Learning Objectives:

- Formally modeling running time, and in particular notions such as $O(n)$ or $O(n^3)$ time algorithms.
- The classes **P** and **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 *can not* compute in $O(n^k)$ time.
- The class $\mathbf{P}_{/\mathbf{poly}}$ of *non uniform* computation and the result that $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$

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.[1] 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 $T(n)$).[2] 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.

then we will think of F as "efficiently computable", while if any such program requires $2^{\Omega(n)}$ 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 $T : \mathbb{N} \rightarrow \mathbb{N}$ be some function mapping natural numbers to natural numbers. We say that a function $F : \{0,1\}^* \rightarrow \{0,1\}$ 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 $F(x)$. [3]

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 $TIME_{<<}(T(n))$ denote the set of Boolean functions that are computable in $T(n)$ NAND« time, and define $TIME_{++}(T(n))$ analogously. The set $TIME(T(n))$ (without any subscript) corresponds to $TIME_{<<}(T(n))$.

P 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, $TIME_{<<}(T(n))$ and $TIME_{++}(T(n))$ are classes of *functions*, not of *programs*. If P is a NAND++ program then a statement such as "P is a member of $TIME_{++}(n^2)$" 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 `foo =`

[3] The relaxation of considering only "sufficiently large" n's is not very important but it is convenient since it allows us to avoid dealing explicitly with un-interesting "edge cases". In most cases we will anyway be interested in determining running time only up to constant and even polynomial factors. Note that we can always compute a function on a finite number of inputs using a lookup table.

`MACRO(bar)` where `MACRO` is some macro/function that is defined elsewhere using 100 lines of vanilla NAND++, then executing this line counts as executing 100 steps rather than a single one.

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.

R **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 $T : \mathbb{N} \rightarrow \mathbb{N}$ 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 $TIME(T(n))$ 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 $T(n) = \lfloor n \rfloor^{1.2} \log n \rfloor$ 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 $F : \{0,1\}^* \rightarrow \{0,1\}$ is *computable in polynomial time* if it is in the class $\mathbf{P} = \cup_{c \in \{1,2,3,\ldots\}} TIME(n^c)$. That is, $F \in \mathbf{P}$ 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 $F : \{0,1\}^* \rightarrow \{0,1\}$ is *computable in exponential time* if it is in the class $\mathbf{EXP} = \cup_{c \in \{1,2,3,\ldots\}} TIME(2^{n^c})$. That is, $F \in \mathbf{EXP}$ 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 $F : \{0,1\}^* \rightarrow \{0,1\}$. We say that $F \in \mathbf{P}$ if there is a polynomial $p : \mathbb{N} \rightarrow \mathbb{R}$ 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 $F \in \mathbf{EXP}$ if there is a polynomial $p : \mathbb{N} \rightarrow \mathbb{R}$ 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 $F(x)$.

P 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, $\mathbf{P} \subseteq \mathbf{EXP}$. All of the problems we listed in Chapter 11 are in **EXP**,[4] 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**.

P	**EXP** (but not known to be in **P**)
Shortest path	Longest Path
Min cut	Max cut
2SAT	3SAT
Linear eqs	Quad. eqs
Zerosum	Nash
Determinant	Permanent
Primality	Factoring

A table of the examples from Chapter 11. All these problems are in **EXP** but the only the ones on the left column are currently known to be in **P** as well (i.e., they have a polynomial-time algorithm).

[4] 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

12.1.2 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 $\overline{TIME}_{<<}(T(n))$ and $\overline{TIME}_{++}(T(n))$ to be the generalization of $TIME_{<<}(T(n))$ and $TIME_{++}(T(n))$ to functions that may be partial or have more than one bit of output, and define $\overline{\mathbf{P}}$ and $\overline{\mathbf{EXP}}$ 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 $\overline{TIME}_{<<}(T(n))$ (respectively $\overline{TIME}_{++}(T(n))$) if there exists a NAND« (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 $\overline{TIME}(T(n))$ (without a subscript) denote the set $\overline{TIME}_{<<}(T(n))$ and let $\overline{\mathbf{P}} = \cup_{c\in\{1,2,3,\ldots\}} \overline{TIME}(n^c)$ and $\overline{\mathbf{EXP}} = \cup_{c\in\{1,2,3,\ldots\}} \overline{TIME}(2^{n^c})$.

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 $F \in TIME_{<<}(T(n))$ then there is a NAND++ program P' that computes F in $T'(n) = a \cdot T(n)^b$. That is, $TIME_{<<}(T(n)) \subseteq TIME_{++}(aT(n)^b)$

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 **P** and **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 **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

NAND« one by one, until we are left with NAND++.

We will not provide the full details but will present the main ideas used in showing that every feature of NAND« can be simulated by NAND++ with at most a polynomial overhead:

1. Recall that every NAND« 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« 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 `Foo` as a two dimensional *bit* array `Foo_2D` (where `Foo_2D[`i`][`j`]` is the j-th bit of `Foo[`i`]`) and then encode the latter as a *one dimensional* bit array `Foo_1D` by mapping `Foo_2D[`i`][`j`]` to `Foo_1D[`$embed(i,j)$`]` where $embed : \mathbb{N} \times N \rightarrow \mathbb{N}$ 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 $embed(i,j) = \frac{1}{2}(i+j)(i+j+1)+j$ as in Exercise 7.1, then we can see that if $i,j \leq O(T(n))$, then $embed(i,j) \leq O(T(n)^2)$. We also use the fact that $embed$ 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 $poly(\log T(n))$ 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 `Foo` 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 `Foo` until it reaches 0), to ensure that an array `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 `Marker` until we reach the point that `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+= foo` and `i-= 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 $O(T^a)$ 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.) ■

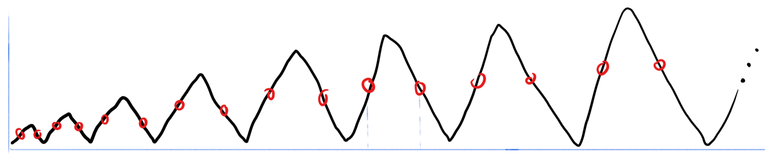

Figure 12.1: The path an index variable takes in a NAND++ program. If we need to simulate T steps of an enhanced NAND++ program using vanilla NAND++ then in the worst case we will need to wait at every time for the next time `i` arrives at the same location, which will yield a cost of $O(1 + 2 + \cdots + T) = O(T^2)$ steps.

R **Turing machines and other models** If we follow the equivalence results between NAND++/NAND« and other models, including Turing machines, RAM machines, Game of life, λ calculus, and many others, then we can see that these results also have at most a polynomial overhead in the simulation in each way.[5] It is a good exercise to go through, for example, the proof of Theorem 6.12 and verify that it establishes that Turing machines and NAND++ programs are equivalent up to polynomial overhead.

[5] For the λ calculus, one needs to be careful about the order of application of the reduction steps, which can matter for computational efficiency, see for example this paper.

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 $F \in \mathbf{P}$ (or whether $F \in \mathbf{EXP}$) 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 $O(T^a)$ 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 $TIMEDEVAL : \{0,1\}^* \rightarrow \{0,1\}^*$ defined as follows:

$$TIMEDEVAL(P, x, 1^T) = P(x) \qquad (12.1)$$

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 $TIMEDEVAL$ 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.).

R **What does 1^T mean?** The function $TIMEDEVAL$ 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 $TIMEDEVAL$ 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 $TIMEDEVAL$ 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.

P Before reading the proof of Theorem 12.6, try to think how you would compute $TIMEDEVAL$ using your favorite programming language. That is, how you would write a program `Timed_Eval(P,x,T)` that gets a NAND« 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« 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« program in full we would need to describe a precise representation scheme, as well as the full NAND« 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« is given in the Appendix, and for the purposes of this simulation, we can simply use the representation of the code NAND« as an ASCII string.

The program U gets as input a NAND« 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 `current_line`, and `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 $var_0, \ldots, var_{t-1}$ then U maintains an array `Var_numbers` that contains a list of

pairs of the form (var_s, s) for $s \in [t]$. Using `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 `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 `Var_values[`s`]`. If it is an array variable then the value of its i-th element will be stored in location `Var_values[`$t \cdot i + s$`]`.

4. To simulate a single step of P, the program U will recover the line corresponding to `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 `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]

[6] We use Python-like syntax in this pseudocode, but it is not valid Python code.

```
def U(P,x,1^T):
    t = number_variable_identifiers(P) # number of
     ↪  distinct identifiers used in P

    L = number_lines(P)

    # denote names of P's variables as var_0,...,
     ↪  var_(t-1)
    Var_numbers = array encoding list [
     ↪  (var_0,0),...,(var_(t-1),t-1)]
    # Var_numbers: encode variable identifiers as
     ↪  number 0...t-1

    Var_values = unbounded array initialized to 0
    # if s in [t] corresponds to scalar then
     ↪  Var_values[s] is value of variable
     ↪  corresponding to s.
    # if s corresponds to array then
     ↪  Var_values[t*i+s] is value of variable
     ↪  corresponding to s at position i
```

```
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 *can not* 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 **EXP**.

[7] There is nothing special about $\log n$, and we could have used any other efficiently computable function that tends to infinity with n.

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. ⋆

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 s.t. 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 $HALT_T(P,x)$ to equal 1 for every NAND« program P such that $|P| \leq \log\log|x|$, and such that P halts on the input x within $100T(|x|)$ steps. $HALT_T$ equals 0 on all other inputs.[8]

[8] The constant 100 and the function $\log\log n$ are rather arbitrary, and are chosen for convenience in this proof.

Theorem 12.7 is an immediate consequence of the following two claims:

Claim 1: $HALT_T \in TIME(T(n)\ logn)$

and

Claim 2: $HALT_T \notin TIME(T(n))$.

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 $HALT_T$ in at most $poly(|P|)T(n)$ 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.

[9] Recall that we use $poly(m)$ to denote a quantity that is bounded by am^b for some constants a, b and every sufficiently large m.

Proof of claim 2: This proof is the heart of Theorem 12.7, and is very reminiscent of the proof that $HALT$ is not computable. Assume, toward the sake of contradiction, that there is some NAND« program P^* that computes $HALT_T(P, x)$ within $T(|P| + |x|)$ steps. We are going to show a contradiction by creating a program Q 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 $HALT_T(Q, Q1^m)$ equals 0 or 1.

On the one hand, if $HALT_T(Q, Q1^m) = 1$, then under our assumption that P^* computes $HALT_T$, 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 $HALT_T(Q, Q1^m) = 1$.

This means that it must hold that $HALT_T(Q, Q1^m) = 0$. 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 $HALT_T(Q, Q1^m) = 1$, 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.

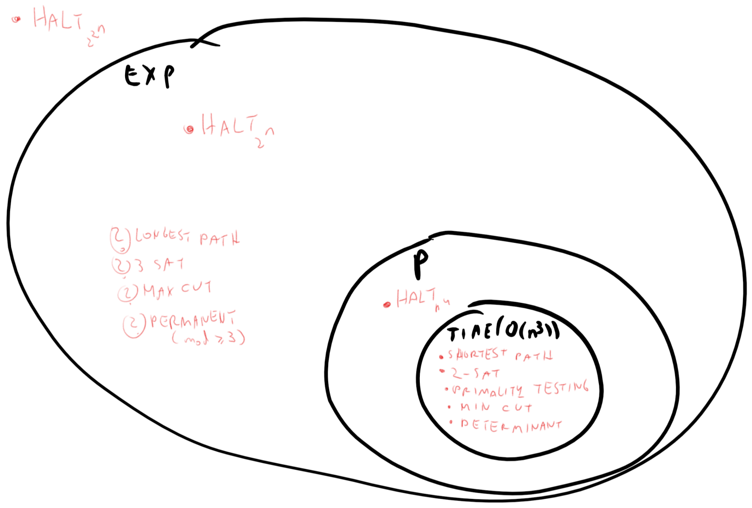

Figure 12.2: Some complexity classes and some of the functions we know (or conjecture) to be contained in them.

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 $G \in SIZE(T)$ 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 $F \in TIME(T(n))$. 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 $TIME_{++}(T(n))$ and every sufficiently large $n \in N$, F_n is in $SIZE(cT(n))$.

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:

```
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:

```
// 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 $T : \mathbb{N} \rightarrow \mathbb{N}$ 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:

```
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`$\langle$`i<-`$j\rangle$ the NAND program that is obtained by replacing all references of the form `Foo[i]` (which are allowed in NAND++, but illegal in NAND that has no index variable **`i`**) with references of the form `Foo[`j`]` (which are allowed in NAND, since j is simply a number). Whenever we see a reference to the variable **`Xvalid`**`[`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 `IF (`**`loop`**`) P`$\langle$**`i`**`<-`$j\rangle$ for $\lfloor T(n)/L \rfloor - 1$ times, replacing each time j by $0, 1, 0, 1, 2, \ldots$ as in the definition of (standard or "vanilla") NAND++ in Section 6.1.3. We replace `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 `IF (`**`loop`**`) P`$\langle$**`i`**`<-`$j\rangle$ corresponds to at most $4L$ lines of standard sugar-free NAND. Thus the total cost is at most $4L \cdot (\frac{T(n)}{L}) \leq 4 \cdot T(n)$ lines.[10] ■

[10] The constant 4 can be improved, but this does not really make much difference.

By combining Theorem 12.8 with Theorem 12.5, we get that if $F \in TIME(T(n))$ then there are some constants a, b such that for every large enough n, $F_n \in SIZE(aT(n)^b)$. (In fact, by direct inspection of the proofs we can see that $a = b = 5$ 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 $COMPILE$ 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 $COMPILE$. (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 the property that it only modifies the value of the `Y[0]` variable once. (A property that can be ensured by adding appropriate flag variables and some `IF` syntactic sugar.)

```
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 $0, 1, 0, 1, 2, 1, 0, 1, 2, \ldots$ 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 = NAND PP program
    T - time bound, n - number of inputs, m - number
 ↪  of outputs
    Produces NAND program of O(T) lines that
 ↪  computes
    the restriction of P to inputs of length n and T
 ↪  steps
    '''
    lines = [l for l in P.split('\n') if l] # lines
     ↪  of P

    # initialization
    result = r'''
temp = NAND(X[0],X[0])
one = NAND(X[0],temp)
zero = NAND(one,one)
nothalted = NAND(X[0],temp)
halted = NAND(one,one)
'''[1:]

    # assign_to = IF(halted,assign_to,new_value)
    IFCODE = r'''
iftemp_0 = NAND(new_value,nothalted)
iftemp_1 = NAND(assign_to,halted)
assign_to = NAND(iftemp_0,iftemp_1)
'''[1:]

    UPDATEHALTED = r'''
halted = NAND(nothalted,loop)
nothalted = NAND(halted,halted)
    '''[1:]

    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')
            line = line.replace('[i]',f'[{j}]')
            idx = line.find("=")
            lefthand = line[:idx].strip()
            righthand = line[idx+1:].strip()
            result += "new_value = " + righthand +
             ↪  "\n"
            result +=
             ↪  IFCODE.replace("assign_to",lefthand)
        result += UPDATEHALTED

    return result
```

Since NAND« programs can be simulated by NAND++ programs with polynomial overhead, we see that we can simulate a $T(n)$ time NAND« program on length n inputs with a $poly(T(n))$ size NAND program.

To make sure you understand this transformation, it is an excellent exercise to verify the following equivalent characterization of the class **P** (see Exercise 12.6). Prove that for every $F : \{0,1\}^* \rightarrow \{0,1\}$, $F \in$ **P** if and only if there is a polynomial-time NAND++ (or NAND«, it doesn't matter) program P such that for every $n \in \mathbb{N}$, $P(1^n)$ outputs a description of an n input NAND program Q_n that computes the restriction F_n of F to inputs in $\{0,1\}^n$. (Note that since P runs in polynomial time and hence has an output of at most polynomial length, Q_n has at most a polynomial number of lines.)

12.5.2 The class $\mathbf{P}_{/\mathbf{poly}}$

We can define the "non uniform" analog of the class **P** as follows:

Definition 12.10 — $\mathbf{P}_{/\mathbf{poly}}$. For every $F : \{0,1\}^* \rightarrow \{0,1\}$, we say that $F \in \mathbf{P}_{/\mathbf{poly}}$ if there is some polynomial $p : \mathbb{N} \rightarrow \mathbb{R}$ such that for every $n \in \mathbb{N}$, $F_n \in SIZE(p(n))$ where F_n is the restriction of F to inputs in $\{0,1\}^n$.

Theorem 12.8 implies that $\mathbf{P} \subseteq \mathbf{P}_{/\mathbf{poly}}$.

Please make sure you understand why this is the case.

Using the equivalence of NAND programs and Boolean circuits, we can also define $P_{/poly}$ as the class of functions $F : \{0,1\}^* \rightarrow \{0,1\}$ such that the restriction of F to $\{0,1\}^n$ is computable by a Boolean circuit of $poly(n)$ size (say with gates in the set $\wedge, \vee, \neg$ though any universal gateset will do); see Fig. 12.3.

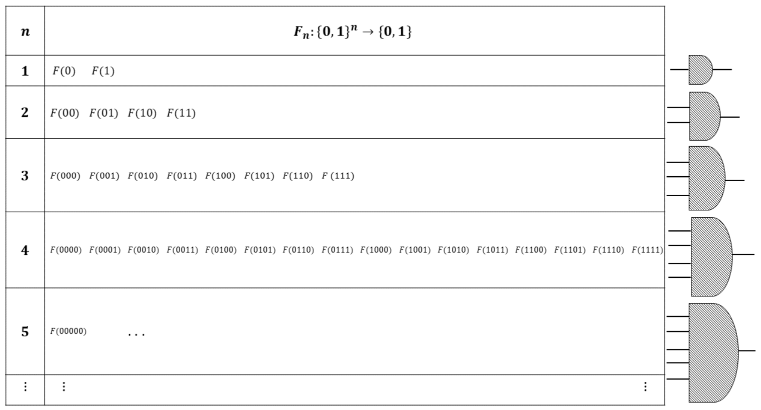
n
$F_n : \{0,1\}^n \to \{0,1\}$
1
$F(0)$ $F(1)$
2
$F(00)$ $F(01)$ $F(10)$ $F(11)$
3
$F(000)$ $F(001)$ $F(010)$ $F(011)$ $F(100)$ $F(101)$ $F(110)$ $F(111)$
4
$F(0000)$ $F(0001)$ $F(0010)$ $F(0011)$ $F(0100)$ $F(0101)$ $F(0110)$ $F(0111)$ $F(1000)$ $F(1001)$ $F(1010)$ $F(1011)$ $F(1100)$ $F(1101)$ $F(1110)$ $F(1111)$
5
$F(00000)$ ...
⋮

Figure 12.3: We can think of an infinite function $F : \{0,1\}^* \to \{0,1\}$ as a collection of finite functions $F_0, F_1, F_2, \ldots$ where $F_n : \{0,1\}^n \to \{0,1\}$ is the restriction of F to inputs of length n. We say F is in $\mathbf{P}_{/\mathbf{poly}}$ if for every n, the function F_n is computable by a polynomial size NAND program, or equivalently, a polynomial sized Boolean circuit. (We drop in this figure the "edge case" of F_0 though as a constant function, it can always be computed by a constant sized NAND program.)

The notation $\mathbf{P}_{/\mathbf{poly}}$ 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 — $\mathbf{P}_{/\mathbf{poly}}$ characterization by advice. Let $F : \{0,1\}^* \to \{0,1\}$. Then $F \in \mathbf{P}_{/\mathbf{poly}}$ 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 $F \in \mathbf{P}_{/\mathbf{poly}}$ 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 $NANDEVAL$ 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 $TIME(T(n))$ is in $SIZE(poly(T(n)))$. 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 $TIME(T(n))$ 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 $UH(x) = HALTONZERO(S(|x|))$. 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 $HALTONZERO$ by transforming the input program P into the integer n such that $P = S(n)$ and then 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 $F : \{0,1\}^* \rightarrow \{0,1\}$, 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 $\mathbf{P_{/poly}}$ 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 $\mathbf{P} \subseteq \mathbf{P_{/poly}}$, 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".

R **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 **P** we discussed before: longest path, 3SAT, factoring, etc., these problems are also not known to be in $\mathbf{P_{/poly}}$ 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 $F : \{0,1\}^* \rightarrow \{0,1\}$ and some nice time bound $T : \mathbb{N} \rightarrow \mathbb{N}$, we know that:

- If F is computable in time $T(n)$ then there is a sequence $\{P_n\}$ of NAND programs with $|P_n| = poly(T(n))$ such that P_n computes F_n (i.e., restriction of F to $\{0,1\}^n$) for every n.
- The reverse direction is not necessarily true - there are examples of functions $F : \{0,1\}^n \rightarrow \{0,1\}$ such that F_n can be computed by even a constant size NAND program but F is uncomputable.

This means that non uniform complexity is more useful to establish *hardness* of a function than its *easiness*.

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 $F : \{0,1\}^* \rightarrow \{0,1\}$ 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

[11] 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.

"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.[12] 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 $poly(T(n))$

12.7 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 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 $\cup_{c\in\{1,2,3,\ldots\}} TIME(n^c)$ and $\cup_{c\in\{1,2,3,\ldots\}} TIME(2^{n^c})$ respectively. (If $S_1, S_2, S_3, \ldots$ is a collection of sets then the set $S = \cup_{c\in\{1,2,3,\ldots\}} S_c$ 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,σ) with $x \in \{0,1\}^*$, $i \in \mathbb{N}$ and $\sigma \in \{0,1\}$,

$$Bool(F)(x,i,\sigma) = \begin{cases} F(x)_i & \sigma = 0, i < |F(x)| \\ 1 & \sigma = 1, i < |F(x)| \\ 0 & \text{otherwise} \end{cases} \qquad (12.2)$$

where $F(x)_i$ is the i-th bit of the string $F(x)$.

Prove that $F \in \overline{\mathbf{P}}$ if and only if $Bool(F) \in \mathbf{P}$. ■

Exercise 12.3 — Composition of polynomial time. Prove that if $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$ are in $\overline{\mathbf{P}}$ then their *composition* $F \circ G$, which is the function H s.t. $H(x) = F(G(x))$, is also in $\overline{\mathbf{P}}$. ■

Exercise 12.4 — Non composition of exponential time. Prove that there is some $F, G : \{0,1\}^* \rightarrow \{0,1\}^*$ s.t. $F, G \in \overline{\mathbf{EXP}}$ but $F \circ G$ is not in **EXP**.[13] ■

[13] TODO: check that this works, idea is that we can do bounded halting.

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.[14] Let $F : \{0,1\}^* \rightarrow \{0,1\}^*$ be such that there is some function $m : \mathbb{N} \rightarrow \mathbb{N}$ satisfying $|F(x)| = m(|x|)$ for every x, and let P be a NAND++ program that computes F in $T(n)$ time for some nice T. Then there is an *oblivious* NAND++ program P' that computes F in time $O(T^2(n)\log T(n))$. ■

[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 $F : \{0,1\}^* \rightarrow \{0,1\}$, $F \in \mathbf{P}$ 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

15

[15] TODO: add reference to best algorithm for longest path - probably the Bjorklund algorithm

12.9 FURTHER EXPLORATIONS

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

12.10 ACKNOWLEDGEMENTS