Learning Objectives:

- Explore the consequences of **P** $=$ **NP**
- *Search-to-decision* reduction: transform algorithms that solve decision version to search version for **NP**-complete problems.
- Optimization and learning problems
- Quantifier elimination and solving polynomial hiearchy.
- What is the evidence for **P** $=$ **NP** vs **P** $\neq$ **NP**?

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** $\neq$ **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]

[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** $=$ **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.

We have mentioned that the question of whether **P** $=$ **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** $=$ **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,

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 $TIME(cn)$ or $TIME(cn^2)$ 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]

[3] Using the relations we've seen between $SIZE(T(n))$ and $TIME(T(n))$ (i.e., Theorem 12.8), $3SAT \notin SIZE(T(n))$ then it is also in $TIME(T(n)^\epsilon)$ for some constant ϵ that can be shown to be at least $1/5$.

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 $\mathbf{P} = \mathbf{NP}$. Then for every polynomial-time algorithm V and $a, b \in \mathbb{N}$,there is a polynomial-time algorithm $FIND_V$ such that for every $x \in \{0,1\}^n$, if there exists $y \in \{0,1\}^{an^b}$ satisfying $V(xy) = 1$, then $FIND_V(x)$ 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 polyomial-time algorithm $VERIFYCUT$ such that $VERIFYCUT(G,k,S) = 1$ 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** $=$ **NP** then there is a polynomial-time algorithm $FINDCUT$ that on input G,k outputs a set S such that $VERIFYCUT(G,k,S) = 1$ if such a set exists. This means that if **P** $=$ **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** $=$ **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 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 **NP**-complete, we can reduce the task of solving the search problem for $MAXCUT$ or any other problem in **NP** to the task of solving it for $3SAT$.) Suppose that **P** $=$ **NP** and we are given a satisfiable 3CNF formula φ, and we now want to find a satisfying assignment y for φ. Define $3SAT_0(\varphi)$ to output 1 if there is a satisfying assignment y for φ 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 **NP**, and so if **P** $=$ **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** $=$ **NP** then for every polynomial-time algorithm V and $a,b \in \mathbb{N}$, there is a polynomial-time algorithm $STARTSWITH_V$ that on input $x \in \{0,1\}^*$ and $z \in \{0,1\}^\ell$, outputs 1 if and only if there exists some $y \in \{0,1\}^{an^b}$ such that the first ℓ bits of y are equal to z and $V(xy) = 1$. Indeed, we leave it as an exercise to verify that the $STARTSWITH_V$ function is in **NP** and hence can be solved in polynomial time if **P** $=$ **NP**.

Now for any such polynomial-time V and $a,b \in \mathbb{N}$, we can implement $FIND_V(x)$ as follows:

Algorithm $FIND_V$:

Input: $x \in \{0,1\}^n$

Goal: Find $z \in \{0,1\}^{an^b}$ such that $V(xz) = 1$, if such z exists.

Operation:

1. For $\ell = 0, \ldots, an^b - 1$ do the following:
   (a) Let $b_0 = STARTSWITH_V(xz_0 \cdots z_{\ell-1}0)$ and $b_1 = STARTSWITH_V(xz_0 \cdots z_{\ell-1}1)$
   (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 $STARTSWITH_V$ 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 ℓ 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 $STARTSWITH_V(xz_0 \cdots z_{\ell-1}0)$ returns 1, then we are guaranteed the invariant by definition of $STARTSWITH_V$. 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 $STARTSWITH_V(xz_0 \cdots z_{\ell-1}0)$ 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 **NP** problems if **P** = **NP**, but it is not immediately clear that we can find the *optimal* solution. For example, suppose that **P** = **NP**, and you are given a graph G. Can you find the *longest* simple path in G in polynomial time?

This is actually an excellent question for you to attempt on your own. That is, assuming **P** = **NP**, give a polynomial-time algorithm that on input a graph G, outputs a maximally long simple path in the graph G.

It turns out the answer is *Yes*. The idea is simple: if **P** = **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

result:

Theorem 15.2 — Optimization from P = NP. Suppose that **P** = **NP**. Then for every polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ there is a polynomial-time algorithm OPT such that on input $x \in \{0,1\}^*$, $OPT(x, 1^m) = \max_{y \in \{0,1\}^m} f(x,y)$ (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 $FINDOPT$ such that for every $x \in \{0,1\}^*$, $FINDOPT(x, 1^m)$ outputs $y^* \in \{0,1\}^*$ such that $f(x, y^*) = OPT(x, y^*)$.

P 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 **P** = **NP** then for every for every string $x \in \{0,1\}^*$ and number k, we can test in in $poly(|x|, m)$ 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 $poly(|x| + |y|)$ (as is the case in the example of longest path) then we can just try out all possiblities 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 $F : \{0,1\}^* \rightarrow \{0,1\}$ as follows.

$$F(x, 1^m, k) = \begin{cases} 1 & \exists_{y \in \{0,1\}^m} f(x,y) \geq k \\ 0 & \text{otherwise} \end{cases} \tag{15.1}$$

Since f is computable in polynomial time, F is in **NP**, and so under our assumption that **P** $=$ **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** $=$ **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 $-M$.)

R **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 agreas 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* $x_0, x_1, \ldots, x_{m-1}$ (where we can think of each x_i as a string in $\{0,1\}^n$ for some n) and the *labels* for them $y_0, \ldots, y_{n-1}$ (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* $h : \{0,1\}^n \to \{0,1\}$ 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 $H : \{0,1\}^{k+n} \to \{0,1\}$. 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 $S = \{(x_0, y_0), \ldots, (x_{m-1}, y_{m-1})\}$, the set of parameters $\theta_0, \ldots, \theta_{k-1} \in \{0,1\}$ that minimizes the number of errors made by the predictor $x \mapsto H(\theta, x)$.[4]

[4] This is often known as Empirical Risk Minimization.

In other words, we can define for every set S as above the function $F_S : \{0,1\}^k \to [m]$ such that $F_S(\theta) = \sum_{(x,y)\in S} |H(\theta, x) - y|$. Now, finding the value θ that minimizes $F_S(\theta)$ is equivalent to solving the supervised learning problem with respect to H. For every polynomial-time computable $H : \{0,1\}^{k+n} \to \{0,1\}$, 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 θ to a "ciphertext" c, then given examples of ciphertext/plaintext pairs of the form $(c_0, p_0), \ldots, (c_{m-1}, p_{m-1})$, our goal is to find the key θ 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 $w^0, \ldots, w^m$ (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 $poly(n)$ (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 $SHORTPROOF_V : \{0,1\}^* \to \{0,1\}$ is in **NP**, where for every input of the form $x1^m$, $SHORTPROOF_V(x,1^m) = 1$ if and only if there exists $w \in \{0,1\}^*$ with $|w| \leq m$ s.t. $V(xw) = 1$. That is, $SHORTPROOF_V(x,1^m) = 1$ if there is a proof (in the system V) of length at most m bits that x is true. Thus, if $\mathbf{P} = \mathbf{NP}$, 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 $SHORTPROOF_V$ 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 **NP** or even "polynomial time" was formally

defined):

> 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 $\varphi(n) = \max_F \psi(F,n)$. 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, [5] 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), $SHORTPROOF_V$ is in fact **NP**-complete, and so Gödel can be thought of as the first person to formulate the **P** vs **NP** question. Unfortunately, the letter was only discovered in 1988. [6]

[5] 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.

[6] TODO: Maybe add example on finding Nash equilibrium

15.4 QUANTIFIER ELIMINATION (ADVANCED)

If **P** = **NP** then we can solve all **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 **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

$$\exists_{y \in \{0,1\}^{p(|x|)}} V(xy) = 1 \tag{15.2}$$

for some polynomial-time algorithm V and polynomial $p : \mathbb{N} \to \mathbb{N}$. 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

$$\exists_{y \in \{0,1\}^{p_0(|x|)}} \forall_{z \in \{0,1\}^{p_1(|x|)}} V(xyz) = 1 \;, \tag{15.3}$$

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

$$\exists_{y \in \{0,1\}^{p_0(|x|)}} \forall_{z \in \{0,1\}^{p_1(|x|)}} \exists_{w \in \{0,1\}^{p_2(|x|)}} V(xyzw) = 1 \tag{15.4}$$

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

$$\exists_{P' \in \{0,1\}^s} \forall_{x \in \{0,1\}^n} P(x) = P'(x) \tag{15.5}$$

which has the form Eq. (15.3).[7] Another example of a statement involving a levels of quantifiers would be to check, given a chess position x, whether there is a strategy that guarantees that White wins within a steps. For example is $a = 3$ we woud want to check if given the borad 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

$$\exists_{y_0 \in \{0,1\}^m} \forall_{y_1 \in \{0,1\}^m} \cdots \mathcal{Q}_{y_{a-1} \in \{0,1\}^m} V(xy_0y_1 \cdots y_{a-1}) = 1 \tag{15.6}$$

where $m = p(n)$ and $\mathcal{Q}$ 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 complelling 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 **NP** (as we assumed V' can be computed in polynomial time) and hence under the assumption that **P** $=$ **NP** we can determine in polynomial time whether the condition $(*)$, and hence our original statement, is true. ⋆

Proof of Theorem 15.4. We prove the theorem by induction. We assume that there is a polynomial-time algorithm $SOLVE_{V,a-1}$ that can solve the problem Eq. (15.6) for $a-1$ and use that to solve the problem for a. For $a = 1$, $SOLVE_{V,a-1}(x) = 1$ iff $V(x) = 1$ which is a polynomial-time computation since V runs in polynomial time. For every x, y_0, define the statement φ_{x,y_0} to be the following:

$$\varphi_{x,y_0} = \forall_{y_1 \in \{0,1\}^m} \exists_{y_2 \in \{0,1\}^m} \cdots Q_{y_{a-1} \in \{0,1\}^m} V(xy_0y_1 \cdots y_{a-1}) = 1 \quad (15.7)$$

By the definition of $SOLVE_{V,a}$, for every $x \in \{0,1\}^n$, our goal is that $SOLVE_{V,a}(x) = 1$ if and only if there exists $y_0 \in \{0,1\}^m$ such that φ_{x,y_0} is true.

The *negation* of φ_{x,y_0} is the statement

$$\overline{\varphi}_{x,y_0} = \exists_{y_1 \in \{0,1\}^m} \forall_{y_2 \in \{0,1\}^m} \cdots \overline{Q}_{y_{a-1} \in \{0,1\}^m} V(xy_0y_1 \cdots y_{a-1}) = 0 \quad (15.8)$$

where $\overline{Q}$ is $\exists$ if Q was $\forall$ and $\overline{Q}$ is $\forall$ otherwise. (Please stop and verify that you understand why this is true, this is a generalization of the fact that if Ψ 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 φ_{x,y_0} is true. Hence we can rephrase the original statement Eq. (15.6) as follows:

$$\exists_{y_0 \in \{0,1\}^m} S(xy_0) = 1 \quad (15.9)$$

but since S is a polynomial-time algorithm, Eq. (15.9) is clearly a statement in **NP** and hence under our assumption that **P** $=$ **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]

[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.

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 $10^6 n$ 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 φ described by a string of length at most n, if $C(\varphi) = 1$ then *there exists* an assignment x to the variables of φ 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 $V : \{0,1\}^* \to \{0,1\}$ be some polynomial-time algorithm, and suppose that **P** $=$ **NP**. Then there exists an algorithm $COUNT_V$ that on input $x, 1^m, \epsilon$, runs in time polynomial in $|x|, m, 1/\epsilon$ and outputs a number

$K \in \{0, \ldots, 2^m\}$ such that

$$(1-\epsilon)K \leq \left|\{y \in \{0,1\}^m \ : \ V(xy) = 1\}\right| \leq (1+\epsilon)K \qquad (15.10)$$

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.

P 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 ± 0.01 factor equal to the number of simple paths in G of langth 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 $\mathbf{P} \neq \mathbf{NP}$.

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 $\mathbf{NP} = \mathbf{P}$, 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.[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:

> *"the laws of nature have this amazing feeling of inevitability… which is associated with local perfection."*

> *"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*

[10] One interesting theory is that $\mathbf{P} = \mathbf{NP}$ and evolution has already discovered this algorithm, which we are already using without realizing it. At the moment, there seems to be very little evidence for such a scenario. In fact, we have some partial results in the other direction showing that, regardless of whether $\mathbf{P} = \mathbf{NP}$, many types of "local search" or "evolutionary" algorithms require exponential time to solve 3SAT and other **NP**-hard problems.

> *major challenges in trying to extend our understanding of physics. We don't have these knobs, and little wheels, and twiddles that we can turn. We have to learn how to make these jumps. And it is a tall order. And that's why things are difficult."*

Finding an efficient algorithm for **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 **NP** problems would bring about a *new type of understanding*. In many of the areas where **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 **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 **NP** is currently a barrier for analytical understanding. Such new insights would be very fruitful regardless of their computational utility.

15.7 CAN $\mathrm{P} \neq \mathrm{NP}$ 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]

[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.

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 $\mathbf{P} \neq \mathbf{NP}$?

In short, the answer is *No*. For example, suppose that we are trying to decide between the "3SAT is easy" conjecture (there is an $10^6 n$ 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 $n = 10^8$, $2^{10^{-6}n} > 10^6 n$, this boils down to the finite question of deciding whether or not there is a 10^{13}-line NAND

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 verification is the (finite in principle though infeasible in practice) process of checking that it succeeds on all inputs.[13] If there isn't such a program, 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.

[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 statements. In contrast, the inefficiency of verifying *failure* of all programs could well be inherent.

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 example, if all the clauses in a 3SAT instance φ contain the same variable x_i (possibly in negated form), then by guessing a value to x_i we can reduce φ to a 2SAT instance which can then be efficiently solved. Generalizations 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.[14] 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 machines) seem to regularly succeed in solving interesting instances of **NP**-hard problems, typically by using some combination of guessing while making local improvements.

[14] Actually, the computational difficulty of problems in economics such as finding optimal (or any) equilibria is quite subtle. Some variants of such problems are **NP**-hard, while others have a certain "intermediate" complexity.

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 application areas, whether it is scientific computing, optimization, control 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

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

[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 apostoriori probability.

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

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

16

[16] TODO: Scott's two surveys

15.12 FURTHER EXPLORATIONS

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

- Polynomial hiearchy hardness for circuit minimization and related problems, see for example this paper.

15.13 ACKNOWLEDGEMENTS