \oplus x \) and \( r \oplus x' \) are identical, where \( r \sim \{0,1\}^L \), it
follows that
\[ \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x)] - \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x')] = 0. \] 
(20.7)

By plugging Eq. (20.7) into Eq. (20.6) we can derive that
\[ \left| \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x)] + \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x')] - \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')] \right| > \frac{1}{p(L)}. \] 
(20.8)

(Please make sure that you can see why this is true.)

Now we can use the triangle inequality that \(|A + B| \leq |A| + |B|\) for every two numbers \(A, B\), applying it for \(A = \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x)]\) and \(B = \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x')] - \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')]\) to derive
\[ \left| \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x)] \right| + \left| \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x')] - \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x')] \right| > \frac{1}{p(L)}. \] 
(20.9)

In particular, either the first term or the second term of the lefthand-side of Eq. (20.9) must be at least \(\frac{1}{2p(L)}\). Let us assume the first case holds (the second case is analyzed in exactly the same way). Then we get that
\[ \left| \mathbb{E}_{k \sim \{0,1\}^n} [Q(G(k) \oplus x)] - \mathbb{E}_{r \sim \{0,1\}^L} [Q(r \oplus x)] \right| > \frac{1}{2p(L)}. \] 
(20.10)

But if we now define the NAND program \(P_x\) that on input \(r \in \{0,1\}^L\) outputs \(Q(r \oplus x)\) then (since XOR of \(L\) bits can be computed in \(O(L)\) lines), we get that \(P_x\) has \(p(L) + O(L)\) lines and by Eq. (20.10) it can distinguish between an input of the form \(G(k)\) and an input of the form \(r \sim \{0,1\}^L\) with advantage better than \(\frac{1}{2p(L)}\). Since a polynomial is dominated by an exponential, if we make \(L\) large enough, this will contradict the \((2^{2n}, 2^{2n})\) security of the pseudorandom generator \(G\). ■

**Stream ciphers in practice** The two most widely used forms of (private key) encryption schemes in practice are **stream ciphers** and **block ciphers**. (To make things more confusing, a block cipher is always used in some mode of operation and some of these modes effectively turn a block cipher into a stream cipher.)

A block cipher can be thought as a sort of a “random invertible map” from \(\{0,1\}^n\) to \(\{0,1\}^n\), and can be used to construct a pseudorandom generator and from it a stream cipher, or to encrypt data directly using other modes of operations. There are a great many other security notions and considerations for encryption schemes beyond computational secrecy. Many of those involve handling scenarios such as
We model breaking the scheme as Eve outputting 0 or 1 corresponding to whether the message sent was 0 or 1. Note that we could have just as well modified Eve to output 0 instead of 1 and 1 instead of 0. The key point is that a priori Eve only had a 50/50 chance of guessing whether Alice sent 0 or 1 but after seeing the ciphertext this chance increases to better than 99/1.

chosen plaintext, man in the middle, and chosen ciphertext attacks, where the adversary is not just merely a passive eavesdropper but can influence the communication in some way. While this chapter is meant to give you some taste of the ideas behind cryptography, there is much more to know before applying it correctly to obtain secure applications, and a great many people have managed to get it wrong.

20.7 COMPUTATIONAL SECRECY AND NP

We’ve also mentioned before that an efficient algorithm for NP could be used to break all cryptography. We now give an example of how this can be done:

Theorem 20.7 — Breaking encryption using NP algorithm. If \( P = NP \) then there is no computationally secret encryption scheme with \( L(n) > n \).

Furthermore, for every valid encryption scheme \( (E, D) \) with \( L(n) > n + 100 \) there is a polynomial \( p \) such that for every large enough \( n \) there exist \( x_0, x_1 \in \{0, 1\}^{L(n)} \) and a \( p(n) \)-line NAND program \( EV E \) s.t.

\[
\Pr_{i \sim \{0, 1\}, k \sim \{0, 1\}^n}[EVE(E_k(x_i)) = i] \geq 0.99 .
\]

(20.11)

Note that the “furthermore” part is extremely strong. It means that if the plaintext is even a little bit larger than the key, then we can already break the scheme in a very strong way. That is, there will be a pair of messages \( x_0, x_1 \) (think of \( x_0 \) as “sell” and \( x_1 \) as “buy”) and an efficient strategy for Eve such that if Eve gets a ciphertext \( y \) then she will be able to tell whether \( y \) is an encryption of \( x_0 \) or \( x_1 \) with probability very close to 1.\(^7\) The condition \( P = NP \) can be relaxed to \( NP \subseteq BPP \) and even the weaker condition \( NP \subseteq P/poly \) with essentially the same proof.

Proof Idea: The proof follows along the lines of Theorem 20.4 but this time paying attention to the computational aspects. If \( P = NP \) then for every plaintext \( x \) and ciphertext \( y \), we can efficiently tell whether there exists \( k \in \{0, 1\}^n \) such that \( E_k(x) = y \). So, to prove this result we need to show that if the plaintexts are long enough, there would exist a pair \( x_0, x_1 \) such that the probability that a random encryption of \( x_1 \) also is a valid encryption of \( x_0 \) will be very small. The details of how to show this are below. ⋆

Proof of Theorem 20.7. We focus on showing only the “furthermore”
part since it is the more interesting and the other part follows by essentially the same proof.

Suppose that \((E, D)\) is such an encryption, let \(n\) be large enough, and let \(x_0 = 0^{L(n)}\). For every \(x \in \{0, 1\}^{L(n)}\) we define \(S_x\) to be the set of all valid encryption of \(x\). That is \(S_x = \{y \mid \exists_{k \in \{0, 1\}^n} y = E_k(x)\}\). As in the proof of Theorem 20.4, since there are \(2^n\) keys, \(|S_x| \leq 2^n\) for every \(x \in \{0, 1\}^{L(n)}\).

We denote by \(S_0\) the set \(S_{x_0}\). We define our algorithm \(EVE\) to output 0 on input \(y \in \{0, 1\}^*\) if \(y \in S_0\) and to output 1 otherwise. This can be implemented in polynomial time if \(P = NP\), since the key \(S\) can serve the role of an efficiently verifiable solution. (Can you see why?) Clearly \(Pr\[EVE(E_k(x_0)) = 0\] = 1\) and so in the case that \(EVE\) gets an encryption of \(x_0\) then she guesses correctly with probability 1. The remainder of the proof is devoted to showing that there exists \(x_1 \in \{0, 1\}^{L(n)}\) such that \(Pr[EVE(E_k(x_1)) = 0] \leq 0.01\), which will conclude the proof by showing that \(EVE\) guesses wrongly with probability at most \(\frac{1}{2} + \frac{1}{2} \cdot 0.01 < 0.01\).

Consider now the following probabilistic experiment (which we define solely for the sake of analysis). We consider the sample space of choosing \(x\) uniformly in \(\{0, 1\}^{L(n)}\) and define the random variable \(Z_k(x)\) to equal 1 if and only if \(E_k(x) \in S_0\). For every \(k\), the map \(x \mapsto E_k(x)\) is one-to-one, which means that the probability that \(Z_k = 1\) is equal to the probability that \(x \in E_k^{-1}(S_0)\) which is \(\frac{|S_0|}{2^{2n}}\). So by the linearity of expectation \(E[\sum_{k \in \{0, 1\}^n} Z_k] \leq \frac{2^n |S_0|}{2^{2n}} = \frac{2^{2n}}{2^{2n}}\).

We will now use the following extremely simple but useful fact known as the averaging principle (see also Remark 1.6.3): for every random variable \(Z\), if \(E[Z] = \mu\), then with positive probability \(Z \leq \mu\). (Indeed, if \(Z > \mu\) with probability one, then the expected value of \(Z\) will have to be larger than \(\mu\), just like you can’t have a class in which all students got A or A- and yet the overall average is B+.) In our case it means that with positive probability \(\sum_{k \in \{0, 1\}^n} Z_k \leq \frac{2^{2n}}{2^{2n}}\). In other words, there exists some \(x_1 \in \{0, 1\}^{L(n)}\) such that \(\sum_{k \in \{0, 1\}^n} Z_k(x_1) \leq \frac{2^{2n}}{2^{2n}}\). Yet this means that if we choose a random \(k \sim \{0, 1\}^n\), then the probability that \(E_k(x_1) \in S_0\) is at most \(\frac{1}{2^n} \cdot \frac{2^n}{2^{2n}} = 2^{n-L(n)}\).

So, in particular if we have an algorithm \(EVE\) that outputs 0 if \(x \in S_0\) and outputs 1 otherwise, then \(Pr[EVE(E_k(x_1)) = 0] = 1\) and \(Pr[EVE(E_k(x_1)) = 0] \leq 2^{n-L(n)}\) which will be smaller than \(2^{-10} < 0.01\) if \(L(n) \geq n + 10\).

In retrospect Theorem 20.7 is perhaps not surprising. After all, as we’ve mentioned before it is known that the Optimal PRG conjecture (which is the basis for the derandomized one-time pad encryption) is false if \(P = NP\) (and in fact even if \(NP \subseteq BPP\) or even \(NP \subseteq P_{\text{poly}}\)).
20.8 PUBLIC KEY CRYPTOGRAPHY

People have been dreaming about heavier-than-air flight since at least the days of Leonardo Da Vinci (not to mention Icarus from the greek mythology). Jules Verne wrote with rather insightful details about going to the moon in 1865. But, as far as I know, in all the thousands of years people have been using secret writing, until about 50 years ago no one has considered the possibility of communicating securely without first exchanging a shared secret key.

Yet in the late 1960’s and early 1970’s, several people started to question this “common wisdom”. Perhaps the most surprising of these visionaries was an undergraduate student at Berkeley named Ralph Merkle. In the fall of 1974 Merkle wrote in a project proposal for his computer security course that while “it might seem intuitively obvious that if two people have never had the opportunity to prearrange an encryption method, then they will be unable to communicate securely over an insecure channel… I believe it is false”. The project proposal was rejected by his professor as “not good enough”. Merkle later submitted a paper to the communication of the ACM where he apologized for the lack of references since he was unable to find any mention of the problem in the scientific literature, and the only source where he saw the problem even raised was in a science fiction story. The paper was rejected with the comment that “Experience shows that it is extremely dangerous to transmit key information in the clear.”

Merkle showed that one can design a protocol where Alice and Bob can use $T$ invocations of a hash function to exchange a key, but an adversary (in the random oracle model, though he of course didn’t use this name) would need roughly $T^2$ invocations to break it. He conjectured that it may be possible to obtain such protocols where breaking is exponentially harder than using them, but could not think of any concrete way to doing so.

We only found out much later that in the late 1960’s, a few years before Merkle, James Ellis of the British Intelligence agency GCHQ was having similar thoughts. His curiosity was spurred by an old World-War II manuscript from Bell labs that suggested the following way that two people could communicate securely over a phone line. Alice would inject noise to the line, Bob would relay his messages, and then Alice would subtract the noise to get the signal. The idea is that an adversary over the line sees only the sum of Alice’s and Bob’s signals, and doesn’t know what came from what. This got James Ellis thinking whether it would be possible to achieve something like that digitally. As Ellis later recollected, in 1970 he realized that in principle this should be possible, since he could think of an hypothetical black box $B$ that on input a “handle” $\alpha$ and plaintext $x$ would give a
“ciphertext” $y$ and that there would be a secret key $\beta$ corresponding to $\alpha$, such that feeding $\beta$ and $y$ to the box would recover $x$. However, Ellis had no idea how to actually instantiate this box. He and others kept giving this question as a puzzle to bright new recruits until one of them, Clifford Cocks, came up in 1973 with a candidate solution loosely based on the factoring problem; in 1974 another GCHQ recruit, Malcolm Williamson, came up with a solution using modular exponentiation.

But among all those thinking of public key cryptography, probably the people who saw the furthest were two researchers at Stanford, Whit Diffie and Martin Hellman. They realized that with the advent of electronic communication, cryptography would find new applications beyond the military domain of spies and submarines, and they understood that in this new world of many users and point to point communication, cryptography will need to scale up. Diffie and Hellman envisioned an object which we now call “trapdoor permutation” though they called “one way trapdoor function” or sometimes simply “public key encryption”. Though they didn’t have full formal definitions, their idea was that this is an injective function that is easy (e.g., polynomial-time) to compute but hard (e.g., exponential-time) to invert. However, there is a certain trapdoor, knowledge of which would allow polynomial time inversion. Diffie and Hellman argued that using such a trapdoor function, it would be possible for Alice and Bob to communicate securely without ever having exchanged a secret key. But they didn’t stop there. They realized that protecting the integrity of communication is no less important than protecting its secrecy. Thus they imagined that Alice could “run encryption in reverse” in order to certify or sign messages.

At the point, Diffie and Hellman were in a position not unlike physicists who predicted that a certain particle should exist but without any experimental verification. Luckily they met Ralph Merkle, and his ideas about a probabilistic key exchange protocol, together with a suggestion from their Stanford colleague John Gill, inspired them to come up with what today is known as the Diffie Hellman Key Exchange (which unbeknownst to them was found two years earlier at GCHQ by Malcolm Williamson). They published their paper “New Directions in Cryptography” in 1976, and it is considered to have brought about the birth of modern cryptography.

The Diffie-Hellman Key Exchange is still widely used today for secure communication. However, it still felt short of providing Diffie and Hellman’s elusive trapdoor function. This was done the next year by Rivest, Shamir and Adleman who came up with the RSA trapdoor function, which through the framework of Diffie and Hellman yielded not just encryption but also signatures.\(^8\) From this point on began a

---

\(^8\) A close variant of the RSA function was discovered earlier by Clifford Cocks at GCHQ, though as far as I can tell Cocks, Ellis and Williamson did not realize the application to digital signatures.
flurry of advances in cryptography which hasn’t died down till this day.

Figure 20.12: Top left: Ralph Merkle, Martin Hellman and Whit Diffie, who together came up in 1976 with the concept of public key encryption and a key exchange protocol. Bottom left: Adi Shamir, Ron Rivest, and Leonard Adleman who, following Diffie and Hellman’s paper, discovered the RSA function that can be used for public key encryption and digital signatures. Interestingly, one can see the equation $P = NP$ on the blackboard behind them. Right: John Gill, who was the first person to suggest to Diffie and Hellman that they use modular exponentiation as an easy-to-compute but hard-to-invert function.

20.8.1 Defining public key encryption

A public key encryption consists of a triple of algorithms:

- The key generation algorithm, which we denote by $KeyGen$ or $KG$ for short, is a randomized algorithm that outputs a pair of strings $(e, d)$ where $e$ is known as the public (or encryption) key, and $d$ is known as the private (or decryption) key. The key generation algorithm gets as input $1^n$ (i.e., a string of ones of length $n$). We refer to $n$ as the security parameter of the scheme. The bigger we make $n$, the more secure the encryption will be, but also the less efficient it will be.

- The encryption algorithm, which we denote by $E$, takes the encryption key $e$ and a plaintext $x$, and outputs the ciphertext $y = E_e(x)$.

- The decryption algorithm, which we denote by $D$, takes the decryption key $d$ and a ciphertext $y$, and outputs the plaintext $x = D_d(y)$.

We now make this a formal definition:

**Definition 20.8 — Public Key Encryption.** A computationally secret public key encryption with plaintext length $L : \mathbb{N} \rightarrow \mathbb{N}$ is a triple of randomized polynomial-time algorithms $(KG, E, D)$ that satisfy the following conditions:
In a public key encryption, Alice generates a private/public keypair \((e, d)\), publishes \(e\) and keeps \(d\) secret. To encrypt a message for Alice, one only needs to know \(e\). To decrypt it we need to know \(d\).

- For every \(n\), if \((e, d)\) is output by \(KG(1^n)\) with positive probability, and \(x \in \{0, 1\}^{L(n)}\), then \(D_d(E_e(x)) = x\) with probability one.

- For every polynomial \(p\), and sufficiently large \(n\), if \(P\) is a NAND program of at most \(p(n)\) lines then for every \(x, x' \in \{0, 1\}^{L(n)}\), 
  \[|\mathbb{E}[P(e, E_e(x))] - \mathbb{E}[P(e, E_e(x'))]| < \frac{1}{p(n)},\]
  where this probability is taken over the coins of \(KG\) and \(E\).

**Definition 20.8** allows \(E\) and \(D\) to be randomized algorithms. In fact, it turns out that it is necessary for \(E\) to be randomized to obtain computational secrecy. It also turns out that, unlike the private key case, obtaining public key encryption for a single bit of plaintext is sufficient for obtaining public key encryption for arbitrarily long messages. In particular this means that we cannot obtain a perfectly secret public key encryption scheme.

We will not give full constructions for public key encryption schemes in this chapter, but will mention some of the ideas that underlie the most widely used schemes today. These generally belong to one of two families:

- **Group theoretic constructions** based on problems such as integer factoring, and the discrete logarithm over finite fields or elliptic curves.

- **Lattice/coding based constructions** based on problems such as the closest vector in a lattice or bounded distance decoding.

Group theory based encryptions are more widely implemented, but the lattice/coding schemes are recently on the rise, particularly
because encryption schemes based on the former can be broken by quantum computers, which we’ll discuss later in this course.  

20.8.2 Diffie-Hellman key exchange

As just one example of how public key encryption schemes are constructed, let us now describe the Diffie-Hellman key exchange. We describe the Diffie-Hellman protocol in a somewhat of an informal level, without presenting a full security analysis.

The computational problem underlying the Diffie Hellman protocol is the discrete logarithm problem. Let’s suppose that \( g \) is some integer. We can compute the map \( x \mapsto g^x \) and also its inverse \( y \mapsto \log_{g} y \). However, suppose now that we use modular arithmetic and work modulo some prime number \( p \). If \( p \) has \( n \) binary digits and \( g \) is in \([p]\) then we can compute the map \( x \mapsto g^x \mod p \) in time polynomial in \( n \). On the other hand, because of the “wraparound” property of modular arithmetic, we cannot run binary search to find the inverse of this map (known as the discrete logarithm). In fact, there is no known polynomial-time algorithm for computing this discrete logarithm map \( (g, x, p) \mapsto \log_{g} x \mod p \), where we define \( \log_{g} x \mod p \) as the number \( a \in [p] \) such that \( g^a = x \mod p \).

The Diffie-Hellman protocol for Bob to send a message to Alice is as follows:

- **Alice**: Chooses \( p \) to be a random \( n \) bit long prime (which can be done by choosing random numbers and running a primality testing algorithm on them), and \( g \) and \( a \) at random in \([p]\). She sends to Bob the triple \((p, g, g^a \mod p)\).

- **Bob**: Given the triple \((p, g, h)\), Bob sends a message \( x \in \{0, 1\}^L \) to Alice by choosing \( b \) at random in \([p]\), and sending to Alice the pair \((g^b \mod p, \text{rep}(h^b \mod p) \oplus x)\) where \( \text{rep} : [p] \to \{0, 1\}^L \) is some “representation function” that maps \([p]\) to \(\{0, 1\}^L\).

- **Alice**: Given \( g', z \), Alice recovers \( x \) by outputting \( \text{rep}(g'^a \mod p) \oplus z \).

The correctness of the protocol follows from the simple fact that \((g^a)^b = (g^b)^a\) for every \( g, a, b \) and this still holds if we work modulo \( p \). Its security relies on the computational assumption that computing this map is hard, even in a certain “average case” sense (this computational assumption is known as the Decisional Diffie Hellman assumption). The Diffie-Hellman key exchange protocol can be thought of as a public key encryption where the Alice’s first message is the public key, and Bob’s message is the encryption.

One can think of the Diffie-Hellman protocol as being based on a “trapdoor pseudorandom generator” whereas the triple \( g^a, g^b, g^{ab} \)
looks “random” to someone that doesn’t know \(a\), but someone that
does know \(a\) can see that raising the second element to the \(a\)-th power
yields the third element. The Diffie-Hellman protocol can be described
abstractly in the context of any finite Abelian group for which we can
efficiently compute the group operation. It has been implemented
on other groups than numbers modulo \(p\), and in particular Elliptic
Curve Cryptography (ECC) is obtained by basing the Diffie Hellman
on elliptic curve groups which gives some practical advantages.\(^\text{13}\)
Another common group theoretic basis for key-exchange/public key
encryption protocol is the RSA function. A big disadvantage of Diffie-
Hellman (both the modular arithmetic and elliptic curve variants)
and RSA is that both schemes can be broken in polynomial time by a
quantum computer. We will discuss quantum computing later in this
course.

### 20.9 OTHER SECURITY NOTIONS

There is a great deal to cryptography beyond just encryption schemes,
and beyond the notion of a passive adversary. A central objective
is integrity or authentication: protecting communications from being
modified by an adversary. Integrity is often more fundamental than
secrecy: whether it is a software update or viewing the news, you
might often not care about the communication being secret as much as
that it indeed came from its claimed source. Digital signature schemes
are the analog of public key encryption for authentication, and are
widely used (in particular as the basis for public key certificates) to
provide a foundation of trust in the digital world.

Similarly, even for encryption, we often need to ensure security
against active attacks, and so notions such as non-malleability and
adaptive chosen ciphertext security have been proposed. An encryp-
tion scheme is only as secure as the secret key, and mechanisms to
make sure the key is generated properly, and is protected against re-
fresh or even compromise (i.e., forward secrecy) have been studied as
well. Hopefully this chapter provides you with some appreciation for
cryptography as an intellectual field, but does not imbue you with a
false self of confidence in implementing it.

Cryptographic hash functions is another widely used tool with a va-
riety of uses, including extracting randomness from high entropy
sources, achieving hard-to-forge short “digests” of files, protecting
passwords, and much more.

### 20.10 MAGIC

Beyond encryption and signature schemes, cryptographers have man-
aged to obtain objects that truly seem paradoxical and “magical”. We

\(^{13}\) The main advantage in ECC is that the best known algorithms for computing
discrete logarithms over elliptic curve groups take time \(2^{n\epsilon}\) for some \(\epsilon > 0\) where \(n\) is the number of bits to
describe a group element. In contrast, for the multiplicative group modulo a
prime \(p\) the best algorithm take time \(2^{(n^{1/3} \log(n))}\) which means that
(assuming the known algorithms are optimal) we need to set the prime
to be bigger (and so have larger key sizes with corresponding overhead in
communication and computation) to get the same level of security.
briefly discuss some of these objects. We do not give any details, but hopefully this will spark your curiosity to find out more.

20.10.1 Zero knowledge proofs
On October 31, 1903, the mathematician Frank Nelson Cole, gave an hourlong lecture to a meeting of the American Mathematical Society where he did not speak a single word. Rather, he calculated on the board the value $2^{67} - 1$ which is equal to 147,573,952,589,676,412,927, and then showed that this number is equal to 193,707,721 × 761,838,257,287. Cole’s proof showed that $2^{67} - 1$ is not a prime, but it also revealed additional information, namely its actual factors. This is often the case with proofs: they teach us more than just the validity of the statements.

In Zero Knowledge Proofs we try to achieve the opposite effect. We want a proof for a statement $X$ where we can rigorously show that the proofs reveals absolutely no additional information about $X$ beyond the fact that it is true. This turns out to be an extremely useful object for a variety of tasks including authentication, secure protocols, voting, anonymity in cryptocurrencies, and more. Constructing these objects relies on the theory of NP completeness. Thus this theory that originally was designed to give a negative result (show that some problems are hard) ended up yielding positive applications, enabling us to achieve tasks that were not possible otherwise.

20.10.2 Fully homomorphic encryption
Suppose that we are given a bit-by-bit encryption of a string $E_k(x_0), \ldots, E_k(x_{n-1})$. By design, these ciphertexts are supposed to be “completely unscrutable” and we should not be able to extract any information about $x_i$’s from it. However, already in 1978, Rivest, Adleman and Dertouzos observed that this does not imply that we could not manipulate these encryptions. For example, it turns out the security of an encryption scheme does not immediately rule out the ability to take a pair of encryptions $E_k(a)$ and $E_k(b)$ and compute from them $E_k(aNANDb)$ without knowing the secret key $k$. But do there exist encryption schemes that allow such manipulations? And if so, is this a bug or a feature?

Rivest et al already showed that such encryption schemes could be immensely useful, and their utility has only grown in the age of cloud computing. After all, if we can compute NAND then we can use this to run any algorithm $P$ on the encrypted data, and map $E_k(x_0), \ldots, E_k(x_{n-1})$ to $E_k(P(x_0, \ldots, x_{n-1}))$. For example, a client could store their secret data $x$ in encrypted form on the cloud, and have the cloud provider perform all sorts of computation on these data without ever revealing to the provider the private key, and so
without the provider ever learning any information about the secret data.

The question of existence of such a scheme took much longer time to resolve. Only in 2009 Craig Gentry gave the first construction of an encryption scheme that allows to compute a universal basis of gates on the data (known as a Fully Homomorphic Encryption scheme in crypto parlance). Gentry’s scheme left much to be desired in terms of efficiency, and improving upon it has been the focus of an intensive research program that has already seen significant improvements.

20.10.3 Multiparty secure computation

Cryptography is about enabling mutually distrusting parties to achieve a common goal. Perhaps the most general primitive achieving this objective is secure multiparty computation. The idea in secure multiparty computation is that \( n \) parties interact together to compute some function \( F(x_0, \ldots, x_{n-1}) \) where \( x_i \) is the private input of the \( i \)-th party. The crucial point is that there is no commonly trusted party or authority and that nothing is revealed about the secret data beyond the function’s output. One example is an electronic voting protocol where only the total vote count is revealed, with the privacy of the individual voters protected, but without having to trust any authority to either count the votes correctly or to keep information confidential. Another example is implementing a second price (aka Vickrey) auction where \( n-1 \) parties submit bids to an item owned by the \( n \)-th party, and the item goes to the highest bidder but at the price of the second highest bid.

Using secure multiparty computation we can implement second price auction in a way that will ensure the secrecy of the numerical values of all bids (including even the top one) except the second highest one, and the secrecy of the identity of all bidders (including even the second highest bidder) except the top one. We emphasize that such a protocol requires no trust even in the auctioneer itself, that will also not learn any additional information. Secure multiparty computation can be used even for computing randomized processes, with one example being playing Poker over the net without having to trust any server for correct shuffling of cards or not revealing the information.

Lecture Recap

- We can formally define the notion of security of an encryption scheme.
- Perfect secrecy ensures that an adversary does not learn anything about the plaintext from the ciphertext, regardless of their computational powers.
- The one-time pad is a perfectly secret encryption with the length of the key equaling the length of
the message. No perfectly secret encryption can have key shorter than the message.

- Computational secrecy can be as good as perfect secrecy since it ensures that the advantage that computationally bounded adversaries gain from observing the ciphertext is exponentially small. If the optimal PRG conjecture is true then there exists a computationally secret encryption scheme with messages that can be (almost) exponentially bigger than the key.

- There are many cryptographic tools that go well beyond private key encryption. These include public key encryption, digital signatures and hash functions, as well as more “magical” tools such as multiparty secure computation, fully homomorphic encryption, zero knowledge proofs, and many others.

20.11 EXERCISES

Disclaimer Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

20.12 BIBLIOGRAPHICAL NOTES

Much of this text is taken from my lecture notes on cryptography.

Shannon’s manuscript was written in 1945 but was classified, and a partial version was only published in 1949. Still it has revolutionized cryptography, and is the forerunner to much of what followed.

John Nash made seminal contributions in mathematics and game theory, and was awarded both the Abel Prize in mathematics and the Nobel Memorial Prize in Economic Sciences. However, he has struggled with mental illness throughout his life. His biography, A Beautiful Mind was made into a popular movie. It is natural to compare Nash’s 1955 letter to the NSA to Gödel’s letter to von Neumann we mentioned before. From the theoretical computer science point of view, the crucial difference is that while Nash informally talks about exponential vs polynomial computation time, he does not mention the word “Turing Machine” or other models of computation, and it is not clear if he is aware or not that his conjecture can be made mathematically precise (assuming a formalization of “sufficiently complex types
of enciphering”).

The definition of computational secrecy we use is the notion of computational indistinguishability (known to be equivalent to semantic security) that was given by Goldwasser and Micali in 1982.

Although they used a different terminology, Diffie and Hellman already made clear in their paper that their protocol can be used as a public key encryption, with the first message being put in a “public file”. In 1985, ElGamal showed how to obtain a signature scheme based on the Diffie-Hellman ideas, and since he described the Diffie-Hellman encryption scheme in the same paper, it is sometimes also known as ElGamal encryption.

Zero-knowledge proofs were constructed by Goldwasser, Micali, and Rackoff in 1982, and their wide applicability was shown (using the theory of NP completeness) by Goldreich, Micali, and Wigderson in 1986.

Two party and multiparty secure computation protocols were constructed (respectively) by Yao in 1982 and Goldreich, Micali, and Wigderson in 1987. The latter work gave a general transformation from security against passive adversaries to security against active adversaries using zero knowledge proofs.

20.13 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

20.14 ACKNOWLEDGEMENTS
Proves and algorithms

“Let’s not try to define knowledge, but try to define zero-knowledge.”, Shafi Goldwasser.

Proofs have captured human imagination for thousands of years, ever since the publication of Euclid’s Elements, a book second only to the bible in the number of editions printed.

Plan:
- Proofs and algorithms
- Interactive proofs
- Zero knowledge proofs
- Propositions as types, Coq and other proof assistants.

21.1 LECTURE SUMMARY

21.2 EXERCISES

Disclaimer  Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a GitHub issue about the exercise, and optionally complement this with an email to me with more details about the attempted solution.

21.3 BIBLIOGRAPHICAL NOTES

21.4 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)
21.5 ACKNOWLEDGEMENTS
There were two schools of natural philosophy in ancient Greece. Aristotle believed that objects have an *essence* that explains their behavior, and a theory of the natural world has to refer to the *reasons* (or “final cause” to use Aristotle’s language) as to why they exhibit certain phenomena. Democritus believed in a purely mechanistic explanation of the world. In his view, the universe was ultimately composed of elementary particles (or *Atoms*) and our observed phenomena arise from the interactions between these particles according to some local rules. Modern science (arguably starting with Newton) has embraced Democritus’ point of view, of a mechanistic or “clockwork” universe of particles and forces acting upon them.

While the classification of particles and forces evolved with time, to a large extent the “big picture” has not changed from Newton till Einstein. In particular it was held as an axiom that if we knew fully the current *state* of the universe (i.e., the particles and their properties such as location and velocity) then we could predict its future state at any point in time. In computational language, in all these theories the
state of a system with \( n \) particles could be stored in an array of \( O(n) \) numbers, and predicting the evolution of the system can be done by running some efficient (e.g., \( \text{poly}(n) \) time) computation on this array.

### 22.1 THE DOUBLE SLIT EXPERIMENT

Alas, in the beginning of the 20th century, several experimental results were calling into question the “billiard ball” theory of the world. One such experiment is the famous double slit experiment. One way to describe it is as following. Suppose that we buy one of those baseball pitching machines, and aim it at a soft plastic wall, but put a metal barrier with a single slit between the machine and the plastic wall (see Fig. 22.1). If we shoot baseballs at the plastic wall, then some of the baseballs would bounce off the metal barrier, while some would make it through the slit and dent the wall. If we now carve out an additional slit in the metal barrier then more balls would get through, and so the plastic wall would be even more dented.

![Figure 22.1](image)

**Figure 22.1:** In the “double baseball experiment” we shoot baseballs from a gun at a soft wall through a hard barrier that has one or two slits open in it. There is only "constructive interference" in the sense that the dent in each position in the wall when both slits are open is the sum of the dents when each slit is open on its own.

So far this is pure common sense, and it is indeed (to my knowledge) an accurate description of what happens when we shoot baseballs at a wall. However, this is not the same when we shoot photons. Amazingly, if we shoot with a “photon gun” (i.e., a laser) at a wall equipped with photon detectors through some barrier, then (as shown in Fig. 22.2) we sometimes see fewer hits when the two slits are open than one only ones of them is! In particular there are positions in the wall that are hit when the first slit is open, hit when the second gun is open, but are not hit at all when both slits are open!

It seems as if each photon coming out of the gun is aware of the global setup of the experiment, and behaves differently if two slits are open than if only one is. If we try to “catch the photon in the act” and

---

1 A nice illustrated description of the double slit experiment appears in this video.
Figure 22.2: The setup of the double slit experiment in the case of photon or electron guns. We see also destructive interference in the sense that there are some positions on the wall that get fewer hits when both slits are open than they get when only one of the slits is open. Image credit: Wikipedia.

You should read the paragraphs above more than once and make sure you appreciate how truly mind boggling these results are.

22.2 QUANTUM AMPLITUDES

The double slit and other experiments ultimately forced scientists to accept the following picture of the world. Let us go back to the baseball experiment. Suppose that the probability a ball passes through the left slit is \( p_L \) and the probability that it passes through the right slit is \( p_R \). Then, if we shoot \( N \) balls out of each gun, we expect the wall will be hit \((p_L + p_R)N\) times.

In the quantum world, it can sometimes be the case that in both the first and second case the wall is hit with positive probabilities \( p_L \) and \( p_R \) respectively but somehow when both slits are open the wall (or a particular position in it) is not hit at all. It’s almost as if the probabilities can “cancel each other out”.

To understand the way we model this in quantum mechanics, it is helpful to think of a “lazy evaluation” approach to probability. We can think of a probabilistic experiment such as shooting a baseball through two slits in two different ways:

- When a ball is shot, “nature” tosses a coin and decides if it will
go through the left slit (which happens with a certain probability $p_L$), right slit (which happens with a certain probability $p_R$), or bounce back. If it passes through one of the slits then it will hit the wall. Later we can look at the wall and find out whether or not this event happened, but the fact that the event happened or not is determined independently of whether or not we look at the wall.

- The other viewpoint is that when a ball is shot, “nature” computes the probabilities $p_L$ and $p_R$ as before, but does not yet “toss the coin” and determines what happened. Only when we actually look at the wall, nature tosses a coin and with probability $p_L + p_R$ ensures we see a dent. That is, nature uses “lazy evaluation”, and only determines the result of a probabilistic experiment when we decide to measure it.

While the first scenario seems much more natural, the end result in both is the same (the wall is hit with probability $p_L + p_R$) and so the question of whether we should model nature as following the first scenario or second one seems like asking about the proverbial tree that falls in the forest with no one hearing about it.

However, when we want to describe the double slit experiment with photons rather than baseballs, it is the second scenario that lends itself better to a quantum generalization. Quantum mechanics associates a number $\alpha$ known as an amplitude with each probabilistic experiment. This number $\alpha$ can be negative, and in fact even complex. We never observe the amplitudes directly, since whenever we measure an event with amplitude $\alpha$, nature tosses a coin and determines that the event happens with probability $|\alpha|^2$. However, the sign (or in the complex case, phase) of the amplitudes can affect whether two different events have constructive or destructive interference.

If you don’t find the above description confusing and unintuitive, you probably didn’t get it. Please make sure to re-read the above paragraphs until you are thoroughly confused.

Specifically, consider an event that can either occur or not (e.g. “detector number 17 was hit by a photon”). In classical probability, we model this by a probability distribution over the two outcomes: a pair of non-negative numbers $p$ and $q$ such that $p + q = 1$, where $p$ corresponds to the probability that the event occurs and $q$ corresponds to the probability that the event does not occur. In quantum mechanics, we model this also by pair of numbers, which we call amplitudes. This is a pair of (potentially negative or even complex) numbers $\alpha$ and $\beta$ such that $|\alpha|^2 + |\beta|^2 = 1$. The probability that the event occurs is $|\alpha|^2$. 

and the probability that it does not occur is $|\beta|^2$. In isolation, these negative or complex numbers don’t matter much, since we anyway square them to obtain probabilities. But the interaction of positive and negative amplitudes can result in surprising cancellations where somehow combining two scenarios where an event happens with positive probability results in a scenario where it never does.

Complex vs real, other simplifications If (like the author) you are a bit intimidated by complex numbers, don’t worry: you can think of all amplitudes as real (though potentially negative) numbers without loss of understanding. All the “magic” of quantum computing already arises in this case, and so we will often restrict attention to real amplitudes in this chapter.

We will also only discuss so-called pure quantum states, and not the more general notion of mixed states. Pure states turn out to be sufficient for understanding the algorithmic aspects of quantum computing.

More generally, this chapter is not meant to be a complete description of quantum mechanics, quantum information theory, or quantum computing, but rather illustrate the main points where these differ from classical computing.

Quantum mechanics is a mathematical theory that allows us to calculate and predict the results of the double-slit and many other experiments. If you think of quantum mechanics as an explanation as to what “really” goes on in the world, it can be rather confusing. However, if you simply “shut up and calculate” then it works amazingly well at predicting experimental results. In particular, in the double slit experiment, for any position in the wall, we can compute numbers $\alpha$ and $\beta$ such that photons from the first and second slit hit that position with probabilities $|\alpha|^2$ and $|\beta|^2$ respectively. When we open both slits, the probability that the position will be hit is proportional to $|\alpha + \beta|^2$, and so in particular, if $\alpha = -\beta$ then it will be the case that, despite being hit when either slit one or slit two are open, the position is not hit at all when they both are. If you are confused by quantum mechanics, you are not alone: for decades people have been trying to come up with explanations for “the underlying reality” behind quantum mechanics, from Bohmian mechanics to Many worlds as well as many others. However, none of these interpretation have gained universal acceptance and all of those (by design) yield the same experimental predictions. Thus at this point many scientists prefer to just ignore the question of what is the “true reality” and go back to simply “shutting up and calculating”.
22.3 Bell’s Inequality

There is something weird about quantum mechanics. In 1935 Einstein, Podolsky and Rosen (EPR) tried to pinpoint this issue by highlighting a previously unrealized corollary of this theory. They showed that the idea that nature does not determine the results of an experiment until it is measured results in so called “spooky action at a distance”. Namely, making a measurement of one object may instantaneously effect the state (i.e., the vector of amplitudes) of another object in the other end of the universe.

Since the vector of amplitudes is just a mathematical abstraction, the EPR paper was considered to be merely a thought experiment for philosophers to be concerned about, without bearing on experiments. This changed when in 1965 John Bell showed an actual experiment to test the predictions of EPR and hence pit intuitive common sense against the quantum mechanics. Quantum mechanics won: it turns out that it is in fact possible to use measurements to create correlations between the states of objects far removed from one another that cannot be explained by any prior theory. Nonetheless, since the results of these experiments are so obviously wrong to anyone that has ever sat in an armchair, that there are still a number of Bell denialists arguing that this can’t be true and quantum mechanics is wrong.

So, what is this Bell’s Inequality? Suppose that Alice and Bob try to convince you they have telepathic ability, and they aim to prove it via the following experiment. Alice and Bob will be in separate closed rooms. You will interrogate Alice and your associate will interrogate Bob. You choose a random bit \( x \in \{0, 1\} \) and your associate chooses a random \( y \in \{0, 1\} \). We let \( a \) be Alice’s response and \( b \) be Bob’s response. We say that Alice and Bob win this experiment if \( a \oplus b = x \land y \). In other words, Alice and Bob need to output two bits that disagree if \( x = y = 1 \) and agree otherwise.

Now if Alice and Bob are not telepathic, then they need to agree in advance on some strategy. It’s not hard for Alice and Bob to succeed with probability \( 3/4 \): just always output the same bit. However, by doing some case analysis, we can show that no matter what strategy they use, Alice and Bob cannot succeed with higher probability than that.

**Theorem 22.1 — Bell’s Inequality.** For every two functions \( f, g : \{0, 1\} \rightarrow \{0, 1\} \), \( \Pr_{x,y \in \{0, 1\}} [f(x) \oplus g(y) = x \land y] \leq 3/4. \)

**Proof.** Since the probability is taken over all four choices of \( x, y \in \{0, 1\} \), the only way the theorem can be violated if if there exist two

2 If you are extremely paranoid about Alice and Bob communicating with one another, you can coordinate with your assistant to perform the experiment exactly at the same time, and make sure that the rooms are sufficiently far apart (e.g., are on two different continents, or maybe even one is on the moon and another is on earth) so that Alice and Bob couldn’t communicate to each other in time the results of their respective coins even if they do so at the speed of light.

3 This form of Bell’s game was shown by Clauser, Horne, Shimony, and Holt.

4 Theorem 22.1 below assumes that Alice and Bob use deterministic strategies \( f \) and \( g \) respectively. More generally, Alice and Bob could use a probabilistic strategy, or equivalently, each could choose \( f \) and \( g \) from some distributions \( \mathcal{F} \) and \( \mathcal{G} \) respectively. However the averaging principle (Remark 1.6.3) implies that if all possible deterministic strategies succeed with probability at most \( 3/4 \), then the same is true for probabilistic strategies.
functions $f, g$ that satisfy

$$f(x) \oplus g(y) = x \land y \quad (22.1)$$

for all the four choices of $x, y \in \{0, 1\}^2$. Let’s plug in all these four choices and see what we get (below we use the equalities $z \oplus 0 = z$, $z \land 0 = 0$ and $z \land 1 = z$):

$$
\begin{align*}
    f(0) \oplus g(0) &= 0 \quad \text{(plugging in } x = 0, y = 0) \\
    f(0) \oplus g(1) &= 0 \quad \text{(plugging in } x = 0, y = 1) \\
    f(1) \oplus g(0) &= 0 \quad \text{(plugging in } x = 1, y = 0) \\
    f(1) \oplus g(1) &= 1 \quad \text{(plugging in } x = 1, y = 1) 
\end{align*} \quad (22.2)$$

If we XOR together the first and second equalities we get $g(0) \oplus g(1) = 0$ while if we XOR together the third and fourth equalities we get $g(0) \oplus g(1) = 1$, thus obtaining a contradiction. □

An amazing experimentally verified fact is that quantum mechanics allows for “telepathy”. Specifically, it has been shown that using the weirdness of quantum mechanics, there is in fact a strategy for Alice and Bob to succeed in this game with probability larger than $3/4$ (in fact, they can succeed with probability about 0.85, see Lemma 22.2).

### 22.4 QUANTUM WEIRDNESS

Some of the counterintuitive properties that arise from quantum mechanics include:

- **Interference** - As we’ve seen, quantum amplitudes can “cancel each other out”.

- **Measurement** - The idea that amplitudes are negative as long as “no one is looking” and “collapse” (by squaring them) to positive probabilities when they are measured is deeply disturbing. Indeed, as shown by EPR and Bell, this leads to various strange outcomes such as “spooky actions at a distance”, where we can create correlations between the results of measurements in places far removed. Unfortunately (or fortunately?) these strange outcomes have been confirmed experimentally.

- **Entanglement** - The notion that two parts of the system could be connected in this weird way where measuring one will affect the other is known as quantum entanglement.

More accurately, one either has to give up on a “billiard ball type” theory of the universe or believe in telepathy (believe it or not, some scientists went for the latter option).

As counter-intuitive as these concepts are, they have been experimentally confirmed, so we just have to live with them.
22.5 QUANTUM COMPUTING AND COMPUTATION - AN EXECUTIVE SUMMARY.

One of the strange aspects of the quantum-mechanical picture of the world is that unlike in the billiard ball example, there is no obvious algorithm to simulate the evolution of $n$ particles over $t$ time periods in $\text{poly}(n, t)$ steps. In fact, the natural way to simulate $n$ quantum particles will require a number of steps that is exponential in $n$. This is a huge headache for scientists that actually need to do these calculations in practice.

In the 1981, physicist Richard Feynman proposed to “turn this lemon to lemonade” by making the following almost tautological observation:

If a physical system cannot be simulated by a computer in $T$ steps, the system can be considered as performing a computation that would take more than $T$ steps.

So, he asked whether one could design a quantum system such that its outcome $y$ based on the initial condition $x$ would be some function $y = f(x)$ such that (a) we don’t know how to efficiently compute in any other way, and (b) is actually useful for something. In 1985, David Deutsch formally suggested the notion of a quantum Turing machine, and the model has been since refined in works of Deutsch and Josza and Bernstein and Vazirani. Such a system is now known as a quantum computer.

For a while these hypothetical quantum computers seemed useful for one of two things. First, to provide a general-purpose mecha-
nism to simulate a variety of the real quantum systems that people care about, such as various interactions inside molecules in quantum chemistry. Second, as a challenge to the Extended Church Turing hypothesis which says that every physically realizable computation device can be modeled (up to polynomial overhead) by Turing machines (or equivalently, NAND++ / NAND« programs).

Quantum chemistry is important (and in particular understanding it can be a bottleneck for designing new materials, drugs, and more), but it is still a rather niche area within the broader context of computing (and even scientific computing) applications. Hence for a while most researchers (to the extent they were aware of it), thought of quantum computers as a theoretical curiosity that has little bearing to practice, given that this theoretical “extra power” of quantum computer seemed to offer little advantage in the majority of the problems people want to solve in areas such as combinatorial optimization, machine learning, data structures, etc..

To some extent this is still true today. As far as we know, quantum computers, if built, will not provide exponential speed ups for 95% of the applications of computing. In particular, as far as we know, quantum computers will not help us solve NP complete problems in polynomial or even sub-exponential time, though Grover’s algorithm (Remark 22.5) does yield a quadratic advantage in many cases.

However, there is one cryptography-sized exception: In 1994 Peter Shor showed that quantum computers can solve the integer factoring and discrete logarithm in polynomial time. This result has captured the imagination of a great many people, and completely energized research into quantum computing. This is both because the hardness of these particular problems provides the foundations for securing such a huge part of our communications (and these days, our economy), as well as it was a powerful demonstration that quantum computers could turn out to be useful for problems that a-priori seemed to have nothing to do with quantum physics.

As we’ll discuss later, at the moment there are several intensive efforts to construct large scale quantum computers. It seems safe to say that, as far as we know, in the next five years or so there will not be a quantum computer large enough to factor, say, a 1024 bit number. On the other hand, it does seem quite likely that in the very near future there will be quantum computers which achieve some task exponentially faster than the best-known way to achieve the same task with a classical computer. When and if a quantum computer is built that is strong enough to break reasonable parameters of Diffie Hellman, RSA and elliptic curve cryptography is anybody’s guess. It could also be a “self destroying prophecy” whereby the existence of a small-scale quantum computer would cause everyone to shift away

---

7 This “95 percent” is a figure of speech, but not completely so. At the time of this writing, cryptocurrency mining electricity consumption is estimated to use up at least 70Twh or 0.3 percent of the world’s production, which is about 2 to 5 percent of the total energy usage for the computing industry. All the current cryptocurrencies will be broken by quantum computing. Also, for many web servers the TLS protocol (which is based on the current non-lattice based systems would be completely broken by quantum computing) is responsible for about 1 percent of the CPU usage.
to lattice-based crypto which in turn will diminish the motivation to invest the huge resources needed to build a large scale quantum computer.\textsuperscript{8}

\begin{center}
\begin{tcolorbox}
\textbf{Quantum computing and NP.} Despite popular accounts of quantum computers as having variables that can take “zero and one at the same time” and therefore can “explore an exponential number of possibilities simultaneously”, their true power is much more subtle and nuanced. In particular, as far as we know, quantum computers do not enable us to solve NP complete problems such as 3SAT in polynomial or even sub-exponential time. However, Grover’s search algorithm does give a more modest advantage (namely, quadratic) for quantum computers over classical ones for problems in NP. In particular, due to Grover’s search algorithm, we know that the $k$-SAT problem for $n$ variables can be solved in time $O(2^{n/2} \text{poly}(n))$ on a quantum computer for every $k$. In contrast, the best known algorithms for $k$-SAT on a classical computer take roughly $2^{(1 - \frac{1}{f})n}$ steps.
\end{tcolorbox}
\end{center}

\subsection*{22.6 Quantum Systems}

Before we talk about quantum computing, let us recall how we physically realize “vanilla” or classical computing. We model a logical bit that can equal 0 or 1 by some physical system that can be in one of two states. For example, it might be a wire with high or low voltage, charged or uncharged capacitor, or even (as we saw) a pipe with or without a flow of water, or the presence or absence of a soldier crab. A classical system of $n$ bits is composed of $n$ such “basic systems”, each of which can be in either a “zero” or “one” state. We can model the state of such a system by a string $s \in \{0, 1\}^n$. If we perform an operation such as writing to the 17-th bit the NAND of the 3rd and 5th bits, this corresponds to applying a local function to $s$ such as setting $s_{17} = 1 - s_3 \cdot s_5$.

In the probabilistic setting, we would model the state of the system by a distribution. For an individual bit, we could model it by a pair of non-negative numbers $\alpha, \beta$ such that $\alpha + \beta = 1$, where $\alpha$ is the probability that the bit is zero and $\beta$ is the probability that the bit is one. For example, applying the negation (i.e., NOT) operation to this bit corresponds to mapping the pair $(\alpha, \beta)$ to $(\beta, \alpha)$ since the probability that $\text{NOT}(\sigma)$ is equal to 1 is the same as the probability that $\sigma$ is equal to 0. This means that we can think of the NOT function as the linear map $N: \mathbb{R}^2 \rightarrow \mathbb{R}^2$ such that $N \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}$ or equivalently as the matrix

\textsuperscript{8}Of course, given that we’re still hearing of attacks exploiting “export grade” cryptography that was supposed to disappear in 1990’s, I imagine that we’ll still have products running 1024 bit RSA when everyone has a quantum laptop.
\[
\begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}.
\]

If we think of the \(n\)-bit system as a whole, then since the \(n\) bits can take one of \(2^n\) possible values, we model the state of the system as a vector \(p\) of \(2^n\) probabilities. For every \(s \in \{0,1\}^n\), we denote by \(e_s\) the \(2^n\) dimensional vector that has 1 in the coordinate corresponding to \(s\) (identifying it with a number in \([2^n]\)), and so can write \(p\) as
\[
\sum_{s \in \{0,1\}^n} p_s e_s
\]
where \(p_s\) is the probability that the system is in the state \(s\).

Applying the operation above of setting the 17-th bit to the NAND of the 3rd and 5th bits, corresponds to transforming the vector \(p\) to the vector \(Fp\) where \(F : \mathbb{R}^{2^n} \to \mathbb{R}^{2^n}\) is the linear map that maps \(e_s\) to
\[
e_{s_0 \cdots s_{16} (1-s_3 \cdot s_5)} e_{s_{16} \cdots s_{n-1}}.
\]

Please make sure you understand why performing the operation will take a system in state \(p\) to a system in the state \(Fp\). Understanding the evolution of probabilistic systems is a prerequisite to understanding the evolution of quantum systems.

If your linear algebra is a bit rusty, now would be a good time to review it, and in particular make sure you are comfortable with the notions of matrices, vectors, (orthogonal and orthonormal) bases, and norms.

22.6.1 Quantum amplitudes

In the quantum setting, the state of an individual bit (or “qubit”, to use quantum parlance) is modeled by a pair of numbers \((\alpha, \beta)\) such that \(|\alpha|^2 + |\beta|^2 = 1\). While in general these numbers can be complex, for the rest of this chapter, we will often assume they are real (though potentially negative), and hence often drop the absolute value operator. (This turns out not to make much of a difference in explanatory power.) As before, we think of \(\alpha^2\) as the probability that the bit equals 0 and \(\beta^2\) as the probability that the bit equals 1. As we did before, we can model the NOT operation by the map \(N : \mathbb{R}^2 \to \mathbb{R}^2\) where \(N(\alpha, \beta) = (\beta, \alpha)\).

Following quantum tradition, instead of using \(e_0\) and \(e_1\) as we did above, from now on we will denote the vector \((1, 0)\) by \(|0\rangle\) and the vector \((0, 1)\) by \(|1\rangle\) (and moreover, think of these as column vectors). This is known as the Dirac “ket” notation. This means that NOT is the unique linear map \(N : \mathbb{R}^2 \to \mathbb{R}^2\) that satisfies \(N|0\rangle = |1\rangle\) and \(N|1\rangle = |0\rangle\). In other words, in the quantum case, as in the probabilistic

\(9\) Since \(\{e_s\}_{s \in \{0,1\}^n}\) is a basis for \(\mathbb{R}^{2^n}\), it suffices to define the map \(F\) on vectors of this form.
case, NOT corresponds to the matrix

\[
N = \begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}.
\] (22.3)

In classical computation, we typically think that there are only two operations that we can do on a single bit: keep it the same or negate it. In the quantum setting, a single bit operation corresponds to any linear map \( OP : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \) that is \textit{norm preserving} in the sense that for every \( \alpha, \beta \), if we apply \( OP \) to the vector \( \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \) then we obtain a vector \( \begin{pmatrix} \alpha' \\ \beta' \end{pmatrix} \) such that \( \alpha'^2 + \beta'^2 = \alpha^2 + \beta^2 \). Such a linear map \( OP \) corresponds to a \textit{unitary} two by two matrix.\(^{10}\) Keeping the bit the same corresponds to the matrix \( I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \) and (as we’ve seen) the NOT operations corresponds to the matrix \( N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \). But there are other operations we can use as well. One such useful operation is the \textit{Hadamard} operation, which corresponds to the matrix

\[
H = \frac{1}{\sqrt{2}} \begin{pmatrix}
+1 & +1 \\
+1 & -1
\end{pmatrix}.
\] (22.4)

In fact it turns out that Hadamard is all that we need to add to a classical universal basis to achieve the full power of quantum computing.

\section*{22.6.2 Recap}

The \textit{state} of a quantum system of \( n \) qubits is modeled by an \( 2^n \) dimensional vector \( \psi \) of unit norm (i.e., squares of all coordinates sums up to 1), which we write as \( \psi = \sum_{x \in \{0,1\}^n} \psi_x |x\rangle \) where \( |x\rangle \) is the column vector that has 0 in all coordinates except the one corresponding to \( x \) (identifying \( \{0,1\}^n \) with the numbers \( \{0,\ldots,2^n-1\} \)). We use the convention that if \( a, b \) are strings of lengths \( k \) and \( \ell \) respectively then we can write the \( 2^{k+\ell} \) dimensional vector with 1 in the \( ab \)-th coordinate and zero elsewhere not just as \(|ab\rangle\) but also as \(|a\rangle|b\rangle\). In particular, for every \( x \in \{0,1\}^n \), we can write the vector \(|x\rangle\) also as \(|x_0\rangle|x_1\rangle\cdots|x_{n-1}\rangle\). This notation satisfies certain nice distributive laws such as \(|a\rangle(|b\rangle + |b'\rangle)|c\rangle = |abc\rangle + |ab'c\rangle\).

A \textit{quantum operation} on such a system is modeled by a \( 2^n \times 2^n \) \textit{unitary matrix} \( U \) (one that satisfies \( UU^\top = I \) where \( U^\top \) is the transpose operation, or conjugate transpose for complex matrices). If the system is in state \( \psi \) and we apply to it the operation \( U \), then the new state of the system is \( U\psi \).
When we measure an \(n\)-qubit system in a state \(\psi = \sum_{x \in \{0, 1\}^n} \psi_x |x\rangle\),
then we observe the value \(x \in \{0, 1\}^n\) with probability \(|\psi_x|^2\). In this case, the system collapses to the state \(|x\rangle\).

### 22.7 ANALYSIS OF BELL’S INEQUALITY (OPTIONAL)

Now that we have the notation in place, we can show a strategy for Alice and Bob to display “quantum telepathy” in Bell’s Game. Recall that in the classical case, Alice and Bob can succeed in the “Bell Game” with probability at most \(3/4 = 0.75\). We now show that quantum mechanics allows them to succeed with probability at least 0.8.\(^\dagger\)

**Lemma 22.2** There is a 2-qubit quantum state \(\psi \in \mathbb{C}^4\) so that if Alice has access to the first qubit of \(\psi\), she can manipulate and measure it and output \(a \in \{0, 1\}\) and Bob has access to the second qubit of \(\psi\) and can manipulate and measure it and output \(b \in \{0, 1\}\) then \(\Pr[a \oplus b = x \land y] \geq 0.8\).

**Proof.** Alice and Bob will start by preparing a 2-qubit quantum system in the state

\[
\psi = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle
\]

(this state is known as an **EPR pair**). Alice takes the first qubit of the system to her room, and Bob takes the qubit to his room. Now, when Alice receives \(x\) if \(x = 0\) she does nothing and if \(x = 1\) she applies the unitary map \(R_{-\pi/8}\) to her qubit where \(R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}\) is the unitary operation corresponding to rotation in the plane with angle \(\theta\). When Bob receives \(y\), if \(y = 0\) he does nothing and if \(y = 1\) he applies the unitary map \(R_{\pi/8}\) to his qubit. Then each one of them measures their qubit and sends this as their response.

Recall that to win the game Bob and Alice want their outputs to be more likely to differ if \(x = y = 1\) and to be more likely to agree otherwise. We will split the analysis in one case for each of the four possible values of \(x\) and \(y\).

- **Case 1:** \(x = 0\) and \(y = 0\). If \(x = y = 0\) then the state does not change. Because the state \(\psi\) is proportional to \(|00\rangle + |11\rangle\), the measurements of Bob and Alice will always agree (if Alice measures 0 then the state collapses to \(|00\rangle\) and so Bob measures 0 as well, and similarly for 1). Hence in the case \(x = y = 1\), Alice and Bob always win.

- **Case 2:** \(x = 0\) and \(y = 1\). If \(x = 0\) and \(y = 1\) then after Alice measures her bit, if she gets 0 then the system collapses to the state \(|00\rangle\), in which case after Bob performs his rotation, his qubit is in the state \(\cos(\pi/8) |0\rangle + \sin(\pi/8) |1\rangle\). Thus, when Bob measures his
We are using the (not too hard) observation that the result of this experiment is the same regardless of the order in which Alice and Bob apply their rotations and measurements.

If the qubit, he will get 0 (and hence agree with Alice) with probability \( \cos^2(\pi/8) \geq 0.85 \). Similarly, if Alice gets 1 then the system collapses to \( |11\rangle \), in which case after rotation Bob’s qubit will be in the state \(-\sin(\pi/8)|0\rangle + \cos(\pi/8)|1\rangle\) and so once again he will agree with Alice with probability \( \cos^2(\pi/8) \).

The analysis for Case 3, where \( x = 1 \) and \( y = 0 \), is completely analogous to Case 2. Hence Alice and Bob will agree with probability \( \cos^2(\pi/8) \) in this case as well.\(^\dagger\)

**Case 4:** \( x = 1 \) and \( y = 1 \). For the case that \( x = 1 \) and \( y = 1 \), after both Alice and Bob perform their rotations, the state will be proportional to

\[
R_{-\pi/8}|0\rangle R_{\pi/8}|0\rangle + R_{-\pi/8}|1\rangle R_{\pi/8}|1\rangle. \tag{22.6}
\]

Intuitively, since we rotate one state by 45 degrees and the other state by -45 degrees, they will become orthogonal to each other, and the measurements will behave like independent coin tosses that agree with probability 1/2. However, for the sake of completeness, we now show the full calculation.

Opening up the coefficients and using \( \cos(-x) = \cos(x) \) and \( \sin(-x) = -\sin(x) \), we can see that Eq. (22.6) is proportional to

\[
\cos^2(\pi/8)|00\rangle + \cos(\pi/8)\sin(\pi/8)|01\rangle \\
-\sin(\pi/8)\cos(\pi/8)|10\rangle + \sin^2(\pi/8)|11\rangle \\
-\sin^2(\pi/8)|00\rangle + \sin(\pi/8)\cos(\pi/8)|01\rangle \\
-\cos(\pi/8)\sin(\pi/8)|10\rangle + \cos^2(\pi/8)|11\rangle. \tag{22.7}
\]

Using the trigonometric identities \( 2\sin(\alpha)\cos(\alpha) = \sin(2\alpha) \) and \( \cos^2(\alpha) - \sin^2(\alpha) = \cos(2\alpha) \), we see that the probability of getting any one of \( |00\rangle, |10\rangle, |01\rangle, |11\rangle \) is proportional to \( \cos(\pi/4) = \sin(\pi/4) = \frac{\sqrt{2}}{2} \). Hence all four options for \( (a, b) \) are equally likely, which mean that in this case \( a = b \) with probability 0.5.

Taking all the four cases together, the overall probability of winning the game is at least \( \frac{1}{4} \cdot 1 + \frac{1}{2} \cdot 0.85 + \frac{1}{4} \cdot 0.5 = 0.8 \).

---

\(^\dagger\) We are using the (not too hard) observation that the result of this experiment is the same regardless of the order in which Alice and Bob apply their rotations and measurements.

---

**R Quantum vs probabilistic strategies** It is instructive to understand what is it about quantum mechanics that enabled this gain in Bell’s Inequality. For this, consider the following analogous probabilistic strategy for Alice and Bob. They agree that each one of them output 0 if he or she get 0 as input and outputs 1 with probability \( p \) if they get 1 as input. In this case one can see that their success probability would be \( \frac{1}{2} \cdot 1 + \frac{1}{2}(1 - p) + \frac{1}{4}[2p(1 - p)] = 0.75 - 0.5p^2 \leq 0.75 \). The quantum strategy we described above can be thought of as a variant of the probabilistic strategy.
for parameter \( p \) set to \( \sin^2(\pi/8) = 0.15 \). But in the case \( x = y = 1 \), instead of disagreeing only with probability \( 2p(1 - p) = 1/4 \), because we can use these negative probabilities in the quantum world and rotate the state in opposite directions, and hence the probability of disagreement ends up being \( \sin^2(\pi/4) = 0.5 \).

### 22.8 QUANTUM COMPUTATION

Recall that in the classical setting, we modeled computation as obtained by a sequence of basic operations. We had two types of computational models:

- **Non uniform models of computation** such as Boolean circuits and NAND programs, where a finite function \( f : \{0,1\}^n \rightarrow \{0,1\} \) is computable in size \( T \) if it can be expressed as a combination of \( T \) basic operations (gates in a circuit or lines in a NAND program).

- **Uniform models of computation** such as Turing machines and NAND++ programs, where an infinite function \( F : \{0,1\}^* \rightarrow \{0,1\} \) is computable in time \( T(n) \) if there is a single algorithm that on input \( x \in \{0,1\}^n \) evaluates \( F(x) \) using at most \( T(n) \) basic steps.

When considering efficient computation, we defined the class \( \mathsf{P} \) to consist of all infinite functions \( F : \{0,1\}^* \rightarrow \{0,1\} \) that can be computed by a Turing machine or NAND++ program in time \( p(n) \) for some polynomial \( p(\cdot) \). We defined the class \( \mathsf{P}_{\text{poly}} \) to consists of all infinite functions \( F : \{0,1\}^* \rightarrow \{0,1\} \) such that for every \( n \), the restriction \( F_n \) of \( F \) to \( \{0,1\}^n \) can be computed by a Boolean circuit or NAND program of size at most \( p(n) \) for some polynomial \( p(\cdot) \).

We will do the same for quantum computation, focusing mostly on the non uniform setting of quantum circuits, since that is simpler, and already illustrates the important differences with classical computing.

#### 22.8.1 Quantum circuits

A quantum circuit is analogous to a Boolean circuit, and can be described as a directed acyclic graph. One crucial difference that the out degree of every vertex in a quantum circuit is at most one. This is because we cannot “reuse” quantum states without measuring them (which collapses their “probabilities”). Therefore, we cannot use the same qubit as input for two different gates.\(^{13}\) Another more technical difference is that to express our operations as unitary matrices, we will need to make sure all our gates are reversible. This is not hard to ensure. For example, in the quantum context, instead of thinking of \( \text{NAND} \) as a (non reversible) map from \( \{0,1\}^2 \)
to \{0,1\}, we will think of it as the reversible map on three qubits that maps \(a,b,c\) to \(a,b,c \oplus \textit{NAND}(a,b)\) (i.e., flip the last bit if \(\textit{NAND}\) of the first two bits is 1). Equivalently, the \(\textit{NAND}\) operation corresponds to the \(8 \times 8\) unitary matrix \(U_{\textit{NAND}}\) such that (identifying \(\{0,1\}^3\) with \([8]\)) for every \(a,b,c \in \{0,1\}\), if \(|a\ b\ c\rangle\) is the basis element with 1 in the 1st position and zero elsewhere, then

\[
U_{\textit{NAND}} |a\ b\ c\rangle = |a\ b\ c \oplus \textit{NAND}(a,b)\rangle.
\]

If we order the rows and columns as 000, 001, 010, \ldots, 111, then \(U_{\textit{NAND}}\) can be written as the following matrix:

\[
U_{\textit{NAND}} = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{pmatrix}
\]  

(22.8)

Another way to write define \(H\) is that for \(b \in \{0,1\}\), \(H|b\rangle = \frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} (-1)^b |1\rangle\). We define \(\textit{HAD}\) to be the \(2^n \times 2^n\) unitary matrix that applies \(\textit{HAD}\) to the \(i\)-th qubit and leaves the others intact. Using the ket notation, we can write this as

\[
\sum_{x \in \{0,1\}^i} v_x |x\rangle = \frac{1}{\sqrt{2}} \sum_{x \in \{0,1\}^n} |x_0 \cdots x_{i-1} \rangle \left( |0\rangle + (-1)^{x_i} |1\rangle \right) |x_i \cdots x_{n-1}\rangle.
\]

(22.10)

A quantum circuit is obtained by composing these basic operations on some \(m\) qubits. If \(m \geq n\), we use a circuit to compute a function \(f: \{0,1\}^n \to \{0,1\}\):

- On input \(x\), we initialize the system to hold \(x_0,\ldots,x_{n-1}\) in the first \(n\) qubits, and initialize all remaining \(m - n\) qubits to zero.
- We execute each elementary operation one by one.

14 Readers familiar with quantum computing should note that \(U_{\textit{NAND}}\) is a close variant of the so called \(\textit{Toffoli gate}\) and so QNAND programs correspond to quantum circuits with the Hadamard and Toffoli gates.
• At the end of the computation, we measure the system, and output the result of the last qubit (i.e. the qubit in location \( m - 1 \)).

• We say that the circuit computes \( f \), if the probability that this output equals \( f(x) \) is at least \( 2/3 \). Note that this probability is obtained by summing up the squares of the amplitudes of all coordinates in the final state of the system corresponding to vectors \( |y⟩ \) where \( y_m = f(x) \).

Formally this is defined as follows:

**Definition 22.3 — Quantum circuit.** A quantum circuit of \( m \) inputs and \( s \) gates over the \( \{U_{NAND}, HAD\} \) basis is a sequence of \( s \) unitary \( 2^n \times 2^n \) matrices \( U_0, \ldots, U_{s-1} \) such that each matrix \( U_0 \) is either of the form \( NAND^{i,j,k} \) for \( i, j, k \in [n] \) or \( HAD^i \) for \( i \in [n] \).

A quantum circuit computes a function \( f : \{0,1\}^n \rightarrow \{0,1\} \) if the following is true for every \( x \in \{0,1\}^n \):

Let \( v \) be the vector

\[
v = U_{s-1} U_{s-2} \cdots U_1 U_0 |x0^{m-n}\rangle
\]

and write \( v \) as \( \sum_{y \in \{0,1\}^m} v_y |y⟩ \). Then

\[
\sum_{y \in \{0,1\}^m \text{ s.t. } y_{m-1} = f(x)} |v_y|^2 \geq \frac{2}{3}.
\]

Once we have the notion of quantum circuits, we can define the quantum analog of \( \text{P/poly} \) (i.e., define the class of functions computable by polynomial size quantum circuits) as follows:

**Definition 22.4 — BQP/poly.** Let \( F : \{0,1\}^* \rightarrow \{0,1\} \). We say that \( F \in \text{BQP/poly} \) if there exists some polynomial \( p : \mathbb{N} \rightarrow \mathbb{N} \) such that for every \( n \in \mathbb{N} \), if \( F_n \) is the restriction of \( F \) to inputs of length \( n \), then there is a quantum circuit of size at most \( p(n) \) that computes \( F_n \).

**The obviously exponential fallacy** A priori it might seem “obvious” that quantum computing is exponentially powerful, since to perform a quantum computation on \( n \) bits we need to maintain the \( 2^n \)
dimensional state vector and apply $2^n \times 2^n$ matrices to it. Indeed popular descriptions of quantum computing (too) often say something along the lines that the difference between quantum and classical computer is that a classical bit can either be zero or one while a qubit can be in both states at once, and so in many qubits a quantum computer can perform exponentially many computations at once.

Depending on how you interpret it, this description is either false or would apply equally well to probabilistic computation, even though we’ve already seen that every randomized algorithm can be simulated by a similar-sized circuit, and in fact we conjecture that $\text{BPP} = \text{P}$.

Moreover, this “obvious” approach for simulating a quantum computation will take not just exponential time but exponential space as well, while can be shown that using a simple recursive formula one can calculate the final quantum state using polynomial space (in physics this is known as “Feynman path integrals”). So, the exponentially long vector description by itself does not imply that quantum computers are exponentially powerful. Indeed, we cannot prove that they are (i.e., as far as we know, every QNAND program could be simulated by a NAND program with polynomial overhead), but we do have some problems (integer factoring most prominently) for which they do provide exponential speedup over the currently best known classical (deterministic or probabilistic) algorithms.

**22.8.2 QNAND programs (optional)**

Just like in the classical case, there is an equivalence between circuits and straightline programs, and so we can define the programming language QNAND that is the quantum analog of our NAND programming language. To do so, we only add a single operation: $\text{HAD}(\text{foo})$ which applies the single-bit operation $H$ to the variable $\text{foo}$. We also use the following interpretation to make NAND reversible: $\text{foo} = \text{NAND}(\text{bar},\text{blah})$ means that we modify $\text{foo}$ to be the XOR of its original value and the NAND of $\text{bar}$ and $\text{blah}$. (In other words, apply the 8 by 8 unitary transformation $U_{\text{NAND}}$ defined above to the three qubits corresponding to $\text{foo}, \text{bar}$ and $\text{blah}$.) If $\text{foo}$ is initialized to zero then this makes no difference.

If $P$ is a QNAND program with $n$ input variables, $\ell$ workspace variables, and $m$ output variables, then running it on the input $x \in \{0,1\}^n$ corresponds to setting up a system with $n+m+\ell$ qubits and performing the following process:

1. We initialize the input variables $X[0] \ldots X[n-1]$ to $x_0, \ldots, x_{n-1}$
2. We execute the program line by line, applying the corresponding physical operation \( H \) or \( U_{NAND} \) to the qubits that are referred to by the line.

3. We measure the output variables \( Y[0], \ldots, Y[m-1] \) and output the result (if there is more than one output then we measure more variables).

### 22.8.3 Uniform computation

Just as in the classical case, we can define uniform computational models. For example, we can define the \( QNAND++ \) programming language to be \( QNAND \) augmented with loops and arrays just like \( NAND++ \) is obtained from \( NAND \). Using this we can define the class \( BQP \) which is the uniform analog of \( BQP/poly \). Just as in the classical setting it holds that \( BPP \subseteq P/poly \), in the quantum setting it can be shown that \( BQP \subseteq BQP/poly \). Just like the classical case, we can also use Quantum Turing Machines instead of \( QNAND++ \) to define \( BQP \).

Yet another way to define \( BQP \) is the following: a function \( \bar{F} : \{0,1\}^* \rightarrow \{0,1\} \) is in \( BQP \) if (1) \( F \in BQP/poly \) and (2) moreover for every \( n \), the quantum circuit that verifies this can be generated by a classical polynomial time \( NAND++ \) program (or, equivalently, a polynomial-time Turing machine).

We use this definition here, though an equivalent one can be made using \( QNAND++ \) or quantum Turing machines:

**Definition 22.5 — The class \( BQP \).** Let \( F : \{0,1\}^* \rightarrow \{0,1\} \). We say that \( F \in BQP \) if there exists a polynomial time \( NAND++ \) program \( P \) such that for every \( n \), \( P(1^n) \) is the description of a quantum circuit \( C_n \) that computes the restriction of \( F \) to \( \{0,1\}^n \).

One way to verify that you’ve understood these definitions is to see that you can prove (1) \( P \subseteq BQP \) and in fact the stronger statement \( BPP \subseteq BQP \), (2) \( BQP \subseteq EXP \), and (3) For every \( NP \)-complete function \( F \), if \( F \in BQP \) then \( NP \subseteq BQP \). Exercise 22.1 asks you to work these out.

The relation between \( NP \) and \( BQP \) is not known (see also Remark 22.5). It is widely believed that \( NP \not\subseteq BQP \), but there is no consensus whether or not \( BQP \subseteq NP \). It is quite possible that these two classes are incomparable, in the sense that \( NP \not\subseteq BQP \) (and in particular no \( NP \)-complete function belongs to \( BQP \)) but also \( BQP \not\subseteq NP \) (and there are some interesting candidates for such problems).
It can be shown that \textit{Q NAND VAL} (evaluating a quantum circuit on an input) is computable by a polynomial size QNAND program, and moreover this program can even be generated \textit{uniformly} and hence \textit{Q NAND VAL} is in \textbf{BQP}. This allows us to “port” many of the results of classical computational complexity into the quantum realm as well.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Superconducting_quantum_computer_prototype_at_Google.jpg}
\caption{Superconducting quantum computer prototype at Google. Image credit: Google / MIT Technology Review.}
\end{figure}

\subsection*{22.9 PHYSICALLY REALIZING QUANTUM COMPUTATION}

To realize quantum computation one needs to create a system with \( n \) independent binary states (i.e., “qubits”), and be able to manipulate small subsets of two or three of these qubits to change their state. While by the way we defined operations above it might seem that one needs to be able to perform arbitrary unitary operations on these two or three qubits, it turns out that there several choices for \textit{universal sets} - a small constant number of gates that generate all others. The biggest challenge is how to keep the system from being measured and \textit{collapsing} to a single classical combination of states. This is sometimes known as the \textit{coherence time} of the system. The \textit{threshold theorem} says that there is some absolute constant level of errors \( \tau \) so that if errors are created at every gate at rate smaller than \( \tau \) then we can recover from those and perform arbitrary long computations. (Of course there are different ways to model the errors and so there are actually several \textit{threshold theorems} corresponding to various noise models).
There have been several proposals to build quantum computers:

- **Superconducting quantum computers** use superconducting electric circuits to do quantum computation. This is the direction where there has been most recent progress towards “beating” classical computers.

- **Trapped ion quantum computers** use the states of an ion to simulate a qubit. People have made some recent advances on these computers too. While it’s not at all clear that’s the right measuring yard, the current best implementation of Shor’s algorithm (for factoring 15) is done using an ion-trap computer.

- **Topological quantum computers** use a different technology. Topological qubits are more stable by design and hence error correction is less of an issue, but constructing them is extremely challenging.

These approaches are not mutually exclusive and it could be that ultimately quantum computers are built by combining all of them together. In the near future, it seems that we will not be able to achieve full fledged large scale universal quantum computers, but rather more restricted machines, sometimes called “Noisy Intermediate-Scale Quantum Computers” or “NISQ”. See this article by John Preskill for some of the progress and applications of such more restricted devices.

### 22.10 SHOR’S ALGORITHM: HEARING THE SHAPE OF PRIME FACTORS

Bell’s Inequality is a powerful demonstration that there is something very strange going on with quantum mechanics. But could this “strangeness” be of any use to solve computational problems not directly related to quantum systems? A priori, one could guess the answer is *no*. In 1994 Peter Shor showed that one would be wrong:

**Theorem 22.6 — Shor’s Algorithm.** There is a polynomial-time quantum algorithm that on input an integer $M$ (represented in base two), outputs the prime factorization of $M$.

Another way to state Theorem 22.6 is that if we define

\[ \text{FACTORIZING} : \{0, 1\}^* \rightarrow \{0, 1\} \]

to be the function that on input a pair of numbers $(M, X)$ outputs 1 if and only if $M$ has a factor $P$ such that $2 \leq P \leq X$, then FACTORIZING is in $\text{BQP}$. This is an exponential improvement over the best known classical algorithms, which take roughly $2^{\tilde{O}(n^{1/3})}$ time, where the $\tilde{O}$ notation hides factors that are polylogarithmic in $n$. While we will not prove Theorem 22.6 in this chapter, we will sketch some of the ideas behind the proof.
22.10.1 Period finding

At the heart of Shor’s Theorem is an efficient quantum algorithm for finding periods of a given function. For example, a function $f : \mathbb{R} \to \mathbb{R}$ is periodic if there is some $h > 0$ such that $f(x + h) = f(x)$ for every $x$ (e.g., see Fig. 22.4).

Figure 22.4: Top: A periodic function. Bottom: An a-periodic function.

Musical notes yield one type of periodic function. When you pull on a string on a musical instrument, it vibrates in a repeating pattern. Hence, if we plot the speed of the string (and so also the speed of the air around it) as a function of time, it will correspond to some periodic function. The length of the period is known as the wave length of the note. The frequency is the number of times the function repeats itself within a unit of time. For example, the “Middle C” note has a frequency of 261.63 Hertz, which means its period is $1/(261.63)$ seconds.

If we play a chord by playing several notes at once, we get a more complex periodic function obtained by combining the functions of the individual notes (see Fig. 22.5). The human ear contains many small hairs, each of which is sensitive to a narrow band of frequencies. Hence when we hear the sound corresponding to a chord, the hairs in our ears actually separate it out to the components corresponding to each frequency.

It turns out that (essentially) every periodic function $f : \mathbb{R} \to \mathbb{R}$ can be decomposed into a sum of simple wave functions (namely functions of the form $x \mapsto \sin(\theta x)$ or $x \mapsto \cos(\theta x)$). This is known as the Fourier Transform (see Fig. 22.6). The Fourier transform makes it easy to compute the period of a given function: it will simply be the least common multiple of the periods of the constituent waves.
Figure 22.5: Left: The air-pressure when playing a “C Major” chord as a function of time. Right: The coefficients of the Fourier transform of the same function, we can see that it is the sum of three frequencies corresponding to the C, E and G notes (261.63, 329.63 and 392 Hertz respectively). Credit: Bjarke Mønsted’s Quora answer.

Figure 22.6: If $f$ is a periodic function then when we represent it in the Fourier transform, we expect the coefficients corresponding to wavelengths that do not evenly divide the period to be very small, as they would tend to “cancel out”.
22.10.2 Shor’s Algorithm: A bird’s eye view

On input an integer $M$, Shor’s algorithm outputs the prime factorization of $M$ in time that is polynomial in $\log M$. The main steps in the algorithm are the following:

**Step 1: Reduce to period finding.** The first step in the algorithm is to pick a random $A \in \{0, 1, \ldots, M - 1\}$ and define the function $F_A : \{0, 1\}^m \to \{0, 1\}^m$ as $F_A(x) = A^x \pmod{M}$ where we identify the string $x \in \{0, 1\}^m$ with an integer using the binary representation, and similarly represent the integer $A^x \pmod{M}$ as a string. (We will choose $m$ to be some polynomial in $M$ and so in particular $\{0, 1\}^m$ is a large enough set to represent all the numbers in $\{0, 1, \ldots, M - 1\}$).

Some not-too-hard (though somewhat technical) calculations show that: (1) The function $F_A$ is periodic (i.e., there is some integer $p_A$ such that $F_A(x + p_A) = F_A(x)$ for almost\(^{17}\) every $x$) and more importantly (2) If we can recover the period $p_A$ of $F_A$ for several randomly chosen $A$’s, then we can recover the factorization of $M$. Hence, factoring $M$ reduces to finding out the period of the function $F_A$. Exercise 22.2 asks you to work out this for the related task of computing the discrete logarithm (which underlies the security of the Diffie-Hellman key exchange and elliptic curve cryptography).

**Step 2: Period finding via the Quantum Fourier Transform.** Using a simple trick known as “repeated squaring”, it is possible to compute the map $x \mapsto F_A(x)$ in time polynomial in $m$, which means we can also compute this map using a polynomial number of NAND gates, and so in particular we can generate in polynomial quantum time a quantum state $\rho$ that is (up to normalization) equal to

\[
\sum_{x \in \{0, 1\}^m} |x\rangle |F_A(x)\rangle .
\]

(22.13)

In particular, if we were to measure the state $\rho$, we would get a random pair of the form $(x, y)$ where $y = F_A(x)$. So far, this is not at all impressive. After all, we did not need the power of quantum computing to generate such pairs: we could simply generate a random $x$ and then compute $F_A(x)$.

Another way to describe the state $\rho$ is that the coefficient of $|x\rangle |y\rangle$ in $\rho$ is proportional to $f_{A,y}(x)$ where $f_{A,y} : \{0, 1\}^m \to \mathbb{R}$ is the function such that

\[
f_{A,y}(x) = \begin{cases} 1 & y = A^x \pmod{M} \\ 0 & \text{otherwise} \end{cases} .
\]

(22.14)

The magic of Shor’s algorithm comes from a procedure known as the Quantum Fourier Transform. It allows to change the state $\rho$ into the state $\tilde{\rho}$ where the coefficient of $|x\rangle |y\rangle$ is now proportional to the $x$-th Fourier coefficient of $f_{A,y}$. In other words, if we measure the state $\tilde{\rho}$, we will obtain a pair $(x, y)$ such that the probability of choosing $x$
is proportional to the square of the weight of the frequency \( x \) in the representation of the function \( f_{A,y} \). Since for every \( y \), the function \( f_{A,y} \) has the period \( p_A \), it can be shown that the frequency \( x \) will be (almost\(^{18}\)) a multiple of \( p_A \). If we make several such samples \( y_0, \ldots, y_k \) and obtain the frequencies \( x_1, \ldots, x_k \), then the true period \( p_A \) divides all of them, and it can be shown that it is going to be in fact the greatest common divisor (g.c.d.) of all these frequencies: a value which can be computed in polynomial time.

As mentioned above, we can recover the factorization of \( M \) from the periods of \( F_{A_0}, \ldots, F_{A_t} \) for some randomly chosen \( A_0, \ldots, A_t \) in \( \{0, \ldots, M - 1\} \) and \( t \) which is polynomial in \( \log M \).

The resulting algorithm can be described in a high (and somewhat inaccurate) level as follows:

Shor’s Algorithm: (sketch)

**Input:** Number \( M \in \mathbb{N} \).

**Output:** Prime factorization of \( M \).

**Operations:**

1. Repeat the following \( k \) = \( \text{poly}(\log M) \) number of times:
   (a) Choose \( A \in \{0, \ldots, M - 1\} \) at random, and let \( f_A : \mathbb{Z}_M \to \mathbb{Z}_M \) be the map \( x \mapsto A^x \mod M \).
   (b) For \( t = \text{poly}(\log M) \), repeat \( t \) times the following step: Quantum Fourier Transform to create a quantum state \( \psi \) over \( \text{poly}(\log(M)) \) qubits, such that if we measure \( \psi \) we obtain a pair of strings \( (j, y) \) with probability proportional to the square of the coefficient corresponding to the wave function \( x \mapsto \cos(\pi y/M) \) or \( x \mapsto \sin(\pi y/M) \) in the Fourier transform of the function \( f_{A,y} : \mathbb{Z}_m \to \{0,1\} \) defined as \( f_{A,y}(x) = 1 \) iff \( f_A(x) = y \).
   (c) If \( j_1, \ldots, j_k \) are the coefficients we obtained in the previous step, then the least common multiple of \( M/j_1, \ldots, M/j_k \) is likely to be the period of the function \( f_A \).

2. If we let \( A_0, \ldots, A_{k-1} \) and \( p_0, \ldots, p_{k-1} \) be the numbers we chose in the previous step and the corresponding periods of the functions \( f_{A_0}, \ldots, f_{A_{k-1}} \) then we can use classical results in number theory to obtain from these a non-trivial prime factor \( Q \) of \( M \) (if such exists). We can now run the algorithm again with the (smaller) input \( M/Q \) to obtain all other factors.

---

\(^{18}\) The “almost” qualifier again appears because the original function was only “almost” periodic, but it turns out this can be handled by using an “approximate greatest common divisor” algorithm instead of a standard g.c.d. below. The latter can be obtained using a tool known as the continued fraction representation of a number.
Reducing factoring to order finding is cumbersome, but can be done in polynomial time using a classical computer. The key quantum ingredient in Shor’s algorithm is the quantum fourier transform.

Quantum Fourier Transform Despite its name, the Quantum Fourier Transform does not actually give a way to compute the Fourier Transform of a function \( f : \{0, 1\}^n \rightarrow \mathbb{R} \). This would be impossible to do in time polynomial in \( n \), as simply writing down the Fourier Transform would require \( 2^n \) coefficients. Rather the Quantum Fourier Transform gives a quantum state where the amplitude corresponding to an element (think: frequency) \( h \) is equal to the corresponding Fourier coefficient. This allows to sample from a distribution where \( h \) is drawn with probability proportional to the square of its Fourier coefficient. This is not the same as computing the Fourier transform, but is good enough for recovering the period.

22.11 QUANTUM FOURIER TRANSFORM (ADVANCED, OPTIONAL)

The above description of Shor’s algorithm skipped over the implementation of the main quantum ingredient: the Quantum Fourier Transform algorithm. In this section we discuss the ideas behind this algorithm. We will be rather brief and imprecise. Remark 22.4 and Section 22.13 contain references to sources of more information about this topic.

To understand the Quantum Fourier Transform, we need to better understand the Fourier Transform itself. In particular, we will need to understand how it applies not just to functions whose input is a real number but to functions whose domain can be any arbitrary commutative group. Therefore we now take a short detour to (very basic) group theory, and define the notion of periodic functions over groups.

Group theory While we define the concepts we use, some background in group or number theory might be quite helpful for fully understanding this section. We will not use anything more than the basic properties of finite Abelian groups. Specifically we use the following notions:

- A finite group \( G \) can be thought of as simply a set of elements and some binary operation \( * \) on these elements (i.e., if \( g, h \in G \) then \( g * h \) is an element of \( G \) as well).
- The operation \( * \) satisfies the sort of properties that
The Fourier transform is a deep and vast topic, on which we will barely touch upon here. Over the real numbers, the Fourier transform of a function \( f \) is obtained by expressing \( f \) in the form \( \sum \hat{f}(\alpha) \chi_\alpha \)

where the \( \chi_\alpha \)'s are “wave functions” (e.g. sines and cosines). However, it turns out that the same notion exists for every Abelian group \( G \). Specifically, for every such group \( G \), if \( f \) is a function mapping \( G \) to \( \mathbb{C} \), then we can write \( f \) as

\[
f = \sum_{g \in G} \hat{f}(g) \chi_g ,
\]

where the \( \chi_g \)'s are functions mapping \( G \) to \( \mathbb{C} \) that are analogs of the “wave functions” for the group \( G \) and for every \( g \in G \), \( \hat{f}(g) \) is a complex number known as the Fourier coefficient of \( f \) corresponding to \( g \).\(^{19}\) The representation Eq. (22.15) is known as the Fourier expansion or Fourier transform of \( f \), the numbers \( (\hat{f}(g))_{g \in G} \) are known as the Fourier coefficients of \( f \) and the functions \( (\chi_g)_{g \in G} \) are known as the Fourier characters. The central property of the Fourier characters is that they are homomorphisms of the group into the complex numbers, in the sense that for every \( x, x' \in G \), \( \chi_g(x \ast x') = \chi_g(x) \chi_g(x') \), where \( \ast \) is the group operation. One corollary of this property is that if \( \chi_g(h) = 1 \) then \( \chi_g \) is \( h \) periodic in the sense that \( \chi_g(x \ast h) = \chi_g(x) \) for every \( x \).

It turns out that if \( f \) is periodic with minimal period \( h \), then the only Fourier characters that have non zero coefficients in the expression Eq. (22.15) are those that are \( h \) periodic as well. This can be used to recover the period of \( f \) from its Fourier expansion.

### 22.11.1 Quantum Fourier Transform over the Boolean Cube: Simon’s Algorithm

We now describe the simplest setting of the Quantum Fourier Transform: the group \( \{0,1\}^n \) with the XOR operation, which we’ll denote by \( (\{0,1\}^n, \oplus) \). It can be shown that the Fourier transform over \( (\{0,1\}^n, \oplus) \) corresponds to expressing \( f : \{0,1\}^n \to \mathbb{C} \) as

\[
f = \sum_{y \in \{0,1\}^n} \hat{f}(y) \chi_y
\]

where \( \chi_y : \{0,1\}^n \to \mathbb{C} \) is defined as \( \chi_y(x) = (-1)^{\sum_i y_i x_i} \)

and

\[
\hat{f}(y) = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} f(x)(-1)^{\sum_i y_i x_i} ,
\]

\(^{19}\) The equation Eq. (22.15) means that if we think of \( f \) as a \(|G|\) dimensional vector over the complex numbers, then we can write this vector as a sum (with certain coefficients) of the vectors \( \{\chi_g\}_{g \in G} \).
The Quantum Fourier Transform over \(\{0,1\}^n, \oplus\) is actually quite simple:

**Theorem 22.7 — QFT Over the Boolean Cube.** Let \(\rho = \sum_{x \in \{0,1\}^n} f(x)|x\rangle\) be a quantum state where \(f : \{0,1\}^n \rightarrow \mathbb{C}\) is some function satisfying \(\sum_{x \in \{0,1\}^n} |f(x)|^2 = 1\). Then we can use \(n\) gates to transform \(\rho\) to the state

\[
\sum_{y \in \{0,1\}^n} \hat{f}(y)|y\rangle
\]

(22.17)

where \(f = \sum_y \hat{f}(y)\chi_y\) and \(\chi_y : \{0,1\}^n \rightarrow \mathbb{C}\) is the function \(\chi_y(x) = -1^{\sum x_i y_i}\).

**Proof Idea:** The idea behind the proof is that the Hadamard operation corresponds to the Fourier transform over the group \(\{0,1\}^n\) (with the XOR operations). To show this, we just need to do the calculations. 

**Proof of Theorem 22.7.** We can express the Hadamard operation \(HAD\) as follows:

\[
HAD|a\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^a|1\rangle).
\]

(22.18)

We are given the state

\[
\rho = \sum_{x \in \{0,1\}^n} f(x)|x\rangle.
\]

(22.19)

Now suppose that we apply the \(HAD\) operation to each of the \(n\) qubits. We can see that we get the state

\[
2^{-n/2} \sum_{x \in \{0,1\}^n} f(x) \prod_{i=0}^{n-1} (|0\rangle + (-1)^{x_i}|1\rangle).
\]

(22.20)

We can now use the distributive law and open up a term of the form

\[
f(x) (|0\rangle + (-1)^{x_0}|1\rangle) \cdots (|0\rangle + (-1)^{x_{n-1}}|1\rangle)
\]

(22.21)

to the following sum over \(2^n\) terms:\n
\[
f(x) \sum_{y \in \{0,1\}^n} (-1)^{\sum y_i x_i} |y\rangle.
\]

(22.22)

But by changing the order of summations, we see that the final state is

\[
\sum_{y \in \{0,1\}^n} 2^{-n/2} \sum_{x \in \{0,1\}^n} f(x) (-1)^{\sum x_i y_i} |y\rangle
\]

(22.23)

which exactly corresponds to \(\hat{\rho}\). 

\[\blacksquare\]
22.11.2 From Fourier to Period finding: Simon’s Algorithm (advanced, optional)

Using Theorem 22.7 it is not hard to get an algorithm that can recover a string \( h^* \in \{0, 1\}^n \) given a circuit that computes a function \( F : \{0, 1\}^n \rightarrow \{0, 1\}^* \) that is \( h^* \) periodic in the sense that \( F(x) = F(x') \) for distinct \( x, x' \) if and only if \( x' = x \oplus h^* \). The key observation is that if we compute the state \( \sum_{x \in \{0, 1\}^n} |x\rangle |F(x)\rangle \), and perform the Quantum Fourier transform on the first \( n \) qubits, then we would get a state such that the only basis elements with nonzero coefficients would be of the form \( |y\rangle \) where

\[
\sum y_i h_i^* = 0 \pmod{2} \tag{22.24}
\]

So, by measuring the state, we can obtain a sample of a random \( y \) satisfying Eq. (22.24). But since Eq. (22.24) is a linear equation modulo 2 about the unknown \( n \) variables \( h_0^*, \ldots, h_{n-1}^* \), if we repeat this procedure to get \( n \) such equations, we will have at least as many equations as variables and (it can be shown that) this will suffice to recover \( h^* \).

This result is known as Simon’s Algorithm, and it preceded and inspired Shor’s algorithm.

22.11.3 From Simon to Shor (advanced, optional)

Theorem 22.7 seemed to really use the special bit-wise structure of the group \( \{0, 1\}^n \), and so one could wonder if it can be extended to other groups. However, it turns out that we can in fact achieve such a generalization.

The key step in Shor’s algorithm is to implement the Fourier transform for the group \( \mathbb{Z}_L \) which is the set of numbers \( \{0, \ldots, L-1\} \) with the operation being addition modulo \( L \). In this case it turns out that the Fourier characters are the functions \( \chi_y(x) = \omega^{yx} \) where \( \omega = e^{2\pi i/L} \) (\( i \) here denotes the complex number \( \sqrt{-1} \)). The \( y \)-th Fourier coefficient of a function \( f : \mathbb{Z}_L \rightarrow \mathbb{C} \) is

\[
\hat{f}(y) = \frac{1}{\sqrt{L}} \sum_{x \in \mathbb{Z}_L} f(x) \omega^{xy}. \tag{22.25}
\]

The key to implementing the Quantum Fourier Transform for such groups is to use the same recursive equations that enable the classical Fast Fourier Transform (FFT) algorithm. Specifically, consider the case that \( L = 2^t \). We can separate the sum over \( x \) in Eq. (22.25) to the terms corresponding to even \( x \)’s (of the form \( x = 2z \)) and odd \( x \)’s (of the form \( x = 2z + 1 \)) too obtain

\[
\hat{f}(y) = \frac{1}{\sqrt{L}} \sum_{z \in \mathbb{Z}_{L/2}} f(2z)(\omega^2)^{yz} + \frac{1}{\sqrt{L}} \sum_{z \in \mathbb{Z}_{L/2}} f(2z + 1)(\omega^2)^{yz}\omega^y \tag{22.26}
\]
which reduces computing the Fourier transform of \( f \) over the group \( \mathbb{Z}_{2^\ell} \) to computing the Fourier transform of the functions \( f_{\text{even}} \) and \( f_{\text{odd}} \) (corresponding to the applying \( f \) to only the even and odd vectors respectively) which have \( 2^{\ell-1} \) inputs that we can identify with the group \( \mathbb{Z}_{2^{\ell-1}} \).

This observation is usually used to obtain a fast (e.g. \( O(L \log L) \)) time to compute the Fourier transform in a classical setting, but it can be used to obtain a quantum circuit of \( \text{poly}(\log L) \) gates to transform a state of the form \( \sum_{x \in \mathbb{Z}_L} f(x) |x\rangle \) to a state of the form \( \sum_{y \in \mathbb{Z}_L} \hat{f}(y) |y\rangle \).

The case that \( L \) is not an exact power of two causes some complications in both the classical case of the Fast Fourier Transform and the quantum setting of Shor’s algorithm. However, it is possible to handle these. The idea is that we can embed \( \mathbb{Z}_L \) in the group \( \mathbb{Z}_{A \cdot L} \) for any integer \( A \), and we can find an integer \( A \) such that \( A \cdot L \) will be close enough to a power of \( 2 \) (i.e., a number of the form \( 2^m \) for some \( m \)), so that if we do the Fourier transform over the group \( \mathbb{Z}_{2^m} \) then we will not introduce too many errors.

**Lecture Recap**

- The state of an \( n \)-qubit quantum system can be modeled as a \( 2^n \) dimensional vector.
- An operation on the state corresponds to applying a unitary matrix to this vector.
- Quantum circuits are obtained by composing basic operations such as \( H, A, D \), and \( U_{\text{NAND}} \).
- We can use quantum circuits to define the classes \( \text{BQP}_{/\text{poly}} \) and \( \text{BQP} \) which are the quantum analogs of \( \text{P}_{/\text{poly}} \) and \( \text{BPP} \) respectively.
- There are some problems for which the best known quantum algorithm is *exponentially faster* than the best known, but quantum computing is not a panacea. In particular, as far as we know, quantum computers could still require exponential time to solve \( \text{NP} \)-complete problems such as \( \text{SAT} \).

**22.12 EXERCISES**

**Disclaimer** Most of the exercises have been written in the summer of 2018 and haven’t yet been fully debugged. While I would prefer people do not post online solutions to the exercises, I would greatly appreciate if you let me know of any bugs. You can do so by posting a [GitHub issue](#) about the exercise, and optionally complement this with an email to me.
Exercise 22.1 — Quantum and classical complexity class relations. Prove the following relations between quantum complexity classes and classical ones:

1. \( \mathbf{P}/\mathbf{poly} \subseteq \mathbf{BQP}/\mathbf{poly} \)
2. \( \mathbf{P} \subseteq \mathbf{BQP} \)
3. \( \mathbf{BPP} \subseteq \mathbf{BQP} \)
4. \( \mathbf{BQP} \subseteq \mathbf{EXP} \)
5. If \( \text{SAT} \in \mathbf{BQP} \) then \( \mathbf{NP} \subseteq \mathbf{BQP} \)

Exercise 22.2 — Discrete logarithm from order finding. Show a probabilistic polynomial time classical algorithm that given an Abelian finite group \( \mathcal{G} \) (in the form of an algorithm that computes the group operation), a generator \( g \) for the group, and an element \( h \in \mathcal{G} \), as well access to a black box that on input \( f \in \mathcal{G} \) outputs the order of \( f \) (the smallest \( a \) such that \( f^a = 1 \)), computes the discrete logarithm of \( h \) with respect to \( g \). That is the algorithm should output a number \( x \) such that \( g^x = h \).

See footnote for hint.

22.13 BIBLIOGRAPHICAL NOTES

Chapters 9 and 10 in the book *Quantum Computing Since Democritus* give an informal but highly informative introduction to the topics of this lecture and much more. Shor’s and Simon’s algorithms are also covered in Chapter 10 of my book with Arora on computational complexity.

There are many excellent videos available online covering some of these materials. The Fourier transform is covered in this videos of Dr. Chris Geoscience, Clare Zhang and Vi Hart. More specifically to quantum computing, the videos of Umesh Vazirani on the Quantum Fourier Transform and Kelsey Houston-Edwards on Shor’s Algorithm are very recommended.

Chapter 10 in Avi Wigderson’s book gives a high level overview of quantum computing. Andrew Childs’ lecture notes on quantum algorithms, as well as the lecture notes of Umesh Vazirani, John Preskill, and John Watrous

Regarding quantum mechanics in general, this video illustrates the double slit experiment, this Scientific American video is a nice exposition of Bell’s Theorem. This talk and panel moderated by Brian

---

21 Hint: You can use \( U_{\text{NAND}} \) to simulate NAND gates.
22 Hint: Use the alternative characterization of \( \mathbf{P} \) as in Exercise 12.6.
23 Hint: You can use the \( HAD \) gate to simulate a coin toss.
24 Hint: In exponential time simulating quantum computation boils down to matrix multiplication.
25 Hint: If a reduction can be implemented in \( \mathbf{P} \) it can be implemented in \( \mathbf{BQP} \) as well.

26 We are given \( h = g^x \) and need to recover \( x \). To do so we can compute the order of various elements of the form \( h^c g^b \). The order of such an element is a number \( c \) satisfying \( c(xa + b) = 0 \) (mod \( \mathcal{G} \)). With a few random examples we will get a non trivial equation on \( x \) (where \( c \) is not zero modulo \( \mathcal{G} \)) and then we can use our knowledge of \( a, b, c \) to recover \( x \).
Greene discusses some of the philosophical and technical issues around quantum mechanics and its so-called “measurement problem”. The Feynmann lecture on the Fourier Transform and quantum mechanics in general are very much worth reading.

The Fast Fourier Transform, used as a component in Shor’s algorithm, is one of the most useful algorithms across many applications areas. The stories of its discovery by Gauss in trying to calculate asteroid orbits and rediscovery by Tukey during the cold war are fascinating as well.

22.14 FURTHER EXPLORATIONS

Some topics related to this chapter that might be accessible to advanced students include: (to be completed)

22.15 ACKNOWLEDGEMENTS

Thanks to Scott Aaronson for many helpful comments about this chapter.
VI
APPENDICES
A

The NAND Programming Language
The NAND Programming Language

Version: 0.2

The NAND programming language was designed to accompany the upcoming book "Introduction to Theoretical Computer Science". This is an appendix to this book, which is also available online as a Jupyter notebook in the boazbk/nandnotebooks on Github. You can also try the live binder version.

The NAND programming language is part of a family of languages:

- **NAND**: The NAND programming language is equivalent in expressive power to *Boolean circuits*. Specifically NAND programs are straightline programs with the NAND operation, which are equivalent to Boolean circuits with NAND gates.
- **NAND++**: The NAND++ programming language is equivalent in expressive power to *Turing Machines*. NAND++ programs are obtained by adding loops and unbounded arrays to NAND programs. Specifically, NAND++ programs can be thought of as capturing oblivious single tape Turing Machines, that are polynomially equivalent to all other standard variants of Turing Machines and Turing-equivalent models of computation.
- **NAND<<**: The NAND<< programming language is a formalization of *RAM Machines*. NAND<< programs are obtained by adding integer-valued variables and arrays, and the standard "C type" operations on them, as well as indirect indexing of arrays via integer variables. NAND<< programs are equivalent up to polylogarithmic terms to standard models of RAM machines, and up to polynomial terms with NAND++ and all other standard Turing-equivalent models.
- **QNAND**: QNAND is only used in a single chapter of the book, and is meant to capture Quantum Boolean Circuits and so can be used to define the class BQP of polynomial time quantum computation.

Syntax of NAND programs

This notebook/appendix is concerned with the first and simplest of these languages: the *NAND Programming Language*, which appears in Chapter 3: "Defining Computation".

A NAND program consists of a sequence of lines, each of the following form:

```nand
foo = NAND(bar,blah)
```

where `foo`, `bar` and `blah` are variable identifiers.

We have two special types of variables: *input variables* have the form `X[i]` where `i` is a natural number, and *output variables* have the form `Y[j]` where `j` is a natural number. When a NAND program is executed on input $x \in \{0,1\}^n$, the variable $X[i]$ is assigned the value $x_i$ for all $i$ in $[n]$. The *output* of the program is the list of $m$ values $Y[j]$ for all $j$ in $[m]$. The *output* of the program is the list of $m$ values.
\[ Y[0] \ldots Y[m-1] \], where \( m-1 \) is the largest index for which the variable \( Y[m-1] \) is assigned a value in the program.

Here is an example of a NAND program:

```
In [1]:
xor = r''
   u = NAND(X[0],X[1])
   v = NAND(X[0],u)
   w = NAND(X[1],u)
   Y[0] = NAND(v,w)
'''
```

To evaluate this program on inputs \( x_0, x_1 \in \{0,1\} \), we can use a simple Python function that would keep track of the values assigned to variables.

```
In [2]:
def NAND(x,y):
    """Compute the NAND of two 0/1 valued variables."""
    return 1 - x*y

In [3]:
import re

def EVAL(prog,x):
    """Evaluate NAND program prog on input x."""
    (n,m) = numinout(prog) # utility function to get num of inputs/outputs
    vartable = {} # dictionary for variables

    for i in range(n): vartable[f'X[{i}]'] = x[i] # assign x[i] to variable "X[i]"

    def split_to_components_of_a_line:
        foo,bar,blah, = filter(None,re.split(r'\s+=\s*NAND\s*|\s*|\s+'),line))
    vartable[foo] = NAND(vartable[bar],vartable[blah])

    return [vartable[f'Y[{j}]'] for j in range(m)]
```

```
In [4]:
# IMPORTANT DISCLAIMER:
# The code in this and other notebooks should not be considered as an example of good coding practice.
# I am not a coder by any means.
# Also, whenever faced with a choice, I picked simplicity of code or brevity over robustness.
# It is quite possible that running these functions with malformed inputs would trigger the Zombie apocalypse.
```

```
In [5]:
# utility function
def numinout(prog):
    """Compute the number of inputs and outputs of a NAND program, given a string of source code.""
    n = max([int(s[2:-1]) for s in re.findall(r'X\[[\d+\]'],prog)])+1
    m = max([int(s[2:-1]) for s in re.findall(r'Y\[[\d+\]'],prog)])+1
    return
```
Specification of NAND Programming language

We now describe a NAND program a little more formally. Every line in a NAND program has the form

\[ \text{id1} = \text{NAND} \left( \text{id2} , \text{id3} \right) \]

where \text{id1}, \text{id2}, \text{id3} are variable identifier. A variable identifier is any sequence of letters, digits, and underscore and square bracket characters \_\_\[\_\_\].

Variables whose identifiers have the form \text{X}[\text{i}] where \text{i} is a sequence of digits are called input variables.

Variables whose identifiers have the form \text{Y}[\text{j}] where \text{j} is a sequence of digits are called output variables.

All variables in NAND are Boolean: the can only get the values $0$ and $1$. If a variable is accessed before it is assigned a value then we assume the value defaults to zero (however such accesses are not legal in standard NAND programs, see below).

Execution of a NAND program

The number of inputs of a NAND program $P$ is one plus the largest number $i$ such that a variable of the form \text{X}[\text{i}] appears in $P$. The number of outputs of $P$ is one plus the largest number $j$ such that a variable of the form \text{Y}[\text{j}] appears in $P$.

If $P$ is a program with $n$ inputs and $m$ outputs, then the _execution of $P$ on input $x \in \{0,1\}^n$ is defined as follows:

1. We assign to the variable \text{X}[\text{i}] the value $x_i$ for all $i \in [n]$.
2. We execute line by line the program, where for each line of the form $\text{id1} = \text{NAND} \left( \text{id2} , \text{id3} \right)$ we assign to the variable identified by $\text{id1}$ the NAND of the values of the variables identified by $\text{id2}$ and $\text{id3}$.
3. We output the $m$ values assigned to the variables \text{Y}[\text{j} \ldots \text{Y}[\text{m-1}]]

Standard or "well formed" NAND programs

We define the following as the "standard form" of a NAND program:
While a priori variables can be any sequence of letters, digits, underscores and brackets, we will assume our identifiers have one of the following two forms:

- A sequence of lower case letters, numbers, and underscores, that contain no upper case letters or brackets. For example, `foo, bar_blah, hello_world, boaz1234`.
- An identifier starting with a single capital letter, optionally followed by lower case letters, numbers, and underscores, and ending in `[ $i$ ]` where `$i$` is a sequence of digits with no leading zeroes. For example `X[17]` or `Foo[22]` or `Bar_blah[103]`.

We will assume no variable is used before it is assigned a value. The output variables are never read from (i.e., never on the lefthand side of an assignment) and the input variables are never written to (i.e., never on the righthand side of an assignment).

All output variables are written to, and all input variables are read from. That is, if a variable of the form `X[ $i$ ]` appears in the program then `X[ $i'$ ]` should also appear for all $0 \leq i' < i$. If a variable of the form `Y[ $j$ ]` appears in the program, then `Y[ $j'$ ]` should also appear for all $0 \leq j' < j$.

By default, every NAND program is assumed to be in standard form. Note that it is very easy to transform a NAND program into standard form, with a small cost in the number of lines. Specifically, we can ensure standard form by doing the following:

1. Modify variable names to fit out convention.
2. Add two lines to define a zero variable that is identically zero, and then use that in place of variables that haven't been assigned a value.
3. Add at most $n+m$ lines to touch all input and output variables.

For simplicity, we will often describe our results only for standard form NAND programs (for example the function `EVAL` above would throw an Exception if a variable is accessed before it's assigned a value), but they are of course easy to extend to general NAND programs through the transformation above.

**NAND interpreter via Python**

While we could easily write a "NAND interpreter" in any programming language, by design a NAND program uses valid Python code. So we can also write NAND programs as python functions:

```
In [7]: def XOR(a,b):
    t1 = NAND(a,b)
    t2 = NAND(a,t1)
    t3 = NAND(b,t1)
    return NAND(t2,t3)

XOR(0,1)
```

```
Out[7]: 1
```

We can translate such a function into standard NAND code using a simple trick. The idea is
that we evaluate the program "symbolically" giving it as inputs the pair of strings \(X[0]\) and \(X[1]\) instead of integers.

In [112]:
```
try:
    XOR("X[0]","X[1"]
except Exception as e:
    print(e)
```
can't multiply sequence by non-int of type 'str'

Of course we get a type error since NAND expects integers. When we need to do is to "override" the NAND function to take a pair of strings bar, blah instead of integers as input and generate the line of code \(foo = NAND(bar,blah)\) instead of computing the NAND function.

For example, the following almost works (except the output variable is not of the right form \(Y[\$\$]\))

In [9]:
```
count = 0
# Redfine NAND to take strings as input rather than integers
def NAND(bar,blah):
    global count
    foo = f"Temp[{count}]"
    count +=1
    print(f"\{foo} = NAND({bar},{blah})")
    return foo
XOR("X[0]","X[1"]
```

\[
\begin{align*}
\text{Temp}[0] &= \text{NAND}(X[0],X[1]) \\
\text{Temp}[1] &= \text{NAND}(X[0],\text{Temp}[0]) \\
\text{Temp}[2] &= \text{NAND}(X[1],\text{Temp}[0]) \\
\text{Temp}[3] &= \text{NAND}(\text{Temp}[1],\text{Temp}[2])
\end{align*}
\]

Out[9]: 'Temp[3]'  

Let us fix the definition of NAND back to its right form, and then give a more robust version of a program that maps a Python function that implements a NAND program into the code/

In [10]:
```
def NAND(a,b): return 1-a*b
```

In [11]:
```
def nandcode(f):
    
    Extract the NAND code of a given function f.
    Works by temporarily modifying NAND to take strings instead of integers,
    and generate on imput bar,blah a temporary variable identifier foo and the line
    'foo = NAND(bar,blah)'
    
    n = numarguments(f)
    
```

Out[11]:
```
counter = 0  # to ensure unique temporary variables.
code = ''

def tempNAND(bar,blah):
    nonlocal code, counter
    foo = f'Temp[{counter}]'
    counter += 1
    code += f'\{foo\} = NAND({bar},{blah})\n'
    return foo

outputs = runwith(lambda: f(*[f'X[{i}]' for i in range(n)]), "NAND", tempNAND)
    # execute f on the strings "X[0]", "X[1]", ... with NAND reassigned to tempNAND

    if type(outputs) == str: outputs = [outputs]  # make single output into singleton list

    for j in range(len(outputs)):
        code = code.replace(outputs[j], f'Y[{j}]')
    return code

In [12]:  # Some utility functions
    # (can ignore on first read, uses some Python trickery that's not so important for us)

    from inspect import signature
    def numarguments(f):
        """Number of arguments a Python function takes."""
        return len(signature(f).parameters)
    def runwith(f,*args):
        """Evaluate function f binding name to func""
        g = globals()
        old = {}
        new = {}
        for i in range(0,len(args),2):
            old[args[i]] = g[args[i]]
            new[args[i]] = args[i+1]
            # a little bit of an ugly hack
            # if you know a nicer way, please let me know
        try:
            for name in new: g[name] = new[name]
            res = f()
        finally:
            for name in old: g[name] = old[name]
        return res

In [13]: print(nandcode(XOR))

Temp[0] = NAND(X[0],X[1])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(X[1],Temp[0])
Y[0] = NAND(Temp[1],Temp[2])
**Syntactic sugar**

NAND is pretty bare bones, and writing NAND code directly gets real old real fast. However, we can use "syntactic sugar" to make it a little less tedious. For example we can use function definitions to avoid copying again and again repetitive code.

We will use the Python-like syntax of `def func(...) :` to define functions and so we can write the XOR on 4 bits function as follows:

```python
In [15]: def XOR4(a,b,c,d):
    return XOR(XOR(a,b),XOR(c,d))
XOR4(0,1,0,1)
Out[15]: 0
```

To verify that this is indeed merely "syntactic sugar" and this can be translated to pure NAND we can use our `nandcode` function:

```python
In [16]: print(nandcode(XOR4))
```

Once we have this, we can also define other functions such as AND, OR, NOT, and also the IF function that on input `cond`, `a` and `b` returns `a` if `cond` equals 1 and `b` otherwise.

```python
In [17]: def NOT(a):
    return NAND(a,a)

def AND(a,b):
    return NOT(NAND(a,b))

def OR(a,b):
    return NAND(NOT(a),NOT(b))

def IF(cond,first,sec):
    T = AND(cond,first)
    I = AND(1,first)
    E = NAND(T,sec)
    return NAND(E,I)
```
temp1 = AND(cond,first) # zero if cond=0, otherwise first
temp2 = AND(NOT(cond),sec) # zero if cond=1, otherwise second
return OR(temp1,temp2)

def one(a):
    return NAND(a,NOT(a))

def zero(a):
    return NOT(one(a))

def COPY(a):
    return NOT(NOT(a))

IF(0,1,0)

Out[17]: 0

We can use more python-inspired syntactic sugar:

In [18]: def increment(X):
    # increment integer given in binary representation
    n = len(X)
    Y = ["*"]*(n+1) # will be overwritten anyway
    carry = one(X[0])
    for i in range(n):
        Y[i] = XOR(X[i],carry)
        carry = AND(X[i],carry)
    Y[n] = COPY(carry)
    return Y

In [19]: inc5(1,1,0,0,0)

Out[19]: [0, 0, 1, 0, 0, 0]

In [20]: print(nandcode(inc5))

Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(X[0],Temp[1])
Temp[3] = NAND(X[0],Temp[2])
Temp[4] = NAND(Temp[1],Temp[2])
Y[0] = NAND(Temp[3],Temp[4])
Temp[6] = NAND(X[0],Temp[1])
Temp[7] = NAND(Temp[6],Temp[6])
Temp[8] = NAND(X[1],Temp[7])
Temp[9] = NAND(X[1],Temp[8])
Temp[10] = NAND(Temp[7],Temp[8])
Y[1] = NAND(Temp[9],Temp[10])
Temp[12] = NAND(X[1],Temp[7])
Temp[13] = NAND(Temp[12],Temp[12])
Temp[14] = NAND(X[2],Temp[13])
Temp[15] = NAND(X[2],Temp[14])
Temp[16] = NAND(Temp[13],Temp[14])
Y[2] = NAND(Temp[15],Temp[16])
We can create functions such as `inc5` for every `n` via a little Python trickery:

```python
In [21]: EVAL(nandcode(inc5),[1,0,1,0,0])
```
```
Out[21]: [0, 1, 1, 0, 0, 0]
```

```python
In [22]: def restrict(f,n):
   ...:     '''Create function that restricts the function f to exactly n input s'''
   ...:     args = ', '.join(f'arg{i}' for i in range(n))
   ...:     code = rf'''...
   ...:     def _temp({args}):
   ...:         return f({{args}})
   ...:     l = dict(locals())
   ...:     exec(code,l)
   ...:     return l['_temp']
```

```python
In [23]: inc7 = restrict(increment,7)
```
```
increment(1,1,1,0,0,0,1)
```
```
Out[23]: [0, 0, 0, 1, 0, 0, 1, 0]
```
```
in [24]: print(nandcode(inc7))
```
```
Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(X[0],Temp[1])
Temp[3] = NAND(X[0],Temp[2])
Temp[4] = NAND(Temp[1],Temp[2])
Y[0] = NAND(Temp[3],Temp[4])
Temp[6] = NAND(X[0],Temp[1])
Temp[7] = NAND(Temp[6],Temp[6])
Temp[8] = NAND(X[1],Temp[7])
Temp[9] = NAND(X[1],Temp[8])
Temp[10] = NAND(Temp[7],Temp[8])
Y[1] = NAND(Temp[9],Temp[10])
```
```
NAND Programs and circuits

NAND programs are equivalent to the model of Boolean circuits. We can present the graph corresponding to a NAND function using ideas similar to those we used to print the code. Below we have some python code to draw circuits; as usual you can skip over it in a first reading.

```python
Temp[1] = NAND(X[1],Temp[0])
Temp[2] = NAND(Temp[1],Temp[0])
Temp[3] = NAND(X[2],Temp[1])
Temp[4] = NAND(Temp[3],Temp[1])
Temp[5] = NAND(Temp[4],Temp[3])
Temp[6] = NAND(X[3],Temp[5])
Temp[7] = NAND(Temp[6],Temp[5])
Temp[8] = NAND(X[4],Temp[7])
Temp[9] = NAND(Temp[8],Temp[7])
Temp[10] = NAND(Temp[9],Temp[8])
Temp[11] = NAND(X[5],Temp[10])
Temp[12] = NAND(Temp[11],Temp[10])
Temp[13] = NAND(Temp[12],Temp[12])
Temp[14] = NAND(X[2],Temp[13])
Temp[15] = NAND(X[2],Temp[14])
Temp[16] = NAND(Temp[13],Temp[14])
Y[2] = NAND(Temp[15],Temp[16])
Temp[18] = NAND(X[2],Temp[13])
Temp[19] = NAND(Temp[18],Temp[18])
Temp[20] = NAND(X[3],Temp[19])
Temp[21] = NAND(X[3],Temp[20])
Temp[22] = NAND(Temp[19],Temp[20])
Y[3] = NAND(Temp[21],Temp[22])
Temp[24] = NAND(X[3],Temp[19])
Temp[25] = NAND(Temp[24],Temp[24])
Temp[26] = NAND(X[4],Temp[25])
Temp[27] = NAND(X[4],Temp[26])
Temp[28] = NAND(Temp[25],Temp[26])
Y[4] = NAND(Temp[27],Temp[28])
Temp[30] = NAND(X[4],Temp[25])
Temp[31] = NAND(Temp[30],Temp[30])
Temp[32] = NAND(X[5],Temp[31])
Temp[33] = NAND(X[5],Temp[32])
Temp[34] = NAND(Temp[31],Temp[32])
Y[5] = NAND(Temp[33],Temp[34])
Temp[36] = NAND(X[5],Temp[31])
Temp[37] = NAND(Temp[36],Temp[36])
Temp[38] = NAND(X[6],Temp[37])
Temp[39] = NAND(X[6],Temp[38])
Temp[40] = NAND(Temp[37],Temp[38])
Y[6] = NAND(Temp[39],Temp[40])
Temp[42] = NAND(X[6],Temp[37])
Temp[43] = NAND(Temp[42],Temp[42])
Temp[44] = NAND(Temp[43],Temp[43])
Y[7] = NAND(Temp[44],Temp[44])
```

In [25]: %matplotlib inline
   `%config InlineBackend.figure_format = 'svg`

In [26]: # Below we have two different implementations, that use two different libraries to draw circuits
   # Schemdraw has nicer gates, but doesn't do automatic layout of graphs
   # Graphviz has automatic layout but I couldn't figure out a way to get the gates to look as nice
   # Hopefully eventually I'll find the optimal way - help is welcome!
In [27]: # Use schemdraw to visualize circuits
    import SchemDraw as schem
    import SchemDraw.elements as e
    import SchemDraw.logic as l

    def disp(self):
        self.draw()

    schem.Drawing._ipython_display_ = disp

In [28]: # Schemdraw version
    def sdnandcircuit(f):
        
        n = numarguments(f)
        counter = 0 # to ensure unique temporary variables.
        G = schem.Drawing(unit=.5, fontsize=8)
        curx, cury = 0, 0

        def incr(jump = False):
            nonlocal curx, cury, n;
            UX = 2.5
            UY = 3.5
            if not jump and (cury>UY*(n-1)):
                cury -= UY
            else:
                cury = 0
                curx = (curx // UX)*UX + UX

            if (cury): curx += abs(cury)*UX/(2*n*UY)

        nodes = {}

        def tempNAND(bar, blah):
            nonlocal G, counter, curx, cury
            var = f'Temp[{counter}]'
            counter += 1
            g = G.add(l.NAND2(xy=[curx,cury], d="right"), label=var)
            incr()
            nodes[var] = g
            i1 = nodes[bar]
            in1 = i1.out if "out" in dir(i1) else i1.end
            i2 = nodes[blah]
            in2 = i2.out if "out" in dir(i2) else i2.end
            G.add(e.LINE, xy=in1, to=g.in1)
            G.add(e.LINE, xy=in2, to=g.in2)
            return curx

        for i in range(n):
            nodes[f'X[{i}]'] = G.add(e.DOT, xy = [curx,cury], lftlabel=f'X[{i}]')
            incr()

    outputs = runwith(lambda: f(*[f'X[{i}]' for i in range(n)]), 'NAND', tempNAND)
if type(outputs) == str: outputs = [outputs] # make single output into singleton list

incr(True)
for j in range(len(outputs)):
g = nodes[outputs[j]]
o = G.add(e.DOT, xy=[curx, cury], rgtlabel=f'Y[\{j}\]')
G.add(e.LINE, xy=g.out, to=o.start)
incr()

return G

In [29]: # Use Graphviz to visualize circuits
import graphviz
from graphviz import Graph
from graphviz import Digraph

In [30]: # Graphviz version
def gvnandcircuit(f):
   """Compute the graph representing a NAND circuit for a NAND program, given as a Python function.""
   n = numarguments(f)
counter = 0 # to ensure unique temporary variables.
G = Digraph(graph_attr={"rankdir":"LR"}) # schem.Drawing(unit=.5)

def tempNAND(bar,blah):
   nonlocal G, counter
   var = f'Temp[{counter}]'
counter += 1
   G.node(var, label="∧\text{\textdeg05}", shape='inhouse', orientation='90')
   G.edge(bar, var)
   G.edge(blah, var)
   return var

for i in range(n):
   G.node(f'X[{i}]', label=f'X[{i}]', fontcolor='blue', shape='circle')

outputs = runwith(lambda: f(*[f'X[{i}]' for i in range(n)]), 'NAND', tempNAND)

if type(outputs) == str: outputs = [outputs] # make single output into singleton list

for j in range(len(outputs)):
   G.node(outputs[j], label=f'Y[{j}]', fontcolor='red', shape='diamond')

return G

In [31]: def nandcircuit(f, method="Graphviz"):
   return gvnandcircuit(f) if method=="Graphviz" else sdnandcircuit(f)

We can now use these functions to draw the circuit corresponding to a NAND function:

In [32]: nandcircuit(XOR,"Schemdraw")
Computing every function

It turns out that we can compute every function $f: \{0,1\}^n \rightarrow \{0,1\}$ by some NAND program.

The crucial element for that is the function LOOKUP that on input an index $i \in [n]$ (represented as a string of length $\lfloor \log n \rfloor$) and a table $T \in \{0,1\}^n$, outputs $t_i$.
return IF([1[1-1]],LOOKUP([1[[1-1]],1[1-1]],LOOKUP([1[[1-1]],1[1-1]]))
LOOKUP([0,1,1,0,1,1,0,1,1,1,1,1])

Out[35]: 1

In [36]: # A more efficient IF .. not strictly necessary
def IF(cond,a,b):
    notcond = NAND(cond,cond)
    temp = NAND(b,notcond)
    temp1 = NAND(a,cond)
    return NAND(temp,temp1)

# Let's check that it works
[f"{(a,b,c)}:IF(a,b,c)" for a in [0,1] for b in [0,1] for c in [0,1]]

Out[36]: ['(0, 0, 0):0',
      '(0, 0, 1):1',
      '(0, 1, 0):0',
      '(0, 1, 1):1',
      '(1, 0, 0):0',
      '(1, 0, 1):0',
      '(1, 1, 0):1',
      '(1, 1, 1):1']

We can extract the NAND code of LOOKUP using the usual tricks.

In [37]: # generalize restrict to handle functions that take more than one array
def restrict(f,*numinputs):
    """Create function that restricts the function f to exactly given input length n0,n1,...""
    k = len(numinputs)
    args = []
    t = 0
    for i in range(k):
        if numinputs[i]: args = args + ["",".join(f'arg_{i}_{j}' for j in range(numinputs[i]))]
        sig = ",".join(args)
        call = ",".join(f"[{a}]" for a in args)
        code = rf'"'
def _temp({sig}): return f({call})
...
1 = dict(locals())
exec(code,1)
return 1["_temp"]

In [38]: def funclookup(l):
    return restrict(LOOKUP,2**l,1)

In [39]: f = funclookup(3)
f

Out[39]: <function _temp(arg_0_0, arg_0_1, arg_0_2, arg_0_3, arg_0_4, arg_0_5, arg_0_6, arg_0_7, arg_1_0, arg_1_1, arg_1_2)>
In [40]: f(1,0,1,1,1,0,1,1,0,1)
Out[40]: 1

In [44]: print(nandcode(funclookup(3)))

Temp[0] = NAND(X[8],X[8])
Temp[1] = NAND(X[6],Temp[0])
Temp[2] = NAND(X[7],X[8])
Temp[3] = NAND(Temp[1],Temp[2])
Temp[4] = NAND(X[8],X[8])
Temp[5] = NAND(X[4],Temp[4])
Temp[6] = NAND(X[5],X[8])
Temp[7] = NAND(Temp[5],Temp[6])
Temp[8] = NAND(X[9],X[9])
Temp[9] = NAND(Temp[7],Temp[8])
Temp[10] = NAND(Temp[3],X[9])
Temp[11] = NAND(Temp[9],Temp[10])
Temp[12] = NAND(X[8],X[8])
Temp[13] = NAND(X[2],Temp[12])
Temp[14] = NAND(X[3],X[8])
Temp[15] = NAND(Temp[13],Temp[14])
Temp[16] = NAND(X[8],X[8])
Temp[17] = NAND(X[0],Temp[16])
Temp[18] = NAND(X[1],X[8])
Temp[19] = NAND(Temp[17],Temp[18])
Temp[20] = NAND(X[9],X[9])
Temp[21] = NAND(Temp[19],Temp[20])
Temp[22] = NAND(Temp[15],X[9])
Temp[23] = NAND(Temp[21],Temp[22])
Temp[24] = NAND(X[10],X[10])
Temp[25] = NAND(Temp[23],Temp[24])
Temp[26] = NAND(Temp[11],X[10])
Y[0] = NAND(Temp[25],Temp[26])

In [45]: nandcircuit(funclookup(3))
Out[45]:
We can also track by how much the number of lines grows: we see that it is about $4 \cdot 2^\ell$:

In [42]: `len(nandcode(funclookup(1)).split('
'))/2**l for l in range(1,8,1)`


### Representing NAND programs as lists of triples

We can represent a NAND program in many ways including the string of its source code, as the graph corresponding to its circuit. One simple representation of a NAND program we will use is as the following:

We represent a NAND program of $t$ intermediate variables, $s$ lines, $n$ input variables, and $m$ input variables as a triple $(n,m,L)$ where $L$ is a list of $s$ triples of the form $(a,b,c)$ of numbers in $[n+t+m]$.

A triple $(a,b,c)$ corresponds to the line assigning to the variable corresponding $a$ the NAND of the variables corresponding to $b$ and $c$. We identify the first $n$ variables with the input and the last $m$ variables with the outputs.
We can again compute this representation using Python:

```python
In [46]:
def nandrepresent(f):
    """Compute the list of triple representation for a NAND program, given by a Python function."""
    n = numarguments(f)
    counter = n # to ensure unique temporary variables.
    L = [] # list of tuples

    def tempNAND(bar,blah):
        nonlocal L, counter
        var = counter
        counter += 1
        L += [(var,bar,blah)]
        return var

    outputs = runwith(lambda: f(*range(n)), "NAND", tempNAND)

    if type(outputs)==int: outputs = [outputs] # make single output into singleton list
    m = len(outputs)

    # make sure outputs are last m variables
    for j in range(m):
        def flip(a):
            nonlocal counter, outputs, j
            if a==outputs[j]: return counter+j
            return a

        L = [(flip(a),flip(b),flip(c)) for (a,b,c) in L]

    return (n,m,compact(L))

nandrepresent(XOR)

Out[46]: (2, 1, [(2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4)])
```

We can directly evaluate a NAND program based on its list of triples representation:

```python
In [47]:
def EVALnand(prog,X):
    """Evaluate a NAND program from its list of triple representation."""
    n,m,L = prog
    vartable = X+[0]*(max(max(a,b,c) for (a,b,c) in L)-n+1)
    for (a,b,c) in L:
        vartable[a] = NAND(vartable[b],vartable[c])

    return [vartable[-m+j] for j in range(m)]
```
Pruning (optional)

We can do some simple transformations to reduce the size of our programs/circuits. For example, if two gates have exactly the same inputs then we can identify them with one another. We can also use the equality NOT(NOT(a))=a, as well as remove unused variables.

```python
In [51]: def prune(prog):
    """Prune representation of program as tuples, removing duplicate lines and unused variables."""
    n,m,L = prog
    L = list(L)
    def identify(L,e,f):
        # identify vertex e with vertex f
        def ident(k):
            nonlocal e,f
            return f if k==e else k
        return [(ident(a),ident(b),ident(c)) for (a,b,c) in L]

    t = max([max(a,b,c) for (a,b,c) in L])+1

    while True:
        neighborhood = {} # a dictionary to hold reference to neighbors
        neighbors = {}
        found = False
        for (a,b,c) in L:
            N = frozenset([b,c])
            if a>=t-m: continue # don't remove output variables
            if N in neighborhood: # there was prior duplicate line
                L.remove((a,b,c))
                L = identify(L,a,neighborhood[N])
                found = True
                break
            if b==c and b in neighbors and len(neighbors[b])==1:
                # Line is NOT of NOT of prior line
                L.remove((a,b,c))
                L = identify(L,a,next(iter(neighbors[b])))
                found = True
                break
        if found: break
```

Out [51]:

```
"""
```
neighborhood[N] = a
neighbors[a] = N
touched = {a: False for a in range(t)}
for (a,b,c) in L:
touched[b] = True
touched[c] = True
for d in range(n,t-m,1): # remove non output and input variables t
    if not touched[d]:
        for (a,b,c) in L:
            if a==d:
                L.remove((a,b,c))
            if not found: break
    if not found: break
return (n,m,compact(L))

Some examples

In [52]:
    # Majority
    def MAJ(a,b,c):
        return NAND(NAND(NAND(a,b),NAND(a,c)),NAND(NAND(a,b),
                   NAND(a,c)),NAND(b,c))

    # Integer addition of two n bit numbers
    def ADD(A,B):
        n = len(A)
        Result = [0]*(n+1)
        Carry = [0]*(n+1)
        Carry[0] = zero(A[0])
        for i in range(n):
            Result[i] = XOR(Carry[i],XOR(A[i],B[i]))
            Carry[i+1] = MAJ(Carry[i],A[i],B[i])
        Result[n] = Carry[n]
        return Result

In [53]: f = restrict(ADD,2,2)
P = nandrepresent(f)

In [54]: all([(f(a,b,c,d)==EVALnand(prune(P),(a,b,c,d))) for a in [0,1] for b in [0,1] for c in [0,1] for d in [0,1] ])
Out[54]: True

From representation to code or graph

We can use the list of triples representation as a starting point to obtain the NAND program as a list of lines of code, or as a circuit (i.e., directed acyclic graph).
In [65]:
# Graphviz version

def gvrep2circuit(P):
    """Return circuit (i.e., graph) corresponding to NAND program P given in list of tuples representation."""

    n,m,L = P
    G = Digraph(graph_attr = {'rankdir':"LR"})

    for i in range(n):
        G.node(f"v{i}",label=f'X[{i}]', fontcolor='blue',shape='square')
        t = n

    for (a,b,c) in L:
        G.node(f"v{a}" ,label='\w0305',shape='invhouse',orientation='90')
        G.edge(f"v{b}" ,f"v{a}"
        G.edge(f"v{c}" ,f"v{a}"
        t = max(t,a,b,c)

    return G

In [66]:
# Schemdraw version

def sdrep2circuit(P):
    """Return circuit (i.e., graph) corresponding to NAND program P given in list of tuples representation."""

    n,m,L = P
    G = schem.Drawing(unit=.5,fontsize=8)
    curx,cury = 0,0

def incr(jump = False):
    nonlocal curx,cury, n;
    UX = 2.5
    UY = 3.5
    if not jump and (cury>-UY*(n-1)):
        cury -= UY
    else:
        cury = 0
        curx = (curx // UX)*UX + UX

    if (curx): curx = abs(cury)*UX/(2+n*UY)

    nodes = {}

    for i in range(n):
        nodes[f'v{i}'] = G.add(e.DOT, xy = [curx,cury], lftlabel=f'X[{i}]'
    incr()
```python
t = n

for (a,b,c) in L:
    var = f"v{a}"
    g = G.add(l.NAND2,xy=[curx,cury], d="right"#, label=var)
    incr()
    nodes[var] = g
    i1 = nodes[f"v{b}"]
    in1 = i1.out if "out" in dir(i1) else i1.end
    i2 = nodes[f"v{c}"]
    in2 = i2.out if "out" in dir(i2) else i2.end
    G.add(e.LINE,xy=in1,to=g.in1)
    G.add(e.LINE,xy=in2,to=g.in2)
    t = max(t,a,b,c)

    t += 1
    incr(True)
    for j in range(m):
        g = nodes[f"v{t-m+j}"],
        o = G.add(e.DOT,xy=[curx,cury],rgtlabel=f'Y[{j}]')
        G.add(e.LINE,xy=o.out,to=o.start)
        incr()

return G
```

```
In [67]: def rep2circuit(P,method="Graphviz"):
     ...:     return gvrep2circuit(P) if method=="Graphviz" else sdrep2circuit(P)

In [68]: gvrep2circuit(P)
Out[68]:
```

In [69]: sdrep2circuit(P)

In [70]: rep2circuit(prune(P))

Out[70]:

In [72]: def rep2code(P):
    """Return NAND code corresponding to NAND program P, given in list of tuples representation"""
    n, m, L = P
    code = ""
    
    t = max(\max(a, b, c) for (a, b, c) in L) + 1
    
    def var(a):
        if a < n: return f"X[{a}]"
        if a >= t - m: return f"Y[{a-t+m}]"
        return f"Temp[{a-n}]"
    
    for (a, b, c) in L:
        code += f"\n{var(a)} = NAND{\{var(b)}{\var(c)}\"
    
    return code
In [74]: `print(rep2code(P))`

Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(Temp[1],Temp[1])
Temp[3] = NAND(X[0],X[2])
Temp[4] = NAND(X[0],Temp[3])
Temp[5] = NAND(X[2],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(Temp[2],Temp[6])
Temp[8] = NAND(Temp[2],Temp[7])
Temp[9] = NAND(Temp[6],Temp[7])
Y[0] = NAND(Temp[8],Temp[9])
Temp[10] = NAND(Temp[2],X[0])
Temp[12] = NAND(Temp[10],Temp[11])
Temp[13] = NAND(Temp[2],X[0])
Temp[14] = NAND(Temp[2],X[2])
Temp[15] = NAND(Temp[13],Temp[14])
Temp[16] = NAND(Temp[12],Temp[15])
Temp[17] = NAND(X[0],X[2])
Temp[18] = NAND(Temp[16],Temp[17])
Temp[19] = NAND(X[1],X[3])
Temp[20] = NAND(X[1],Temp[19])
Temp[21] = NAND(X[3],Temp[19])
Temp[22] = NAND(Temp[20],Temp[21])
Temp[23] = NAND(Temp[18],Temp[22])
Temp[24] = NAND(Temp[18],Temp[23])
Temp[25] = NAND(Temp[22],Temp[23])
Y[1] = NAND(Temp[24],Temp[25])
Temp[26] = NAND(Temp[18],X[1])
Temp[27] = NAND(Temp[18],X[3])
Temp[28] = NAND(Temp[26],Temp[27])
Temp[29] = NAND(Temp[18],X[1])
Temp[30] = NAND(Temp[18],X[3])
Temp[31] = NAND(Temp[29],Temp[30])
Temp[32] = NAND(Temp[28],Temp[31])
Temp[33] = NAND(X[1],X[3])
Y[2] = NAND(Temp[32],Temp[33])

In [75]: `print(rep2code(prune(P)))`

Temp[0] = NAND(X[0],X[0])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(Temp[1],Temp[1])
Temp[3] = NAND(X[0],X[2])
Temp[4] = NAND(X[0],Temp[3])
Temp[5] = NAND(X[2],Temp[3])
Temp[6] = NAND(Temp[4],Temp[5])
Temp[7] = NAND(Temp[2],Temp[6])
Temp[8] = NAND(Temp[2],Temp[7])
Temp[9] = NAND(Temp[6],Temp[7])
Y[0] = NAND(Temp[8],Temp[9])
Temp[10] = NAND(Temp[2],X[0])
Temp[12] = NAND(Temp[10],Temp[11])
We can now redefine the nandcircuit and nandcode functions to work as follows:

1. First obtain the list of triples representation
2. Then prune it
3. Then transform it to either code or circuit appropriately

In [76]:
```python
def nandcode(f):
    return rep2code(prune(nandrepresent(f)))

def nandcircuit(f, method="Graphviz"):
    return rep2circuit(prune(nandrepresent(f)),method)
```

In [77]:
nandcircuit(inc7,"Graphviz")
Out[77]:

Universal circuit evaluation or NAND interpreter in NAND

We can construct a NAND program $S$ that given the representation of a NAND program $Q$ and an input $x$, outputs $Q(x)$. We can obviously compute such a function since every finite function is computable by a NAND program, but it turns out we can do so in a program that is polynomial in the size of $S$ (even quasilinear but we won't show that here).

We start with a reimplementation of NANDEVAL in Python:

In [78]:
```python
def GET(V,i): return V[i]
def UPDATE(V,i,b):
    V[i]=b
    return V
```
def NANDEVAL(n, m, L, X):
    # Evaluate a NAND program from its list of triple representation.
    s = len(L)  # number of lines
    t = max(max(a, b, c) for (a, b, c) in L) + 1  # maximum index in L + 1

    Vartable = [0] * t  # we'll simply use an array to store data

    # Load input values to Vartable:
    for i in range(n):
        Vartable = UPDATE(Vartable, i, X[i])

    # Run the program
    for (i, j, k) in L:
        a = GET(Vartable, j)
        b = GET(Vartable, k)
        c = NAND(a, b)
        Vartable = UPDATE(Vartable, i, c)

    # Return outputs Vartable[t-m], Vartable[t-m+1],.....,Vartable[t-1]
    return [GET(Vartable, t - m + j) for j in range(m)]

In [79]: L = ((2, 0, 1), (3, 0, 2), (4, 1, 2), (5, 3, 4))
print(NANDEVAL(2, 1, L, (0, 1)))  # XOR(0,1)
print(NANDEVAL(2, 1, L, (1, 1)))  # XOR(1,1)

[1]
[0]

Now transform this to work with the representation of L as a binary string, namely as a sequence of $3s$ numbers in $[t]$, each represented as a string of length $\ell = \lceil \log 3s \rceil$.

In [80]: from math import ceil, floor, log

def triplelist2string(L):
    """Transform list of triples into its representation as a binary string""
    s = len(L)
    ell = ceil(log2(3*s))
    B = [0] * (3*s*ell)
    FlatL = [a for T in L for a in T]
    for i in range(3*s):
        for j in range(ell):
            B[ell*i + j] = floor(FlatL[i] / 2**j) % 2
    return B

Evaluating a NAND program given its string representation

We can now present NANDEVALBIN which will be a Python function that evaluates a NAND program given the representation of the program as a binary string. (We assume the
parameters $n,m,s,t$ are given: we could have assumed they are part of the string representation, but this only makes things messier.)

In [81]:

```python
def NANDVALBIN(n,m,s,t,B,X):
    
    
    
    ell = ceil(log2(3*s))
    Vartable = [0] * (2**ell)  # we'll simply use an array to store data

    # Load input values to Vartable:
    for i in range(n): Vartable[i] = X[i]

    # Run the program
    for c in range(s):
        i = B[c*3*ell+d] for d in range(ell)]
        j = B[c*3*ell+ell+d] for d in range(ell)]
        k = B[c*3*ell+2*ell+d] for d in range(ell)]
        a = GETB(Vartable,j)
        b = GETB(Vartable,k)
        c = NAND(a,b)
        Vartable = UPDATEB(Vartable,i,c)

    # Return outputs Vartable[t-m], Vartable[t-m+1],....,Vartable[t-1]
    return [Vartable[t-m+j] for j in range(m)]
```

We'll need some utility functions to deal with the binary representation (you can ignore these at a first read)

In [82]:

```python
# utility functions

def nandconst(b,x):
    
    
    
    if b: return one(x)
    return zero(x)

def i2s(i,ell=0):
    
    
    
    if not ell: ell = ceil(log2(i))
    return [floor(i/2**j) % 2 for j in range(ell)]

def GETB(V,i):
    
    
    
    return LOOKUP(V,i)

def EQUALB(j,i):
    flag = zero(i[0])  # if flag is one then i is different from j
    for t in range(len(j)):
        if type(j[t])==int:
            temp = NOT(i[t]) if j[t] else COPY(i[t])
        else:
            temp = OR(AND(j[t],NOT(i[t])),AND(NOT(j[t]),i[t]))
        flag = OR(temp,flag)
    return NOT(flag)

def UPDATEB(V,i,b):
    ell = ceil(log2(len(V)))
    ```
Now let's test this out on the XOR function

```python
UV = [v]~\text{len}(v)
for j in range(\text{len}(V)):
    a = \text{EQUALB}(i2s(j,\ell),i)
    UV[j] = IF(a,b,V[j])
return UV
```

We can also try this on the XOR of 4 bits

```python
\text{def} \text{XOR4}(a,b,c,d): \text{return} \text{XOR}(\text{XOR}(a,b),\text{XOR}(c,d))
```

From Python to NAND

We now transform the Python program \text{NANDEVALBIN} into a NAND program. In fact, it turns out that all our python code can be thought of as "syntactic sugar" and hence we can do this transformation automatically.

Specifically, for every numbers $n,m,s,t$ we will construct a NAND program $P$ on $3s\ell+n$ inputs (for $\ell = \lceil \log_2(3s) \rceil$) that on input a string $B \in \{0,1\}^{3s\ell+n}$ and $x \in \{0,1\}^n$ outputs $P(x)$ where $P$ is the program represented by $B$.
To do so, we simply first restrict \texttt{NANDEVALBIN} to the parameters $n,m,s,t$ and then run our usual "unsweetener" to extract the NAND code from it.

```python
In [89]: def nandevalfunc(n,m,s,t):
    """Given $n,m,s,t$, return a function $f$ that on input $B,X$ returns the evaluation of the program encoded by $B$ on $X$""
    ell = ceil(log2(3*s))
    return restrict(lambda B,X: NANDEVALBIN(n,m,s,t,B,X),3*s*ell,n)
```

For example, let us set $n,m,s,t$ to be the parameters as in the XOR function.

```python
In [90]: n,m,L = nandrepresent(XOR)
s = len(L)
t = max(max(T for T in L)+1
XORstring = triplelist2string(L)
```

```python
In [91]: f = nandevalfunc(n,m,s,t)
```

```python
In [92]: f(*XORstring+[0,1])
Out[92]: [1]
```

```python
In [93]: f(*XORstring+[1,1])
Out[93]: [0]
```

$f$ above is still a Python function, but we now transform it into a NAND function.

```python
In [94]: nand_eval_in_nand = nandrepresent(f)
```

And test it out.

```python
In [95]: NANDEVAL(*nand_eval_in_nand,XORstring+[1,0])
Out[95]: [1]
```

It is important to note that \texttt{nand\_eval\_in\_nand} is not specific to the XOR function: it will evaluate any NAND program of the given parameters $n,m,s,t$. Some "hardwiring" of parameters is inherently needed since NAND programs only take a fixed number of inputs.

We could have also generated a NAND program that computes $t$ from the other parameters. We just avoided it because it's a little more cumbersome.

Let's see that this doesn't work out just for XOR.

```python
In [96]: n,m,L = nandrepresent(restrict(increment,1))
```

```python
In [97]: s = len(L)
t = max(max(T for T in L)+1
( +
```
If you are curious, here is the code and circuit representation of the NAND eval function for certain parameters:

```python
In [111]: show_code_and_circ = False  # Change to "True" to run the (long) computation

In [109]: # (pruning took too long, so skipped it for now)
if show_code_and_circ:
    code = rep2code(nand_eval_in_nand)
    # since it's so long, let's just print the first and last 10 lines:
    lines = code.split("\n")
    print("\n".join(lines[:10]+["..."]+lines[-10:]))
```

Temp[0] = NAND(X[5],X[5])
Temp[1] = NAND(X[0],Temp[0])
Temp[2] = NAND(X[0],X[5])
Temp[3] = NAND(Temp[1],Temp[2])
Temp[4] = NAND(X[5],X[5])
Temp[5] = NAND(X[0],Temp[4])
Temp[6] = NAND(X[0],X[5])
Temp[7] = NAND(Temp[5],Temp[6])
Temp[8] = NAND(X[6],X[6])
...
Temp[12238] = NAND(Temp[12236],Temp[12237])
Temp[12239] = NAND(X[139],X[139])
Temp[12240] = NAND(Temp[12239],Temp[12239])
Temp[12241] = NAND(Temp[12238],Temp[12238])
Temp[12242] = NAND(Temp[12240],Temp[12241])
Temp[12243] = NAND(Temp[12242],Temp[12242])
Temp[12244] = NAND(Temp[12243],Temp[12243])
Temp[12245] = NAND(Temp[11024],Temp[12244])
Temp[12246] = NAND(Temp[11273],Temp[12243])
Temp[12247] = NAND(Temp[12245],Temp[12246])
B

The NAND++ Programming Language
The NAND++ Programming language

*Version: 0.2*

The NAND++ programming language was designed to accompany the upcoming book "Introduction to Theoretical Computer Science". This is an appendix to this book, which is also available online as a Jupyter notebook in the boazbk/nandnotebooks on Github. You can also try the live binder version.

The NAND++ programming language is defined in Chapter 6: "Loops and Infinity"

The NAND programming language we saw before corresponds to non uniform, finite computation.

NAND++ captures uniform computation and is equivalent to Turing Machines.

One way to think about NAND++ is

\[ \text{NAND}++ = \text{NAND} + \text{loops} + \text{arrays} \]

Enhanced NAND++

We start by describing "enhanced NAND++ programs", and later we will describe "vanilla" or "standard" NAND++.

Enhanced NAND++ programs have the following form: every line is either of the form

- `foo = NAND(bar,blah)`
- `i += foo`
- `i -= foo`

where `foo` is a variable identifier that is either a *scalar variable*, which is a sequence of letters, numbers and underscores, or an *array element*, which starts with a capital letter, and ends with `[i]`

We have a special variable `loop`. If `loop` is set to 1 at the end of the program then execution goes back to the beginning.
We have the special input and output arrays $X[.]$ and $Y[.]$ but because their length is not fixed in advance, we also have $X\text{valid}[.]$ and $Y\text{valid}[.]$ arrays. The input is $X[0], \ldots, X[n-1]$ where $n$ is the smallest integer such that $X\text{valid}[n] = 0$. The output is $Y[0], \ldots, Y[m-1]$ where $m$ is the smallest integer such that $Y\text{valid}[m] = 0$.

The default value of every variable in NAND++ is zero.

**Ignore in first read: utility code**

We use some utility code, which you can safely ignore in first read, to allow us to write NAND++ code in Python.

```python
In [1]: # utility code
%run "NAND programming language.ipynb"
from IPython.display import clear_output
clear_output()

In [2]: # Ignore this utility function in first and even second and third read
import inspect
import ast
import astor

def noop(f):
    return f;

def runwithstate(f):
    """Modify a function f to take and return an argument state and make all names relative to state.""
    tree = ast.parse(inspect.getsource(f))
    tmp = ast.parse("def _temp(state):
        pass\n").body[0]
    args = tmp.args
    name = tmp.name
    tree.body[0].args = args
    tree.body[0].name = name
    tree.body[0].decorator_list = []

class AddState(ast.NodeTransformer):
    def visit_Name(self, node: ast.Name):
        if node.id == "enandpp":
            return ast.Name(id="noop", ctx=Load())
    
        new_node = ast.Attribute(ast.copy_location(ast.Name('state'), a
st.Load()), node), node.id,
        ast.copy_location(ast.Load(), node))
        return ast.copy_location(new_node, node)

    tree = AddState().visit(tree)
    tree.body[0].body = tree.body[0].body + [ast.parse("return state")]
    tree = ast.fix_missing_locations(tree)
    src = astor.to_source(tree)
    # print(src)
    exec(src, globals())
    _temp.original_func = f
    return _temp
```
```python
def enandpp(f):
g = runwithstate(f)
def _temp1(X):
    nonlocal g
    return ENANDPPEVAL(g,X)
_temp1.original_func = f
_temp1.transformed_func = g
return _temp1

In [3]: # ignore utility class in first and even second or third read

from collections import defaultdict
class NANDPPstate:
    """State of a NAND++ program.""
    def __init__(self):
        self.scalars = defaultdict(int)
        self.arrays = defaultdict(lambda: defaultdict(int))
        # eventually should make self.i non-negative integer type
    def __getattr__(self, var):
        g = globals()
        if var in g and callable(g[var]): return g[var]
        if var[0].isupper():
            return self.arrays[var]
        else:
            return self.scalars[var]

In [4]: def ENANDPPEVAL(f,X):
    """Evaluate an enhanced NAND++ function on input X""
    s = NANDPPstate()
    for i in range(len(X)):
        s.X[i] = X[i]
        s.Xvalid[i] = 1
    while True:
        s = f(s)
        if not s.loop: break
    res = []
    i = 0
    while s.Yvalid[i]:
        res += [s.Y[i]]
        i+= 1
    return res

In [5]: def rreplace(s, old, new, occurrence=1):
    # from stackoverflow
    li = s.rsplit(old, occurrence)
    return new.join(li)

def ENANDPPcode(P):
    """Return ENAND++ code of given function""
```
code = ''
counter = 0

class CodeENANDPPcounter:
    def __init__(self, name="i"):  
        self.name = name

    def __iadd__(self, var):
        nonlocal code
        code += f'\ni += {var}'
        return self

    def __isub__(self, var):
        nonlocal code
        code += f'\ni -= {var}'
        return self

    def __str__(self): return self.name

class CodeNANDPPstate:

    def __getattribute__(self, var):
        # print(f"getting {var}"")
        if var=='i': return CodeENANDPPcounter()
        g = globals()
        if var in g and callable(g[var]): return g[var]
        if var[0].isupper():
            class Temp:
                def __getitem__(self, k):
                    return f"{var}[{str(k)}]"
                def __setitem__(self, k, v):
                    setattr(self, f"{var}[{str(k)}]", v)
            return Temp()
        return var

    def __init__(self):
        pass

    def __setattr__(self, var, val):
        nonlocal code
        if var=='i': return
        if code.find(val)==-1:
            code += f'\n{var} = {val}'
        else:
            code = rreplace(code, val, var)

s = CodeNANDPPstate()

def myNAND(a,b):
    nonlocal code, counter
    var = f'temp_{counter}'
    counter += 1
    code += f'\n{var} = NAND({a},{b})'
    return var
Our first NAND++ program

Here is an enhanced NAND++ program to increment a number:

```
In [6]: @enandpp
def inc():
    carry = IF(started,carry,one(started))
    started = one(started)
    Y[i] = XOR(X[i],carry)
    carry = AND(X[i],carry)
    Yvalid[i] = one(started)
    loop = COPY(Xvalid[i])
    i += loop

In [7]: inc([1,1,0,0,1])
Out[7]: [0, 0, 1, 0, 1, 0]
```

```
In [8]: print(ENANDPPcode/inc))
```

And here is an enhanced NAND++ program to compute the XOR function on unbounded length inputs (it uses XOR on two variables as a subroutine):

```
In [9]: @enandpp
def UXOR():
    Yvalid[0] = one(X[0])
    Y[0] = XOR(X[i],Y[0])
```

```
loop = Xvalid[i]
i += Xvalid[i]
```

```
In [10]: UXOR([1,1,0,0,1,1])
Out[10]: [0]
```

```
In [11]: print(ENANDPPcode(UXOR))
```

```
temp_0 = NAND(X[0],X[0])
Yvalid[0] = NAND(X[0],temp_0)
temp_2 = NAND(X[i],Y[0])
temp_3 = NAND(X[i],temp_2)
temp_4 = NAND(Y[0],temp_2)
Y[0] = NAND(temp_3,temp_4)
loop = Xvalid[i]
i += Xvalid[i]
```

"""Vanilla" NAND++

In "vanilla" NAND++ we do not have the commands `i += foo` and `i -= foo` but rather `i` travels obliviously according to the sequence 0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, 2, …

```
In [12]: def index():
    """Generator for the values of i in the NAND++ sequence""
    i = 0
    last = 0
direction = 1
    while True:
        yield i
        i += direction
        if i>last:
            direction = -1
            last = i
        if i==0: direction = +1

a = index()
[next(a) for i in range(20)]
```

```
Out[12]: [0, 1, 0, 1, 2, 1, 0, 1, 2, 3, 2, 1, 0, 1, 2, 3, 2, 1]
```

```
In [13]: def NANDPPEVAL(f,X):
    """Evaluate a NAND++ function on input X""
    s = NANDPPstate() # initialize state

    # copy input:
    for i in range(len(X)):
        s.X[i] = X[i]
        s.Xvalid[i] = 1

    # main loop:
    for i in index():
        s.i = i
        #/ /
```
Here is the increment function in vanilla NAND++. Note that we need to keep track of an Array Visited to make sure we only add the carry once per location.

```python
In [14]: @nandpp
def inc():
    carry = IF(started,carry,one(started))
    started = one(started)
    Y[i] = IF(Visited[i],Y[i],XOR(X[i],carry))
    Visited[i] = one(started)
    carry = AND(X[i],carry)
    Yvalid[i] = one(started)
    loop = Xvalid[i]

In [15]: inc([1,1,0,1,1])
Out[15]: [0, 0, 1, 1, 1, 0]
```

And here is the "vanilla NAND++" version of XOR:

```python
In [16]: def vuXOR():
    Yvalid[0] = one(X[0])
    Y[0] = IF(Visited[i],Y[0],XOR(X[i],Y[0]))
    Visited[i] = one(X[0])
    loop = Xvalid[i]

In [17]: vuXOR([1,0,0,1,0,1,1])
Out[17]: [0]
```

```python
In [18]: def NANDPPcode(P):
    """R t NAND++ d f i f ti """
```

Return NAND++ code of given function

code = ''
counter = 0

class CodeNANDPPstate:

def __getattr__(self, var):
    # print(f"getting \{var\}")
    g = globals()
    if var in g and callable(g[var]): return g[var]
    if var[0].isupper:
        class Temp:
            def __getitem__(self, k):
                return var+"[i]"
            def __setitem__(self, k, v):
                setattr(self, var+"[i]", v)
            return Temp()
        return var

def __init__(self):
    pass

def __setattr__(self, var, val):
    nonlocal code
    # print(f"setting \{var\} to \{val\}")
    if code.find(val)==-1:
        code += f'\n{var} = {val}''
    else:
        code = rreplace(code, val, var)

s = CodeNANDPPstate()

def myNAND(a, b):
    nonlocal code, counter
    var = f'temp_{counter}''
    counter += 1
    code += f'\n{var} = NAND({a},{b})''
    return var

s = runwith(lambda : P.transformed_func(s),"NAND",myNAND)

return code

# utility code - replace string from right, taken from stackoverflow

def rreplace(s, old, new, occurrence=1):
    li = s.rsplit(old, occurrence)
    return new.join(li)

In [19]: print(NANDPPcode(inc))

temp_0 = NAND(started,started)
temp 1 = NAND(started temp 0)
Eventually we will have here code to automatically transform an enhanced NAND++ program into a NAND++ program. At the moment, let us just give the high level ideas. See Chapter 6 in the book for more details.

To transform an enhanced NAND++ program to a standard NAND++ program we do the following:

1. We make all our operations "guarded" in the sense that there is a special variable `noop` such that if `noop` equals 1 then we do not make any writes.
2. We use a `Visited` array to keep track of all locations we visited, and use that to keep track of an decreasing variable that is equal to 1 if and only the value of `i` in the next step will be one smaller.
3. If we have an operation of the form `i += foo` or `i -= foo` at line $\ell$ then we replace it with lines of code that do the following:
   a. (Guarded) set `temp_\ell = foo`
   b. (Unguarded) If `Waitingline_\ell` and `Restart[i]`: set `noop=0` if increasing is equal to `wait_increasing`. (Otherwise `noop` stays the same.)
   c. (Guarded) set `Restart[i]` to 1.
   d. (Guarded) set `Waitingline_\ell` to 1.
   e. (Guarded) set `wait_increasing` to 1 if the operation is `i += foo` and to 0 if it's `i -= foo`
f. (Guarded) set noop = temp_\ell

g. (Unguarded) set temp_\ell = 0

h. (Guarded) set Restart[i] to 0.

i. (Guarded) set Waitingline_\ell to 0.
C

The Lambda Calculus
λ calculus

This is an appendix to upcoming book "Introduction to Theoretical Computer Science", which is also available online as a Jupyter notebook in the boazbk/nandnotebooks on Github. You can also try the live binder version.

The λ calculus is discussed in Chapter 7: "Equivalent Models of Computation"

Click here for the live Binder version. (Service can sometimes be slow.)

This Python notebook provides a way to play with the lambda calculus and evaluate lambda expressions of the form λvar1(exp1) λvar2(exp2) .... If you don't know Python you can safely ignore the Python code and skip below to where we actually talk about the λ calculus itself.

To better fit with python there are two main differences:

- Instead of writing λvar.exp we write λvar(exp)
- Instead of simply concatenating two expressions exp1 exp2 we use the * operator and write exp1 * exp2. We can also use exp1, exp2 if they are inside a function call or a variable binding parenthesis.
- To reduce an expression exp, use exp.reduce()
- Since Python does not allow us to override the default 0 and 1 we use _0 for λx(y(y)) and _1 for λx(y(x)).

Python code (can skip at first read)

If you don't know Python feel free to skip ahead to the part where we play with the λ calculus itself.

```python
In [1]:
# We define an abstract base class Lambdaexp for lambda expressions
# It has the following subclasses:
# Applicableexp: an expression of the form λx.exp
# Combinedexp: an expression of the form (exp,exp')
# Boundvar: an expression corresponding to a bounded variable
# Unboundvar: an expression corresponding to a free variable
#
# The main operations in a Lambdaexp are:
# 1. Replace: given exp,x and exp', obtain the expression exp[x-->exp']
# 2. Reduce: continuously evaluate expressions to obtain a simpler form
# 3. Apply: given exp,exp', if exp is applicable then apply it to exp', otherwise combine the two
# (we also use the * operator for it)

import operator ,functools
```
class Lambdaexp:
    """Lambda expressions base class""

counter = 0
call_by_name = True  # if False then do normal form evaluation.

def __init__(self):
    self.mykey = {}

def apply(self, other):
    """Apply expression on an argument""
    return self*other

def _reduce(self, maxlevel=100):
    """Reduce expression""
    return self

def replace(self, old, new):
    """Replace all occurrences of old with new""
    raise NotImplemented

def bounded(self):
    """Set of bounded variables inside expression""
    return set()

def asstring(self, m, pretty=False):
    """Represent self as a string mapping bounded variables to particular numbers.""
    raise NotImplemented

#---------------------------------------------------------------
# Ignore this code in first read: Python specific details

lambdanames = {}
reducedstrings = {}

def reduce(self, maxlevel=100):
    if not maxlevel: return self
    # m = {b:b for b in self.bounded()} 
    # t = Lambdaexp.reducedstrings.get((self.asstring(m),maxlevel),None)
    # if t: return t
    return self._reduce(maxlevel)
    # k = t.asstring(m)
    # for i in range(maxlevel+1):
    #     Lambdaexp.reducedstrings[(k,i)] = t
    # return t

def __mul__(self, other):
    """Use * for combining.""
    return Combinedexp(self, other) if other else self

def __call__(self, *args)


def __call__(self, *args):
    """Use function call for application""
    return functools.reduce(operator.mul, args, self)

def _key(self, maxlevel=100):
    if maxlevel not in self.mykey:
        return self.reduce(maxlevel)._repr()  
        # for i in range(maxlevel+1): self.mykey[i] = s
        # return self.mykey[maxlevel]

    def __eq__(self, other):
        return self._key() == other._key() if isinstance(other, Lambdaexp) else False

def __hash__(self):
    return hash(self._key())

def __repr__(self, pretty=False):
    B = sorted(self.bounded())
    m = {}
    for v in B: m[v] = len(m)
    return self.asstring(m, pretty)

def _repr_pretty_(self, p, cycle):
    if cycle: p.text(self._repr())
    p.text(self.reduce()).__repr__(True)

    def addconst(self, srep):
        """Return either exp.string or replaced with a keyword if it's in table.""
        if self in Lambdaexp.lambdanames: return blue(Lambdaexp.lambdanames[self])
        return srep

In [2]:
# Utility functions: print color
def bold(s, justify=0):
    return "\x1b[1m"+s.ljust(justify)+"\x1b[21m"

def underline(s, justify=0):
    return "\x1b[4m"+s.ljust(justify)+"\x1b[24m"

def red(s, justify=0):
    return "\x1b[31m"+s.ljust(justify)+"\x1b[0m"

def green(s, justify=0):
    return "\x1b[32m"+s.ljust(justify)+"\x1b[0m"
def blue(s, justify=1):
    return r"\x1b[34m"+s.ljust(justify)+r"\x1b[0m"

---

In [3]:

class Applicableexp(Lambdaexp):
    """Lambda expression that can be applied"""

    def __init__(self, exp, name):
        Lambdaexp.counter += 1
        self.arg = Lambdaexp.counter
        self.inner = exp.replace(name, Boundvar(self.arg))
        super().__init__()

    def apply(self, other):
        return self.inner.replace(self.arg, other)

    def replace(self, old, new):
        if self.arg==old: self.arg = new.myid
        return Applicableexp(self.inner.replace(old, new), self.arg)

    def bounded(self): return self.inner.bounded()|{self.arg}

    def _reduce(self, maxlevel=100):
        if Lambdaexp.call_by_name: return self
        inner = self.inner.reduce(maxlevel-1)
        return Applicableexp(inner, self.arg)

    def asstring(self, m, pretty=False):
        if not pretty: return r"λ"+Boundvar(self.arg).asstring(m, False)+"."
        return self.addconst(green(r"λ")+Boundvar(self.arg).asstring(m, True)+"."

In [4]:
class Boundvar(Lambdaexp):
    """Bounded variable"""

    def __init__(self, arg):
        self.myid = arg
        super().__init__()

    def replace(self, argnum, exp): return exp if argnum==self.myid else self

    def bounded(self): return { self.myid }

    def asstring(self, m, pretty=False):
        arg = m.get(self.myid, self.myid)
        return r"λ"+Boundvar(arg).asstring(m, pretty)+"."
class Unboundvar(Lambdaexp):
    """Unbounded (free) variable."""
    def __init__(self, name):
        self.name = name
        super().__init__()
    def replace(self, name, arg):
        return arg if name == self.name else self
    def asstring(self, m, pretty=False):
        return self.addconst(self.name) if pretty else self.name

class Combinedexp(Lambdaexp):
    """Combined expression of two expressions."""
    def __init__(self, exp1, exp2):
        self.exp1 = exp1
        self.exp2 = exp2
        super().__init__()
    def replace(self, arg, exp):
        return Combinedexp(self.exp1.replace(arg, exp), self.exp2.replace(arg, exp))
    def bounded(self):
        return self.exp1.bounded() | self.exp2.bounded()
    def _reduce(self, maxlevel=100):
        e1 = self.exp1.reduce(maxlevel-1)
        if isinstance(e1, Applicableexp):
            return e1.apply(self.exp2).reduce(maxlevel-1)
        return Combinedexp(e1, self.exp2)
    def asstring(self, m, pretty=False):
        s = f'{{self.exp1.asstring(m,FALSE)}} {{self.exp2.asstring(m,FALSE)}}'
        if not pretty:
            return s
        return f'{{self.exp1.asstring(m,TRUE)}} {{self.exp2.asstring(m,TRUE)}}'

In [5]: class λ:
    """Binds a variable name in a Lambda expression""
    def __init__(self,*varlist):
        """Get list of unbounded variables (for example a,b,c) and returns an operator that binds an expression exp to Aa(λb(λc(exp))) and so on."""
        if not varlist:
            raise Exception("Need to bind at least one variable")
        self.varlist = varlist[::1]
def bindexp(self,exp):
    res = exp
    for v in self.varlist:
        res = Applicableexp(res,v.name)
    return res

# Ignore this code in first read: Python specific details

def __call__(self,*args):
    exp = functools.reduce(operator.mul, args[1:], args[0])
    return self.bindexp(exp)

Initalization

The above is all the code for implementing the λ calculus. We now add some convenient global variables: λa .... λz and a ... z for variables, and 0 and 1.

In [6]:
Lambdaexp.lambdanames = {}
import string

def initids():
    """Set up parameters a...z and correpsonding Binder objects λa..λz""
    lcase = list(string.ascii_lowercase)
    ids = lcase + [n + "_" for n in lcase]
    for name in ids:
        var = Unboundvar(name)
        g[name] = var
        g["λ"+name] = λ(var)
        Lambdaexp.lambdanames[var] = name

In [7]: initids()()

In [8]: # testing...
    λy(y)

Out[8]: λα.(α)

In [9]: λ(a,b)(a)

Out[9]: λα.(λβ.(α))

In [10]: def setconstants(g,consts):
    """Set up constants for easier typing and printing.""

    for name in consts:
        Lambdaexp.lambdanames[consts[name]] = name
        if name[0].isalpha():
            g[name]=consts[name]
else: # Numeric constants such as 0 and 1 are replaced by _0 and _1
    g["_"+name] = consts[name]

setconstants(globals(),{"1" : λ(x,y)(x) , "0" : λ(x,y)(y) })

def register(g,*args):
    for name in args:
        Lambdaexp.lambdanames[g[name]] = name

In [11]: # testing
   λa(λz(a))

Out[11]: 1

\textbf{λ calculus playground}

We can now start playing with the \textit{λ} calculus

If you want to use the \textit{λ} character you can copy paste it from here: \texttt{λ}

Let's start with the function \textit{λ}x,y.y, also known as 0

In [12]: \texttt{λa(λb(b))}

Out[12]: 0

Our string representation recognizes that this is the 0 function and so "pretty prints" it. To see the underlying \textit{λ} expression you can use \texttt{__repr__}()

In [13]: \texttt{λa(λb(b)).__repr__()}

Out[13]: '\texttt{λα.(λβ.(β))}'

Let's check that _0 and _1 behave as expected

In [14]: _1(a,b)

Out[14]: a

In [15]: _0(a,b)

Out[15]: b

In [16]: _1

Out[16]: 1

In [17]: _1(_0)

Out[17]: \texttt{λα.(θ)}
Here is an exercise:

**Question:** Suppose that $F = \lambda f. (\lambda x. (fx)f)$, $1 = \lambda x. (\lambda y. x)$ and $0 = \lambda x. (\lambda y. y)$. What is $F\ 1\ 0$?

a. $1$

b. $0$

c. $\lambda x. 1$

d. $\lambda x. 0$

Let's evaluate the answer

**Some useful functions**

Let us now add some of the basic functions in the $\lambda$ calculus

And test them out
We can make lists of bits as follows:

```
In [30]: def makelist(L):
    """Construct a λ list of 0's and 1's."""
    if not L: return NIL
    h = 1 if L[0] else 0
    return PAIR(h, makelist(L[1:]))
```

```
In [31]: L = makelist(1, 0, 1)
L
Out[31]: λ.(((μ 1) ((PAIR 0) ((PAIR 1) NIL))))
```

```
In [32]: HEAD(L)
Out[32]: 1
```

```
In [33]: TAIL(L)
Out[33]: λ.(((μ 0) ((PAIR 1) NIL)))
```

```
In [34]: HEAD(TAIL(L))
Out[34]: 0
```

```
In [35]: HEAD(TAIL(TAIL(L)))
Out[35]: 1
```

```
Recursion
```
We now show how we can implement recursion in the λ calculus. We start by doing this in Python. Let's try to define XOR in a recursive way and then avoid recursion.

```python
# XOR of 2 bits
def xor2(a,b): return 1-b if a else b

# XOR of a list - recursive definition
def xor(l): return xor2(l[0],xor(l[1:])) if l else 0
xor([1,0,0,1,1])
```

Now let's try to make a non recursive definition, by replacing the recursive call with a call to `me` which is a function that is given as an extra argument:

```python
def myxor(me,L):
    return 0 if not L else xor2(me[0],me[1:]),me[2],me[3],me[4]
```

The first idea is to try to implement `xor(L)` as `myxor(me,xor,L)` but this will not work:

```python
def xor(L):
    return myxor(me,L)
try:
xor([0,1,1])
except Exception as e:
    print(e)
```

myxor() missing 1 required positional argument: 'L'

The issue is that `myxor` takes two arguments, while in `me` we only supply one. Thus, we will modify `myxor` to `tempxor` where we replace the call `me(x)` with `me(me,x)`:  

```python
def tempxor(me,L):
    return myxor(lambda x: me(me,x),L)
def xor(L):
    return tempxor(tempxor,L)
```

This works!

Let's now generatilize this to any function. The `RECURSE` operator will take a function `f` that takes two arguments `me` and `x` and return a function `g` where the calls to `me` are replaced with calls to `g`:

```python
def RECURSE(f):
    def fttemp(me,x):
        return f(lambda x: me(me,x),x)
    return fttemp
```

```python
RECURSE(f)
```
The λ version

We now repeat the same arguments with the λ calculus:

```
# XOR of two bits
XOR2 = \(a, b\) IF(a, IF(b, _0, _1), b))

# Recursive XOR with recursive calls replaced by m parameter
myXOR = \(m, l\) IF(IS_EMPTY(l), _0, XOR2(HEAD(l), m(TAIL(l)))))

# Recurse operator (aka Y combinator)
RECURSE = \(\lambda f\) ((\(\lambda m(f(m*m))\))(\(\lambda m(f(m*m))\)))

# XOR function
XOR = RECURSE(myXOR)
```

Let's test this out:

```
In [43]: XOR(PAIR(_1,NIL)) # List [1]
Out[43]: 1

In [44]: XOR(PAIR(_1,PAIR(_0,PAIR(_1,NIL)))) # List [1,0,1]
Out[44]: 0

In [45]: XOR(makelist(1,0,1))
Out[45]: 0

In [46]: XOR(makelist(1,0,0,1,1))
Out[46]: 1
```