
Computational Complexity: A Modern Approach

Draft of a book: Dated January 2007
Comments welcome!

Sanjeev Arora and Boaz Barak
Princeton University
complexitybook@gmail.com

Not to be reproduced or distributed without the authors' permission

This is an Internet draft. Some chapters are more finished than others. References and attributions are very preliminary and we apologize in advance for any omissions (but hope you will nevertheless point them out to us).

Please send us bugs, typos, missing references or general comments to
complexitybook@gmail.com — Thank You!!

Chapter 2

NP and NP completeness

“(if $\phi(n) \approx Kn^2$)^a then this would have consequences of the greatest magnitude. That is to say, it would clearly indicate that, despite the unsolvability of the (Hilbert) Entscheidungsproblem, the mental effort of the mathematician in the case of the yes-or-no questions would be completely replaced by machines.... (this) seems to me, however, within the realm of possibility.”

Kurt Gödel in a letter to John von Neumann, 1956

^aIn modern terminology, if SAT has a quadratic time algorithm

“I conjecture that there is no good algorithm for the traveling salesman problem. My reasons are the same as for any mathematical conjecture: (1) It is a legitimate mathematical possibility, and (2) I do not know.”

Jack Edmonds, 1966

“In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually.”

Richard Karp, 1972

If you have ever attempted a crossword puzzle, you know that it is much harder to solve it from scratch than to verify a solution provided by someone else. Likewise, solving a math homework problem by yourself is usually much harder than reading and understanding a solution provided by your instructor. The usual explanation for this difference of effort is that finding a solution to a crossword puzzle, or a math problem, requires *creative effort*. Verifying a solution is much easier since somebody else has already done the creative part.

This chapter studies the computational analog of the above phenomenon. In Section 2.1 we define a complexity class **NP** that aims to capture the set of problems whose solutions can be efficiently *verified*. By contrast, the class **P** of the previous chapter contains decision problems that can be efficiently *solved*. The famous **P** versus **NP** question asks whether or not the two classes are the same. If the answer is yes, then computers can efficiently (i.e., in polynomial time) solve many tasks that we usually think of as requiring “creativity” (see Section 2.7.3). Thus resolving the **P** versus **NP** question is of great practical, scientific and philosophical interest.

In Section 2.2 we introduce the important phenomenon of **NP-completeness**. A very large set of interesting problems are known to be **NP-complete**, implying they have a polynomial algorithm if and only if **P = NP**. The study of **NP-completeness** involves *reductions*, a basic notion used to relate the computational complexity of two different problems. This notion and its various siblings will often reappear in later chapters (e.g., in chapters 7, 16 and 17). The framework of ideas introduced in this chapter motivates much of the rest of this book.

2.1 The class NP

Now we formalize the intuitive notion of *efficiently verifiable solutions* by defining a complexity class **NP**.

DEFINITION 2.1 (THE CLASS NP)

A language $L \subseteq \{0, 1\}^*$ is in **NP** if there exists a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM M (called the *verifier* for L) such that for every $x \in \{0, 1\}^*$,

$$x \in L \Leftrightarrow \exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1$$

If $x \in L$ and $u \in \{0, 1\}^{p(|x|)}$ satisfy $M(x, u) = 1$ then we call u a *certificate* for x (with respect to the language L and machine M).

Some texts use the term *witness* instead of certificate.

EXAMPLE 2.2 (INDSET \in NP)

To get a sense for the definition, we show that the **INDSET** language defined in Example 0.1 is in **NP**. Recall that this language contains all pairs $\langle G, k \rangle$ such that the graph G has a subgraph of at least k vertices with no edges between them (such a subgraph is called an *independent set*). Consider the following polynomial-time algorithm M : given a pair $\langle G, k \rangle$ and a string $u \in \{0, 1\}^*$, output 1 if and only if u encodes a list of k vertices of G such that there is no edge between any two members of the list. Clearly, $\langle G, k \rangle$ is in **INDSET** if and only if there exists a string u such that $M(\langle G, k \rangle, u) = 1$ and hence **INDSET** is in **NP**. The list u of k vertices forming the independent set in G serves as the *certificate* that $\langle G, k \rangle$ is in **INDSET**. Note that if n is the number of vertices in G then a list of k vertices can be encoded using $O(k \log n)$ bits where n is the number of vertices in G . Thus u is a string of at most $O(n \log n)$ bits, which is polynomial in the size of the representation of G .

EXAMPLE 2.3

Here are a few additional examples for decision problems in **NP** (see also Exercise 2.2):

Traveling salesperson: Given a set of n nodes, $\binom{n}{2}$ numbers $d_{i,j}$ denoting the distances between all pairs of nodes, and a number k , decide if there is a closed circuit (i.e., a “salesperson tour”) that visits every node exactly once and has total length at most k . The certificate is the sequence of nodes in the tour.

Subset sum: Given a list of n numbers A_1, \dots, A_n and a number T , decide if there is a subset of the numbers that sums up to T . The certificate is the list of members in this subset.

Linear programming: Given a list of m linear inequalities with rational coefficients over n variables u_1, \dots, u_n (a linear inequality has the form $a_1 u_1 + a_2 u_2 + \dots + a_n u_n \leq b$ for some coefficients a_1, \dots, a_n, b), decide if there is an assignment of rational numbers to the variables u_1, \dots, u_n that satisfies all the inequalities. The certificate is the assignment (see Exercise [np:ex:linprog]).

0/1 Integer programming: Given a list of m linear inequalities with rational coefficients over n variables u_1, \dots, u_m , find out if there is an assignment of zeroes and ones to u_1, \dots, u_n satisfying all the inequalities. The certificate is the assignment.

Graph isomorphism: Given two $n \times n$ adjacency matrices M_1, M_2 , decide if M_1 and M_2 define the same graph, up to renaming of vertices. The certificate is the permutation $\pi : [n] \rightarrow [n]$ such that M_2 is equal to M_1 after reordering M_1 's indices according to π .

Composite numbers: Given a number N decide if N is a composite (i.e., non-prime) number. The certificate is the factorization of N .

Factoring: Given three numbers N, L, U decide if N has a prime factor P in the interval $[L, U]$. The certificate is the factor P .¹

Connectivity: Given a graph G and two vertices s, t in G , decide if s is connected to t in G . The certificate is the path from s to t .

In the above list, the **Connectivity**, **Composite Numbers** and **Linear programming** problems are known to be in **P**. For connectivity this follows from the simple and well known breadth-first search algorithm (see any algorithms text such as [KT06, CLRS01]). The composite numbers problem was only recently shown to be in **P** (see the beautiful algorithm of [AKS??]). For the linear programming problem this is again highly non-trivial, and follows from the Ellipsoid algorithm of Khachiyan [khachiyan??].

All the other problems in the list are not known to be in **P**, though we do not have any proof that they are not in **P**. The **Independent Set** (INDSET), **Traveling Salesperson**, **Subset Sum**, and **Integer Programming** problems are known to be **NP-complete**, which, as we will see in Section 2.2, implies that they are not in **P** unless **P** = **NP**. The **Graph Isomorphism** and **Factoring** problems are not known to be either in **P** nor **NP-complete**.

2.1.1 Relation between NP and P

We have the following trivial relationships between **NP** and the classes **P** and **DTIME**($T(n)$) of Chapter 1 (see Definitions 1.12 and 1.13):

CLAIM 2.4

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c}).$$

PROOF: (**P** \subseteq **NP**): Suppose $L \in \mathbf{P}$ is decided in polynomial-time by a TM N . Then $L \in \mathbf{NP}$ since we can take N as the machine M in Definition 2.1 and make $p(x)$ the zero polynomial (in other words, u is an empty string).

(**NP** $\subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$): If $L \in \mathbf{NP}$ and $M, p()$ are as in Definition 2.1 then we can decide L in time $2^{O(p(n))}$ by enumerating all possible strings u and using M to check whether u is a valid certificate for the input x . The machine accepts iff such a u is ever found. Since $p(n) = O(n^c)$ for some $c > 1$ the number of choices for u is $2^{O(n^c)}$, and the running time of the machine is similar. Thus the theorem is proven. ■

Currently, we do not know of any stronger relation between **NP** and deterministic time classes than the trivial ones stated in Claim 2.4. The question whether or not **P** = **NP** is considered *the* central open question of complexity theory, and is also an important question in mathematics and science at large (see Section 2.7). Most researchers believe that **P** \neq **NP** since years of effort have failed to yield efficient algorithms for certain **NP** languages.

¹There is a polynomial-time algorithm to check primality [AKS??]. We can also show that **Factoring** is in **NP** by using the primality certificate of Exercise 2.4.

2.1.2 Non-deterministic Turing machines.

The class **NP** can also be defined using a variant of Turing machines called *non-deterministic* Turing machines (abbreviated NDTM). In fact, this was the original definition and the reason for the name **NP**, which stands for *non-deterministic polynomial-time*. The only difference between an NDTM and a standard TM (as Defined in Section 1.1.1) is that an NDTM has *two* transition functions δ_0 and δ_1 , and a special state denoted by q_{accept} . When an NDTM M computes a function, we envision that at each computational step M makes an arbitrary choice as to which of its two transition functions to apply. We say that M outputs 1 on a given input x if there is *some* sequence of these choices (which we call the *non-deterministic choices* of M) that would make M reach q_{accept} on input x . Otherwise— if *every* sequence of choices makes M halt without reaching q_{accept} — then we say that $M(x) = 0$. We say that M runs in $T(n)$ time if for every input $x \in \{0, 1\}^*$ and every sequence of non-deterministic choices, M reaches either the halting state or q_{accept} within $T(|x|)$ steps.

DEFINITION 2.5

For every function $T : \mathbb{N} \rightarrow \mathbb{N}$ and $L \subseteq \{0, 1\}^*$, we say that $L \in \mathbf{NTIME}(T(n))$ if there is a constant $c > 0$ and a $cT(n)$ -time NDTM M such that for every $x \in \{0, 1\}^*$, $x \in L \Leftrightarrow M(x) = 1$

The next theorem gives an alternative characterization of **NP** as the set of languages computed by polynomial-time *non-deterministic* Turing machines:

THEOREM 2.6

$$\mathbf{NP} = \bigcup_{c \in \mathbb{N}} \mathbf{NTIME}(n^c)$$

PROOF: The main idea is that the sequence of nondeterministic choices made by an accepting computation of an NDTM can be viewed as a certificate that the input is in the language, and vice versa.

Suppose $p : \mathbb{N} \rightarrow \mathbb{N}$ is a polynomial and L is decided by a NDTM N that runs in time $p(n)$. For every $x \in L$, there is a sequence of nondeterministic choices that makes N reach q_{accept} on input x . We can use this sequence as a *certificate* for x . This certificate has length $p(|x|)$ and can be verified in polynomial time by a *deterministic* machine, which checks that N would have entered q_{accept} after using these nondeterministic choices. Thus $L \in \mathbf{NP}$ according to Definition 2.1.

Conversely, if $L \in \mathbf{NP}$ according to Definition 2.1, then we describe a polynomial-time NDTM N that decides L . On input x , it uses the ability to make non-deterministic choices to write down a string u of length $p(|x|)$. (Concretely, this can be done by having transition δ_0 correspond to writing a 0 on the tape and transition δ_1 correspond to writing a 1.) Then it runs the deterministic verifier M of Definition 2.1 to verify that u is a valid certificate for x , and if so, enters q_{accept} . Clearly, N enters q_{accept} on x if and only if a valid certificate exists for x . Since $p(n) = O(n^c)$ for some $c > 1$, we conclude that $L \in \mathbf{NTIME}(n^c)$. ■

As is the case with deterministic TM's, NDTM's can be easily represented as strings and there exists a *universal* non-deterministic Turing machine (see Exercise 2.5). In fact, using non-determinism we can even make the simulation by a universal TM slightly more efficient.

One should note that, unlike standard TMs, NDTMs are not intended to model any physically realizable computation device.

2.2 Reducibility and NP-completeness

It turns out that the independent set problem is at least as hard as any other language in **NP**: if it has a polynomial-time algorithm then so do all the problems in **NP**. This fascinating property is called **NP-hardness**. Since most scientists conjecture that $\mathbf{NP} \neq \mathbf{P}$, the fact that a language is **NP-hard** can be viewed as evidence that it cannot be decided in polynomial time.

How can we prove that a language C is at least as hard as some other language B ? The crucial tool we use is the notion of a *reduction* (see Figure 2.1):

DEFINITION 2.7 (REDUCTIONS, \mathbf{NP} -HARDNESS AND \mathbf{NP} -COMPLETENESS)
 A language $B \subseteq \{0, 1\}^*$ is *polynomial-time Karp reducible* to a language $C \subseteq \{0, 1\}^*$ (sometimes shortened to just “polynomial-time reducible”) denoted by $B \leq_p C$ if there is a polynomial-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ such that for every $x \in \{0, 1\}^*$, $x \in B$ if and only if $f(x) \in C$.
 We say that C is **NP-hard** if $B \leq_p C$ for every $B \in \mathbf{NP}$. We say that C is **NP-complete** if C is **NP-hard** and $C \in \mathbf{NP}$.

Some texts use the names “many-to-one reducibility” or “polynomial-time mapping reducibility” instead of “polynomial-time Karp reducibility”.

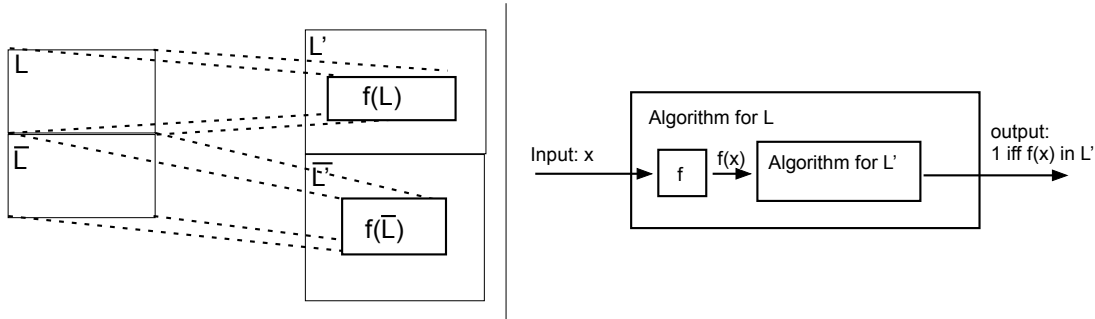


Figure 2.1: A Karp reduction from L to L' is a polynomial-time function f that maps strings in L to strings in L' and strings in $\bar{L} = \{0, 1\}^* \setminus L$ to strings in \bar{L}' . It can be used to transform a polynomial-time TM M' that decides L' into a polynomial-time TM M for L by setting $M(x) = M'(f(x))$.

Note that \leq_p is a *relation* among languages, and part 1 of Theorem 2.8 below shows that this relation is *transitive*. Later we will define other notions of reduction, and all will satisfy transitivity. Part 2 of the Theorem suggests the reason for the term **NP-hard** —namely, an **NP-hard** languages is *at least as hard* as any other **NP** language. Part 3 similarly suggests the reason for the term **NP-complete**: to study the **P** versus **NP** question it suffices to study whether any **NP-complete** problem can be decided in polynomial time.

THEOREM 2.8

1. (Transitivity) If $B \leq_p C$ and $C \leq_p D$, then $B \leq_p D$.
2. If language B is **NP-hard** and $B \in \mathbf{P}$ then $\mathbf{P} = \mathbf{NP}$.
3. If language B is **NP-complete** then $B \in \mathbf{P}$ if and only if $\mathbf{P} = \mathbf{NP}$.

PROOF: The main observation is that $(n^c)^d = n^{cd}$, and that if c, d are some constants independent of n then so is cd . Hence if p, q are two functions that have polynomial growth then their composition $p(q(n))$ also has polynomial growth. We now prove part 1 and leave the others as simple exercises.

If f_1 is a polynomial-time reduction from B to C and f_2 is a reduction from C to D then the mapping $x \mapsto f_2(f_1(x))$ is a polynomial-time reduction from B to D since $f_2(f_1(x))$ takes polynomial time to compute given x and $f_2(f_1(x)) \in D$ iff $x \in B$. ■

Do **NP-complete** languages exist? In other words, does **NP** contain a single language that is as hard as any other language in the class? There is a simple example of such a language:

THEOREM 2.9

The following language is **NP**-complete:

$$\text{TMSAT} = \{ \langle \alpha, x, 1^n, 1^t \rangle : \exists u \in \{0, 1\}^n \text{ s.t. } M_\alpha \text{ outputs 1 on input } \langle x, u \rangle \text{ within } t \text{ steps} \}$$

where M_α denotes the TM represented by the string α .²

PROOF: Once you internalize the definition of **NP**, the proof of Theorem 2.9 is straightforward. Let L be an **NP**-language. By Definition 2.1 there is a polynomial p and a verifier TM M such that $x \in L$ iff there is a string $u \in \{0, 1\}^{p(|x|)}$ satisfying $M(x, u) = 1$ and M runs in time $q(n)$ for some polynomial q . To reduce L to **TMSAT** we simply map every string $x \in \{0, 1\}^*$ to the tuple $\langle \ulcorner M \urcorner, x, 1^{p(|x|)}, 1^{q(|n|)} \rangle$, where $\ulcorner M \urcorner$ denotes the representation of M as a string. This mapping can clearly be performed in polynomial time and by the definition of **TMSAT**,

$$\langle \ulcorner M \urcorner, x, 1^{p(|x|)}, 1^{q(|n|)} \rangle \in \text{TMSAT} \Leftrightarrow \exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) \text{ outputs 1 within } q(|x|) \text{ steps.}$$

■

TMSAT is not a very useful **NP**-complete problem since its definition is intimately tied to the notion of the Turing machine. Hence the fact that **TMSAT** is **NP**-complete does not provide much new insight. In Section 2.3 we show examples of more “natural” **NP**-complete problems.

2.3 The Cook-Levin Theorem: Computation is Local

Around 1971, Cook and Levin independently discovered the notion of **NP**-completeness and gave examples of combinatorial **NP**-complete problems whose definition seems to have nothing to do with Turing machines. Soon after, Karp showed that **NP**-completeness occurs widely and many problems of practical interest are **NP**-complete. To date, thousands of computational problems in a variety of disciplines have been shown to be **NP**-complete.

2.3.1 Boolean formulae, CNF and SAT.

Some of the simplest examples of **NP**-complete problems come from propositional logic. A *Boolean formula* over the variables u_1, \dots, u_n consists of the variables and the logical operators AND (\wedge), NOT (\neg) and OR (\vee). For example, $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$ is a Boolean formula. If φ is a Boolean formula over variables u_1, \dots, u_n , and $z \in \{0, 1\}^n$, then $\varphi(z)$ denotes the value of φ when the variables of φ are assigned the values z (where we identify 1 with TRUE and 0 with FALSE). A formula φ is *satisfiable* if there exists some assignment z such that $\varphi(z)$ is TRUE. Otherwise, we say that φ is *unsatisfiable*.

The above formula $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$ is satisfiable, since the assignment $u_1 = 1, u_2 = 0, u_3 = 1$ satisfies it. In general, an assignment $u_1 = z_1, u_2 = z_2, u_3 = z_3$ satisfies the formula iff at least two of the z_i 's are 1.

A Boolean formula over variables u_1, \dots, u_n is in *CNF form* (shorthand for *Conjunctive Normal Form*) if it is an AND of OR's of variables or their negations. For example, the following is a 3CNF formula:

$$(u_1 \vee \bar{u}_2 \vee u_3) \wedge (u_2 \vee \bar{u}_3 \vee u_4) \wedge (\bar{u}_1 \vee u_3 \vee \bar{u}_4).$$

where \bar{u} denotes the negation of the variable u .

More generally, a CNF formula has the form

$$\bigwedge_i \left(\bigvee_j v_{i_j} \right),$$

²Recall that 1^k denotes the string consisting of k 1's. it is a common convention in complexity theory to provide a polynomial TM with such an input to allow it to run in time polynomial in k .

where each v_{i_j} is either a variable u_k or its negation $\neg u_k$. The terms v_{i_j} are called the *literals* of the formula and the terms $(\bigvee_j v_{i_j})$ are called its *clauses*. A k CNF is a CNF formula in which all clauses contain at most k literals.

We denote by **SAT** the language of all satisfiable CNF formulae and by **3SAT** the language of all satisfiable 3CNF formulae.³

2.3.2 The Cook-Levin Theorem

The following theorem provides us with our first natural **NP**-complete problems:

THEOREM 2.10 (COOK-LEVIN THEOREM [Coo71, Lev73])

1. **SAT** is **NP**-complete.
2. **3SAT** is **NP**-complete.

We now prove Theorem 2.10 (an alternative proof, using the notion of *Boolean circuits*, is described in Section 6.1). Both **SAT** and **3SAT** are clearly in **NP**, since a satisfying assignment can serve as the certificate that a formula is satisfiable. Thus we only need to prove that they are **NP**-hard. We do so by (a) proving that **SAT** is **NP**-hard and then (b) showing that **SAT** is polynomial-time Karp reducible to **3SAT**. This implies that **3SAT** is **NP**-hard by the transitivity of polynomial-time reductions. Part (a) is achieved by the following lemma:

LEMMA 2.11

SAT is **NP**-hard.

To prove Lemma 2.11 we have to show how to reduce *every* **NP** language L to **SAT**. In other words, we need a polynomial-time transformation that turns any $x \in \{0, 1\}^*$ into a CNF formula φ_x such that $x \in L$ iff φ_x is satisfiable. Since we know nothing about the language L except that it is in **NP**, this reduction has to rely only upon the definition of computation, and express it in some way using a Boolean formula.

2.3.3 Warmup: Expressiveness of Boolean formulae

As a warmup for the proof of Lemma 2.11 we show how to express various conditions using CNF formulae.

EXAMPLE 2.12

The formula $(x_1 \vee \overline{y_1}) \wedge (\overline{x_1} \vee x_2)$ is in CNF form. It is satisfied by only those values of x_1, y_1 that are equal. Thus, the formula

$$(x_1 \vee \overline{y_1}) \wedge (\overline{x_1} \vee y_1) \wedge \cdots \wedge (x_n \vee \overline{y_n}) \wedge (\overline{x_n} \vee y_n)$$

is satisfied by an assignment if and only if each x_i is assigned the same value as y_i .

Thus, though $=$ is not a standard Boolean operator like \vee or \wedge , we will use it as a convenient shorthand since the formula $\phi_1 = \phi_2$ is equivalent to (in other words, has the same satisfying assignments as) $(\phi_1 \vee \overline{\phi_2}) \wedge (\overline{\phi_1} \vee \phi_2)$.

In fact, CNF formulae of exponential size can express *every* Boolean function, as shown by the following simple claim:

³Strictly speaking, a string representing a Boolean formula has to be *well-formed*: strings such as $u_1 \wedge \wedge u_2$ do not represent any valid formula. As usual, we ignore this issue since it is easy to identify strings that are not well-formed, and decide that such strings represent some fixed formula.

CLAIM 2.13 (UNIVERSALITY OF AND, OR, NOT)

For every Boolean function $f : \{0, 1\}^\ell \rightarrow \{0, 1\}$ there is an ℓ -variable CNF formula φ of size $\ell 2^\ell$ such that $\varphi(u) = f(u)$ for every $u \in \{0, 1\}^\ell$, where the size of a CNF formula is defined to be the number of \wedge/\vee symbols it contains.

PROOF SKETCH: For every $v \in \{0, 1\}^\ell$, it is not hard to see that there exists a clause $C_v(z_1, z_2, \dots, z_\ell)$ in ℓ variables such that $C_v(v) = 0$ and $C_v(u) = 1$ for every $u \neq v$. For example, if $v = \langle 1, 1, 0, 1 \rangle$, the corresponding clause is $\bar{z}_1 \vee \bar{z}_2 \vee z_3 \vee \bar{z}_4$.

We let φ be the AND of all the clauses C_v for v such that $f(v) = 0$. In other words

$$\varphi = \bigwedge_{v: f(v)=0} C_v(z_1, z_2, \dots, z_\ell).$$

Note that φ has size at most $\ell 2^\ell$. For every u such that $f(u) = 0$ it holds that $C_u(u) = 0$ and hence $\varphi(u)$ is also equal to 0. On the other hand, if $f(u) = 1$ then $C_v(u) = 1$ for every v such that $f(v) = 0$ and hence $\varphi(u) = 1$. We get that for every u , $\varphi(u) = f(u)$. ■

In this chapter we will use Claim 2.13 only when the number of variables is some fixed constant.

2.3.4 Proof of Lemma 2.11

Let L be an **NP** language and let M be the polynomial time TM such that that for every $x \in \{0, 1\}^*$, $x \in L \Leftrightarrow M(x, u) = 1$ for some $u \in \{0, 1\}^{p(|x|)}$, where $p : \mathbb{N} \rightarrow \mathbb{N}$ is some polynomial. We show L is polynomial-time Karp reducible to **SAT** by describing a polynomial-time transformation $x \rightarrow \varphi_x$ from strings to CNF formulae such that $x \in L$ iff φ_x is satisfiable.

How can we construct such a formula φ_x ? The trivial idea is to use the transformation of Claim 2.13 on the Boolean function that maps $u \in \{0, 1\}^{p(|x|)}$ to $M(x, u)$. This would give a CNF formula ψ_x , such that $\psi_x(u) = M(x, u)$ for every $u \in \{0, 1\}^{p(|x|)}$. Thus a string u such that $M(x, u) = 1$ exists if and only if ψ_x is satisfiable. But this trivial idea is not useful for us, since the size of the formula ψ_x obtained from Claim 2.13 can be as large as $p(|x|)2^{p(|x|)}$. To get a smaller formula we use the fact that M runs in polynomial time, and that each basic step of a Turing machine is highly *local* (in the sense that it examines and changes only a few bits of the machine's tapes).

In the course of the proof we will make the following simplifying assumptions about the TM M : **1.** M only has two tapes: an input tape and a work/output tape and **2.** M is an *oblivious* TM in the sense that its head movement does not depend on the contents of its tapes. That is, M 's computation takes the same time for all inputs of size n and for every i the location of M 's heads at the i^{th} step depends only on i and the length of the input.

We can make these assumptions without loss of generality because for every $T(n)$ -time TM M there exists a two-tape oblivious TM \tilde{M} computing the same function in $O(T(n)^2)$ time (see Remark 1.7 and Exercise 1.8).⁴ Thus in particular, if L is in **NP** then there exists a two-tape oblivious polynomial-time TM M and a polynomial p such that

$$x \in L \Leftrightarrow \exists u \in \{0, 1\}^{p(|x|)} \text{ s.t. } M(x, u) = 1. \quad (1)$$

Denote by Q the set of M 's possible states and by Γ its alphabet. The *snapshot* of M 's execution on some input y at a particular step i is the triple $\langle a, b, q \rangle \in \Gamma \times \Gamma \times Q$ such that a, b are the symbols read by M 's heads from the two tapes and q is the state M is in at the i^{th} step (see Figure 2.2). For every $y \in \{0, 1\}^*$, the snapshot of M 's execution on input y at the i^{th} step depends on **(a)** its state

⁴In fact, with some more effort we even simulate a non-oblivious $T(n)$ -time TM by an oblivious TM running in $O(T(n) \log T(n))$ -time, see Exercise 1.9. This oblivious machine may have more than two tapes, but the proof below easily generalizes to this case.

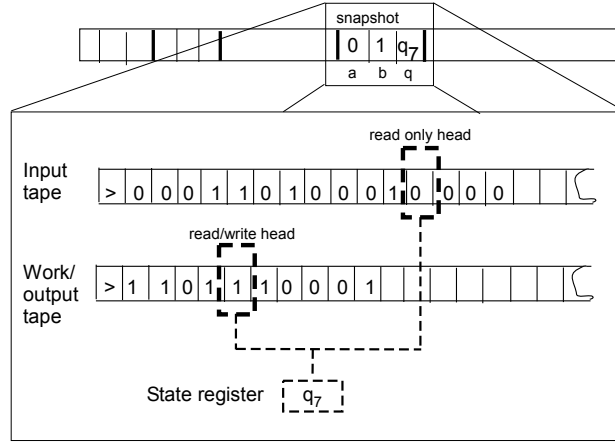


Figure 2.2: A snapshot of a TM contains the current state and symbols read by the TM at a particular step. If at the i^{th} step M reads the symbols 0, 1 from its tapes and is in the state q_7 then the snapshot of M at the i^{th} step is $\langle 0, 1, q_7 \rangle$.

in the $i - 1^{\text{st}}$ step and **(b)** the contents of the current cells of its input and work tapes. Letting z_i be the encoding of the i^{th} snapshot as a binary string of some constant length c , we write this relation as

$$z_i = F(z_{i-1}, z_{\text{prev}(i)}, y_{\text{inputpos}(i)}), \quad (2)$$

where $\text{inputpos}(i)$ denotes the location of M 's input tape head at the i^{th} step, $\text{prev}(i)$ denotes the last step before i at which M visited the same location on its work tape,⁵ and F is some function (derived from M 's transition function) that maps $\{0, 1\}^{2c+1}$ to $\{0, 1\}^c$; see Figure 2.3. Because M is oblivious, the values $\text{inputpos}(i)$ and $\text{prev}(i)$ do not depend on the particular input y . Also, these numbers can be computed in polynomial-time by simulating M on, say, the all-zeroes input.

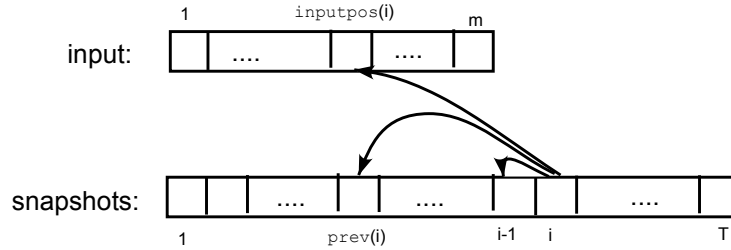


Figure 2.3: The snapshot of M at the i^{th} step depends on its previous state (contained in the snapshot at the $i - 1^{\text{st}}$ step), and the symbols read from the input tape, which is in position $\text{inputpos}(i)$, and from the work tape, which was last written to in step $\text{prev}(i)$.

To prove Lemma 2.11 we need to show a way to compute from every string $x \in \{0, 1\}^n$ in $\text{poly}(n)$ time a CNF formula φ_x such that $\varphi_x \in \text{SAT} \Leftrightarrow x \in L$, or equivalently, using (1), $\varphi_x \in \text{SAT}$ if and only if

$$\exists u \in \{0, 1\}^{p(n)} \text{ s.t. } M(x \circ u) = 1, \quad (3)$$

(where \circ denotes concatenation). Since the sequence of snapshots in M 's execution completely determines its outcome, (3) happens if and only if there exists a string $y \in \{0, 1\}^{n+p(n)}$ and a sequence of strings $z_1, \dots, z_{T(n)} \in \{0, 1\}^c$ (where $T(n)$ is the number of steps M takes on inputs of length $n + p(n)$) satisfying the following four conditions:

⁵If i is the first step that M visits a certain location, then we define $\text{prev}(i) = 1$.

1. The first n bits of y are equal to x .
2. The string z_1 encodes the initial snapshot of M . That is, z_1 encodes the triple $\langle \triangleright, \square, q_{\text{start}} \rangle$ where \triangleright is the start symbol of the input tape, \square is the blank symbol, and q_{start} is the initial state of the TM M .
3. For every $i \in \{2, \dots, T(n)\}$, $z_i = F(z_{i-1}, z_{\text{inputpos}(i)}, z_{\text{prev}(i)})$.
4. The last string $z_{T(n)}$ encodes a snapshot in which the machine halts and outputs 1.

The formula φ_x will take variables $y \in \{0, 1\}^{n+p(n)}$ and $z \in \{0, 1\}^{cT(n)}$ and will verify that y, z satisfy the AND of these four conditions. Clearly $x \in L \Leftrightarrow \varphi_x \in \text{SAT}$ and so all that remains is to show that we can express φ_x as a polynomial-sized CNF formula.

Condition 1 can be expressed as a CNF formula of size $4n$ (see Example 2.12). Conditions 2 and 4 each depend on c variables and hence by Claim 2.13 can be expressed by CNF formulae of size $c2^c$. Condition 3, which is an AND of T conditions each depending on at most $3c + 1$ variables, can be expressed as a CNF formula of size at most $T(n)(3c + 1)2^{3c+1}$. Hence the AND of all these conditions can be expressed as a CNF formula of size $d(n + T(n))$ where d is some constant depending only on M . Moreover, this CNF formula can be computed in time polynomial in the running time of M . ■

2.3.5 Reducing SAT to 3SAT.

To complete the proof of Theorem 2.10 it suffices to prove the following lemma:

LEMMA 2.14

$\text{SAT} \leq_p \text{3SAT}$.

PROOF: We give a transformation that maps each CNF formula φ into a 3CNF formula ψ such that ψ is satisfiable if and only if φ is. We demonstrate first the case that φ is a 4CNF. Let C be a clause of φ , say $C = u_1 \vee \bar{u}_2 \vee \bar{u}_3 \vee u_4$. We add a new variable z to the φ and replace C with the pair of clauses $C_1 = u_1 \vee \bar{u}_2 \vee z$ and $C_2 = \bar{u}_3 \vee u_4 \vee \bar{z}$. Clearly, if $u_1 \vee \bar{u}_2 \vee \bar{u}_3 \vee u_4$ is true then there is an assignment to z that satisfies both $u_1 \vee \bar{u}_2 \vee z$ and $\bar{u}_3 \vee u_4 \vee \bar{z}$ and vice versa: if C is false then no matter what value we assign to z either C_1 or C_2 will be false. The same idea can be applied to a general clause of size 4, and in fact can be used to change every clause C of size k (for $k > 3$) into an equivalent pair of clauses C_1 of size $k - 1$ and C_2 of size 3 that depend on the k variables of C and an additional auxiliary variable z . Applying this transformation repeatedly yields a polynomial-time transformation of a CNF formula φ into an equivalent 3CNF formula ψ . ■

2.3.6 More thoughts on the Cook-Levin theorem

The Cook-Levin theorem is a good example of the power of abstraction. Even though the theorem holds regardless of whether our computational model is the C programming language or the Turing machine, it may have been considerably more difficult to discover in the former context.

The proof of the Cook-Levin Theorem actually yields a result that is a bit stronger than the theorem's statement:

1. If we use the efficient simulation of a standard TM by an oblivious TM (see Exercise 1.9, Chapter 1) then for every $x \in \{0, 1\}^*$, the size of the formula φ_x (and the time to compute it) is $O(T \log T)$, where T is the number of steps the machine M takes on input x (see Exercise 2.11).

2. The reduction f from an **NP**-language L to **SAT** presented in Lemma 2.11 not only satisfied that $x \in L \Leftrightarrow f(x) \in \text{SAT}$ but actually the proof yields an efficient way to transform a certificate for x to a satisfying assignment for $f(x)$ and vice versa. We call a reduction with this property a *Levin reduction*. One can also verify that the proof supplied a one-to-one and onto map between the set of certificates for x and the set of satisfying assignments for $f(x)$, implying that they are of the same size. A reduction with this property is called *parsimonious*. Most of the known **NP**-complete problems (including all the ones mentioned in this chapter) have parsimonious Levin reductions from all the **NP** languages (see Exercise 2.12). As we will see in this book, this fact is sometimes useful for certain applications.

Why 3SAT? The reader may wonder why the fact that 3SAT is **NP**-complete is so much more interesting than the fact that, say, the language TMSAT of Theorem 2.9 is **NP**-complete. One reason is that 3SAT is useful for proving the **NP**-completeness of other problems: it has very minimal combinatorial structure and thus easy to use in reductions. Another reason is that propositional logic has had a central role in mathematical logic, which is why Cook and Levin were interested in 3SAT in the first place. A third reason is its practical importance: 3SAT is a simple example of *constraint satisfaction problems*, which are ubiquitous in many fields including artificial intelligence.

2.4 The web of reductions

Cook and Levin had to show how *every* **NP** language can be reduced to **SAT**. To prove the **NP**-completeness of any other language L , we do not need to work as hard: it suffices to reduce **SAT** or 3SAT to L . Once we know that L is **NP**-complete we can show that an **NP**-language L' is in fact **NP**-complete by reducing L to L' . This approach has been used to build a “web of reductions” and show that thousands of interesting languages are in fact **NP**-complete. We now show the **NP**-completeness of a few problems. More examples appear in the exercises (see Figure 2.4).

Recall the problem of planning a dinner party where every pair of guests is on speaking terms, formalized in Example 0.1 as the language $\text{INDSET} = \{\langle G, k \rangle : G \text{ has independent set of size } k\}$.

THEOREM 2.15

INDSET is **NP**-complete.

PROOF: As shown in Example 2.2, INDSET is in **NP**, and so we only need to show that it is **NP**-hard, which we do by reducing 3SAT to INDSET. Specifically, we will show how to transform in polynomial time every m -clause 3CNF formula φ into a $7m$ -vertex graph G such that φ is satisfiable if and only if G has an independent set of size at least m .

The graph G is defined as follows (see Figure 2.5): we associate a cluster of 7 vertices in G with each clause of φ . The vertices in a cluster associated with a clause C correspond to the 7 possible satisfying partial assignments to the three variables on which C depends (we call these *partial assignments*, since they only give values for some of the variables). For example, if C is $\overline{u_2} \vee \overline{u_5} \vee u_7$ then the 7 vertices in the cluster associated with C correspond to all partial assignments of the form $u_1 = a, u_2 = b, u_3 = c$ for a binary vector $\langle a, b, c \rangle \neq \langle 1, 1, 0 \rangle$. (If C depends on less than three variables then we repeat one of the partial assignments and so some of the 7 vertices will correspond to the same assignment.) We put an edge between two vertices of G if they correspond to *inconsistent* partial assignments. Two partial assignments are consistent if they give the same value to all the variables they share. For example, the assignment $u_2 = 0, u_{17} = 1, u_{26} = 1$ is inconsistent with the assignment $u_2 = 1, u_5 = 0, u_7 = 1$ because they share a variable (u_2) to which they give a different value. In addition, we put edges between every two vertices that are in the same cluster.

Clearly, transforming φ into G can be done in polynomial time, and so all that remains to show is that φ is satisfiable iff G has an independent set of size m :

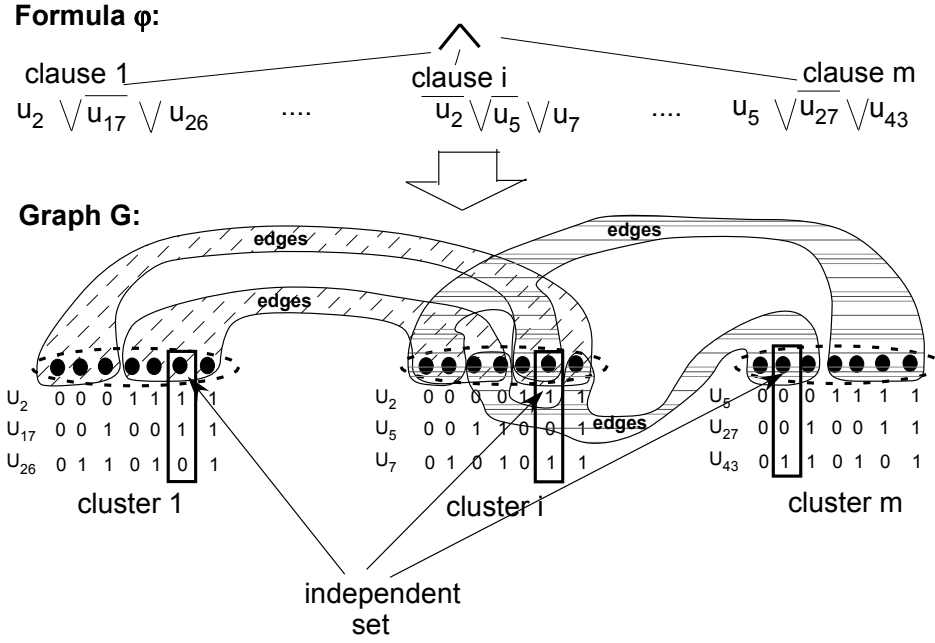


Figure 2.5: We transform a 3CNF formula φ with m clauses into a graph G with $7m$ vertices as follows: each clause C is associated with a cluster of 7 vertices corresponding to the 7 possibly satisfying assignments to the variables C depends on. We put edges between any two vertices in the same cluster and any two vertices corresponding to *inconsistent* partial assignments. The graph G will have an independent set of size m if and only if φ was satisfiable.

- Suppose that φ has a satisfying assignment u . Define a set S of m of G 's vertices as follows: for every clause C of φ put in S the vertex in the cluster associated with C that corresponds to the restriction of u to the variables C depends on. Because we only choose vertices that correspond to restrictions of the assignment u , no two vertices of S correspond to inconsistent assignments and hence S is an independent set of size m .
- Suppose that G has an independent set S of size m . We will use S to construct a satisfying assignment u for φ . We define u as follows: for every $i \in [n]$, if there is a vertex in S whose partial assignment gives a value a to u_i , then set $u_i = a$; otherwise set $u_i = 0$. This is well defined because S is an independent set, and hence each variable u_i can get at most a single value by assignments corresponding to vertices in S . On the other hand, because we put all the edges within each cluster, S can contain at most a single vertex in each cluster, and hence there is an element of S in every one of the m clusters. Thus, by our definition of u , it satisfies all of φ 's clauses.

■

We let 0/1IPROG be the set of satisfiable 0/1 *Integer programs*, as defined in Example 2.3. That is, a set of linear inequalities with rational coefficients over variables u_1, \dots, u_n is in 0/1IPROG if there is an assignment of numbers in $\{0, 1\}$ to u_1, \dots, u_n that satisfies it.

THEOREM 2.16

Then, 0/1IPROG is **NP**-complete.

PROOF: 0/1IPROG is clearly in **NP** since the assignment can serve as the certificate. To reduce SAT to 0/1IPROG note that every CNF formula can be easily expressed as an integer program by expressing every clause as an inequality. For example, the clause $u_1 \vee \bar{u}_2 \vee \bar{u}_3$ can be expressed as $u_1 + (1 - u_2) + (1 - u_3) \geq 1$. ■

A *Hamiltonian path* in a directed graph is a path that visits all vertices exactly once. Let dHAMPATH denote the set of all directed graphs that contain such a path.

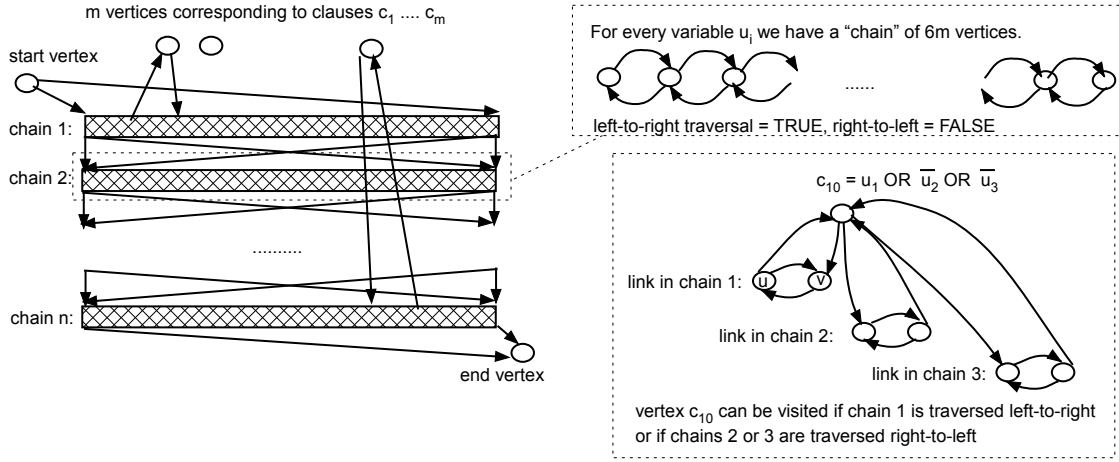


Figure 2.6: Reducing SAT to dHAMPATH. A formula φ with n variables and m clauses is mapped to a graph G that has m vertices corresponding to the clauses and n doubly linked chains, each of length $6m$, corresponding to the variables. Traversing a chain left to right corresponds to setting the variable to TRUE, while traversing it right to left corresponds to setting it to FALSE. Note that in the figure every Hamiltonian path that takes the edge from u to c_{10} must immediately take the edge from c_{10} to v , as otherwise it would get “stuck” the next time it visits v .

THEOREM 2.17

dHAMPATH is NP-complete.

PROOF: dHAMPATH is in NP since the ordered list of vertices in the path can serve as a certificate. To show that dHAMPATH is NP-hard we show a way to map every CNF formula φ into a graph G such that φ is satisfiable if and only if G has a Hamiltonian path (i.e. a path that visits all of G 's vertices exactly once).

The reduction is described in Figure 2.6. The graph G has (1) m vertices for each of φ 's clauses c_1, \dots, c_m , (2) a special starting vertex v_{start} and ending vertex v_{end} and (3) n “chains” of $4m$ vertices corresponding to the n variables of φ . A chain is a set of vertices v_1, \dots, v_{4m} such that for every $i \in [4m - 1]$, v_i and v_{i+1} are connected by two edges in both directions.

We put edges from the starting vertex v_{start} to the two extreme points of the first chain. We also put edges from the extreme points of the j^{th} chain to the extreme points to the $j + 1^{th}$ chain for every $j \in [n - 1]$. We put an edge from the extreme points of the n^{th} chain to the ending vertex v_{end} .

In addition to these edges, for every clause C of φ , we put edges between the chains corresponding to the variables appearing in C and the vertex v_C corresponding to C in the following way: if C contains the literal u_j then we take two neighboring vertices v_i, v_{i+1} in the j^{th} chain and put an edge from v_i to C and from C to v_{i+1} . If C contains the literal \bar{u}_j then we connect these edges in the opposite direction (i.e., v_{i+1} to C and C to v_i). When adding these edges, we never “reuse” a link v_i, v_{i+1} in a particular chain and always keep an unused link between every two used links. We can do this since every chain has $4m$ vertices, which is more than sufficient for this. We now prove that $\varphi \in \text{SAT} \Leftrightarrow G \in \text{dHAMPATH}$:

($\varphi \in \text{SAT} \Rightarrow G \in \text{dHAMPATH}$): Suppose that φ has a satisfying assignment u_1, \dots, u_n . We will show a path that visits all the vertices of G . The path will start at v_{start} , travel through all the chains in order, and end at v_{end} . For starters, consider the path that travels the j^{th} chain in left-to-right order if $u_j = 1$ and travels it in right-to-left order if $u_j = 0$. This path visits all the vertices except for those corresponding to clauses. Yet, if u is a satisfying assignment then the path can be easily modified to visit all the vertices corresponding to clauses: for each clause C there is at least one literal that is true, and we can use one link on the chain corresponding to that literal to “skip” to the vertex v_C and continue on as before.

($G \in \text{dHAMPATH} \Rightarrow \varphi \in \text{SAT}$): Suppose that G has an Hamiltonian path P . We first note that

the path P must start in v_{start} (as it has no incoming edges) and end at v_{end} (as it has no outgoing edges). Furthermore, we claim that P needs to traverse all the chains in order, and within each chain traverse it either in left-to-right order or right-to-left order. This would be immediate if the path did not use the edges from a chain to the vertices corresponding to clauses. The claim holds because if a Hamiltonian path takes the edge $u \rightarrow w$, where u is on a chain and w corresponds to a clause, then it must at the next step take the edge $w \rightarrow v$ where v is the vertex adjacent to u in the link. Otherwise, the path will get stuck the next time it visits v (see Figure 2.1). Now, define an assignment u_1, \dots, u_n to φ as follows: $u_j = 1$ if P traverses the j^{th} chain in left-to-right order, and $u_j = 0$ otherwise. It is not hard to see that because P visits all the vertices corresponding to clauses, u_1, \dots, u_n is a satisfying assignment for φ . ■

In praise of reductions

Though originally invented as part of the theory of **NP**-completeness, the polynomial-time reduction (together with its first cousin, the randomized polynomial-time reduction defined in Section 7.6) has led to a rich understanding of complexity above and beyond **NP**-completeness. Much of complexity theory and cryptography today (thus, many chapters of this book) consists of using reductions to make connections between disparate complexity theoretic conjectures. Why do complexity theorists excel at reductions but not at actually proving lower bounds on Turing machines? A possible explanation is that humans have evolved to excel at problem solving, and hence are more adept at algorithms (after all, a reduction is merely an algorithm to transform one problem into another) than at proving lower bounds on Turing machines.

2.5 Decision versus search

We have chosen to define the notion of **NP** using Yes/No problems (“Is the given formula satisfiable?”) as opposed to *search* problems (“Find a satisfying assignment to this formula if one exists”). Clearly, the search problem is harder than the corresponding decision problem, and so if $\mathbf{P} \neq \mathbf{NP}$ then neither one can be solved for an **NP**-complete problem. However, it turns out that for **NP**-complete problems they are equivalent in the sense that if the decision problem can be solved (and hence $\mathbf{P} = \mathbf{NP}$) then the search version of any **NP** problem can also be solved in polynomial time.

THEOREM 2.18

*Suppose that $\mathbf{P} = \mathbf{NP}$. Then, for every **NP** language L there exists a polynomial-time TM B that on input $x \in L$ outputs a certificate for x .*

That is, if, as per Definition 2.1, $x \in L$ iff $\exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x,u) = 1$ where p is some polynomial and M is a polynomial-time TM, then on input $x \in L$, $B(x)$ will be a string $u \in \{0,1\}^{p(|x|)}$ satisfying $M(x,B(x)) = 1$.

PROOF: We start by showing the theorem for the case of **SAT**. In particular we show that given an algorithm A that decides **SAT**, we can come up with an algorithm B that on input a satisfiable CNF formula φ with n variables, finds a satisfying assignment for φ using $2n + 1$ calls to A and some additional polynomial-time computation.

The algorithm B works as follows: we first use A to check that the input formula φ is satisfiable. If so, we substitute $x_1 = 0$ and $x_1 = 1$ in φ (this transformation, that simplifies and shortens the formula a little, leaving a formula with $n - 1$ variables, can certainly be done in polynomial time) and then use A to decide which of the two is satisfiable (it is possible that both are). Say the first is satisfiable. Then we fix $x_1 = 0$ from now on and continue with the simplified formula. Continuing this way we end up fixing all n variables while ensuring that each intermediate formula is satisfiable. Thus the final assignment to the variables satisfies φ .

To solve the search problem for an arbitrary **NP**-language L , we use the fact that the reduction of Theorem 2.10 from L to **SAT** is actually a *Levin* reduction. This means that we have a polynomial-time computable function f such that not only $x \in L \Leftrightarrow f(x) \in \mathbf{SAT}$ but actually we can map a satisfying assignment of $f(x)$ into a certificate for x . Therefore, we can use the algorithm above to come up with an assignment for $f(x)$ and then map it back into a certificate for x . ■

The proof of Theorem 2.18 shows that **SAT** is *downward self-reducible*, which means that given an algorithm that solves **SAT** on inputs of length smaller than n we can solve **SAT** on inputs of length n . Using the Cook-Levin reduction, one can show that all **NP**-complete problems have a similar property.

2.6 coNP, EXP and NEXP

Now we define some additional complexity classes related to **P** and **NP**.

2.6.1 coNP

If $L \subseteq \{0,1\}^*$ is a language, then we denote by \bar{L} the *complement* of L . That is, $\bar{L} = \{0,1\}^* \setminus L$. We make the following definition:

DEFINITION 2.19

coNP = $\{L : \bar{L} \in \mathbf{P}\}$.

coNP is *not* the complement of the class **NP**. In fact, **coNP** and **NP** have a non-empty intersection, since every language in **P** is in $\mathbf{NP} \cap \mathbf{coNP}$ (see Exercise 2.22). The following is an example of a **coNP** language: $\overline{\mathbf{SAT}} = \{\varphi : \varphi \text{ is not satisfiable}\}$. Students sometimes mistakenly convince themselves that $\overline{\mathbf{SAT}}$ is in **NP**. They have the following polynomial time NDTM in mind: on input φ , the machine guesses an assignment. If this assignment does not satisfy φ then it accepts (i.e., goes into q_{accept} and halts) and if it does satisfy φ then the machine halts without accepting. This NDTM does not do the job: indeed it accepts every unsatisfiable φ but in addition it also accepts many satisfiable formulae (i.e., every formula that has a single unsatisfying assignment). That is why pedagogically we prefer the following definition of **coNP** (which is easily shown to be equivalent to the first, see Exercise 2.23):

DEFINITION 2.20 (**coNP**, ALTERNATIVE DEFINITION)

For every $L \subseteq \{0,1\}^*$, we say that $L \in \mathbf{coNP}$ if there exists a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$,

$$x \in L \Leftrightarrow \forall u \in \{0,1\}^{p(|x|)} \text{ such that } M(x, u) = 1$$

Note the use of the “ \forall ” quantifier in this definition where Definition 2.1 used \exists .

We can define **coNP**-completeness in analogy to **NP**-completeness: a language is **coNP**-complete if it is in **coNP** and every **coNP** language is polynomial-time Karp reducible to it.

EXAMPLE 2.21

The following language is **coNP**-complete:

$$\mathbf{TAUTOLOGY} = \{\varphi : \varphi \text{ is a Boolean formula that is satisfied by every assignment}\}.$$

It is clearly in **coNP** by Definition 2.20 and so all we have to show is that for every $L \in \mathbf{coNP}$, $L \leq_p \mathbf{TAUTOLOGY}$. But this is easy: just modify the Cook-Levin reduction from \bar{L} (which is in **NP**) to **SAT**. For every input $x \in \{0,1\}^*$ that reduction produces a formula φ_x that is satisfiable iff $x \in \bar{L}$. Now consider the formula $\neg\varphi_x$. It is in **TAUTOLOGY** iff $x \in L$, and this completes the description of the reduction.

It's not hard to see that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{NP} = \mathbf{coNP} = \mathbf{P}$ (Exercise 2.24). Put in the contrapositive, if we can show that $\mathbf{NP} \neq \mathbf{coNP}$ then we have shown $\mathbf{P} \neq \mathbf{NP}$. Most researchers believe that $\mathbf{NP} \neq \mathbf{coNP}$. The intuition is almost as strong as for the \mathbf{P} versus \mathbf{NP} question: it seems hard to believe that there is any short certificate for certifying that a given formula is a TAUTOLOGY, in other words, to certify that *every* assignment satisfies the formula.

2.6.2 EXP and NEXP

The following two classes are exponential time analogues of \mathbf{P} and \mathbf{NP} .

DEFINITION 2.22

$\mathbf{EXP} = \cup_{c \geq 0} \mathbf{DTIME}(2^{n^c})$.

$\mathbf{NEXP} = \cup_{c \geq 0} \mathbf{NTIME}(2^{n^c})$.

Because every problem in \mathbf{NP} can be solved in exponential time by a brute force search for the certificate, $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP}$. Is there any point to studying classes involving exponential running times? The following simple result —providing merely a glimpse of the rich web of relations we will be establishing between disparate complexity questions— may be a partial answer.

THEOREM 2.23

If $\mathbf{EXP} \neq \mathbf{NEXP}$ then $\mathbf{P} \neq \mathbf{NP}$.

PROOF: We prove the contrapositive: assuming $\mathbf{P} = \mathbf{NP}$ we show $\mathbf{EXP} = \mathbf{NEXP}$. Suppose $L \in \mathbf{NTIME}(2^{n^c})$ and NDTM M decides it. We claim that then the language

$$L_{\text{pad}} = \left\{ \langle x, 1^{2^{|x|^c}} \rangle : x \in L \right\} \quad (4)$$

is in \mathbf{NP} . Here is an NDTM for L_{pad} : given y , first check if there is a string z such that $y = \langle z, 1^{2^{|z|^c}} \rangle$. If not, output REJECT. If y is of this form, then run M on z for $2^{|z|^c}$ steps and output its answer. Clearly, the running time is polynomial in $|y|$, and hence $L_{\text{pad}} \in \mathbf{NP}$. Hence if $\mathbf{P} = \mathbf{NP}$ then L_{pad} is in \mathbf{P} . But if L_{pad} is in \mathbf{P} then L is in \mathbf{EXP} : to determine whether an input x is in L , we just pad the input and decide whether it is in L_{pad} using the polynomial-time machine for L_{pad} . ■

The *padding* technique used in this proof, whereby we transform a language by “padding” every string in a language with a string of (useless) symbols, is also used in several other results in complexity theory (see, e.g., Section 13.4.1). In many settings it can be used to show that equalities between complexity classes “scale up”; that is, if two different type of resources solve the same problems within bound $T(n)$ then this also holds for functions T' larger than T . Viewed contrapositively, padding can be used to show that inequalities between complexity classes involving resource bound $T'(n)$ “scale down” to resource bound $T(n)$.

Like \mathbf{P} and \mathbf{NP} , most of the complexity classes we study in this book are also contained in both \mathbf{EXP} and \mathbf{NEXP} .

2.7 More thoughts about \mathbf{P} , \mathbf{NP} , and all that

2.7.1 The philosophical importance of \mathbf{NP}

At a totally abstract level, the \mathbf{P} versus \mathbf{NP} question may be viewed as a question about the power of nondeterminism in the Turing machine model. Similar questions have been completely answered for simpler models such as finite automata.

However, the certificate definition of **NP** also suggests that the **P** versus **NP** question captures a widespread phenomenon of some philosophical importance (and a source of great frustration): recognizing the correctness of an answer is often much easier than coming up with the answer. Appreciating a Beethoven sonata is far easier than composing the sonata; verifying the solidity of a design for a suspension bridge is easier (to a civil engineer anyway!) than coming up with a good design; verifying the proof of a theorem is easier than coming up with a proof itself (a fact referred to in Gödel's letter quoted at the start of the chapter), and so forth. In such cases, coming up with the right answer seems to involve *exhaustive search* over an exponentially large set. The **P** versus **NP** question asks whether exhaustive search can be avoided in general. It seems obvious to most people—and the basis of many false proofs proposed by amateurs—that exhaustive search cannot be avoided: checking that a given salesperson tour (provided by somebody else) has length at most k ought to be a lot easier than coming up with such a tour by oneself. Unfortunately, turning this intuition into a proof has proved difficult.

2.7.2 NP and mathematical proofs

By definition, **NP** is the set of languages where membership has a short certificate. This is reminiscent of another familiar notion, that of a mathematical proof. As noticed in the past century, in principle all of mathematics can be axiomatized, so that proofs are merely formal manipulations of axioms. Thus the correctness of a proof is rather easy to verify—just check that each line follows from the previous lines by applying the axioms. In fact, for most known axiomatic systems (e.g., Peano arithmetic or Zermelo-Fraenkel Set Theory) this verification runs in time *polynomial* in the length of the proof. Thus the following problem is in **NP** for any of the usual axiomatic systems \mathcal{A} :

$$\text{THEOREMS} = \{(\varphi, 1^n) : \varphi \text{ has a formal proof of length } \leq n \text{ in system } \mathcal{A}\}.$$

Exercise 2.10 asks you to prove that this problem is **NP**-complete. Hence the **P** versus **NP** question is a *rephrasing* of Gödel's question, which asks whether or not there is a algorithm that finds mathematical proofs in time polynomial in the length of the proof.

Of course, you all know in your guts that finding correct proofs is far harder than verifying their correctness. So presumably, you believe at an intuitive level that $\mathbf{P} \neq \mathbf{NP}$.

2.7.3 What if $\mathbf{P} = \mathbf{NP}$?

If $\mathbf{P} = \mathbf{NP}$ —specifically, if an **NP**-complete problem like 3SAT had a very efficient algorithm running in say $O(n^2)$ time—then the world would be mostly a Utopia. Mathematicians could be replaced by efficient theorem-discovering programs (a fact pointed out in Kurt Gödel's 1956 letter and discovered three decades later). In general for every search problem whose answer can be efficiently verified (or has a short certificate of correctness), we will be able to find the correct answer or the short certificate in polynomial time. AI software would be perfect since we could easily do exhaustive searches in a large tree of possibilities. Inventors and engineers would be greatly aided by software packages that can design the perfect part or gizmo for the job at hand. VLSI designers will be able to whip up optimum circuits, with minimum power requirements. Whenever a scientist has some experimental data, she would be able to automatically obtain the simplest theory (under any reasonable measure of simplicity we choose) that best explains these measurements; by the principle of Occam's Razor the simplest explanation is likely to be the right one. Of course, in some cases it took scientists centuries to come up with the simplest theories explaining the known data. This approach can be used to solve also non-scientific problems: one could find the simplest theory that explains, say, the list of books from the New York *Times*' bestseller list. All these applications will be a consequence of our study of the Polynomial Hierarchy in Chapter 5.

Somewhat intriguingly, this Utopia would have no need for randomness. As we will later see, if $\mathbf{P} = \mathbf{NP}$ then randomized algorithms would buy essentially no efficiency gains over deterministic

algorithms; see Chapter 7. (Philosophers should ponder this one.)

This Utopia would also come at one price: there would be no privacy in the digital domain. Any encryption scheme would have a trivial decoding algorithm. There would be no digital cash, no SSL, RSA or PGP (see Chapter 9). We would just have to learn to get along better without these, folks.

This utopian world may seem ridiculous, but the fact that we can't rule it out shows how little we know about computation. Taking the half-full cup point of view, it shows how many wonderful things are still waiting to be discovered.

2.7.4 What if $\mathbf{NP} = \mathbf{coNP}$?

If $\mathbf{NP} = \mathbf{coNP}$, the consequences still seem dramatic. Mostly, they have to do with existence of short certificates for statements that do not seem to have any. To give an example, consider the **NP**-complete problem of finding whether or not a set of multivariate polynomials has a common root (see Exercise 2.19). In other words, it is **NP** complete to decide whether a system of equations of the following type has a solution:

$$\begin{aligned} f_1(x_1, \dots, x_n) &= 0 \\ f_2(x_1, \dots, x_n) &= 0 \\ &\vdots \\ f_m(x_1, \dots, x_n) &= 0 \end{aligned}$$

where each f_i is a polynomial of degree at most 2.

If a solution exists, then that solution serves as a *certificate* to this effect (of course, we have to also show that the solution can be described using a polynomial number of bits, which we omit). The problem of deciding that the system does *not* have a solution is of course in **coNP**. Can we give a certificate to the effect that the system does *not* have a solution? Hilbert's Nullstellensatz Theorem seems to do that: it says that the system is infeasible iff there is a sequence of polynomials g_1, g_2, \dots, g_m such that $\sum_i f_i g_i = 1$, where 1 on the right hand side denotes the constant polynomial 1.

What is happening? Does the Nullstellensatz prove $\mathbf{coNP} = \mathbf{NP}$? No, because the degrees of the g_i 's—and hence the number of bits used to represent them—could be exponential in n, m . (And it is simple to construct f_i 's for which this is necessary.)

However, if $\mathbf{NP} = \mathbf{coNP}$ then there would be some *other* notion of a short certificate to the effect that the system is infeasible. The effect of such a result on mathematics could be even greater than the effect of Hilbert's Nullstellensatz.

2.7.5 Coping with **NP** hardness.

NP-complete problems turn up in great many applications, from flight scheduling to genome sequencing. What do you do if the problem you need to solve turns out to be **NP**-complete? At the outset, the situation looks bleak: if $\mathbf{P} \neq \mathbf{NP}$ then there simply does not *exist* an efficient algorithm to solve such a problem. However, there may still be some hope: **NP** completeness only means that (assuming $\mathbf{P} \neq \mathbf{NP}$) the problem does not have an algorithm that solves it *exactly* on *every* input. But for many applications, an *approximate* solution on *some* of the inputs might be good enough.

A case in point is the traveling salesperson problem (**TSP**), of computing, given a list of pairwise distances between n cities, the shortest route that travels through all of them. Assume that you are indeed in charge of coming up with travel plans for traveling salespeople that need to visit various cities around your country. Does the fact that **TSP** is **NP**-complete means that you are bound to do a hopelessly suboptimal job? This does not have to be the case.

First note that you do not need an algorithm that solves the problem on *all* possible lists of pairwise distances. We might model the inputs that actually arise in this case as follows: the n cities are points on a plane, and the distance between a pair of cities is the distance between the corresponding points (we are neglecting here the difference between travel distance and direct/arial distance). It is an easy exercise to verify that not all possible lists of pairwise distances can be generated in such a way. We call those that do *Euclidean* distances. Another observation is that computing the *exactly* optimal travel plan may not be so crucial. If you could always come up with a travel plan that is at most 1% longer than the optimal, this should be good enough.

It turns out that neither of these observations on its own is sufficient to make the problem tractable. The TSP problem is still **NP** complete even for Euclidean distances. Also if $\mathbf{P} \neq \mathbf{NP}$ then TSP is hard to approximate within any constant factor. However, *combining* the two observations together actually helps: for every ϵ there is a $\text{poly}(n(\log n)^{O(1/\epsilon)})$ -time algorithm that given Euclidean distances between n cities comes up with a tour that is at most a factor of $(1 + \epsilon)$ worse than the optimal tour [Aro98].

Discovering that the problem you encounter is **NP**-complete should not be cause for immediate despair. Rather you should view this as indication that a more careful modeling of the problem is needed, letting the literature on complexity and algorithms guide you as to what features might make the problem more tractable. Alternatives to worst-case exact computation are explored in Chapters 17 and 21, that investigate *average-case complexity* and *approximation algorithms* respectively.

WHAT HAVE WE LEARNED?

- The class **NP** consists of all the languages for which membership can be certified to a polynomial-time algorithm. It contains many important problems not known to be in **P**. **NP** can also be defined using non-deterministic Turing machines.
- **NP**-complete problems are the hardest problems in **NP**, in the sense that they have a polynomial-time algorithm if and only if $\mathbf{P} = \mathbf{NP}$. Many natural problems that seemingly have nothing to do with Turing machines turn out to be **NP**-complete. One such example is the language 3SAT of satisfiable Boolean formulae in 3CNF form.
- If $\mathbf{P} = \mathbf{NP}$ then for every search problem for which one can efficiently verify a given solution, one can also efficiently find such a solution from scratch.
- The class **coNP** is the set of complements of **NP**-languages. We believe that $\mathbf{coNP} \neq \mathbf{NP}$. This is a stronger assumption than $\mathbf{P} \neq \mathbf{NP}$.

Chapter notes and history

In the 1950's, Soviet scientists were aware of the undesirability of using exhaustive or brute force search, which they called *perebor*, for combinatorial problems, and asked the question of whether certain problems *inherently* require such search (see [Tra84]). In the west the first published description of this issue is by Edmonds [Edm65], in the paper quoted in the previous chapter. However, on both sides of the iron curtain it took some time to realize the right way to formulate the problem and to arrive at the modern definition of the classes **NP** and **P**. Amazingly, in his 1956 letter to von Neumann we quoted above, Gödel essentially asks the question of **P** vs. **NP**, although there is no hint that he realized that one of the particular problems he mentions is **NP**-complete.

Unfortunately, von Neumann was very sick at the time, and as far as we know, no further research was done by either on them on this problem, and the letter was only discovered in the 1980's.

In 1971 Cook published his seminal paper defining the notion of **NP**-completeness and showing that **SAT** is **NP** complete [Coo71]. Soon afterwards, Karp [Kar72] showed that 21 important problems are in fact **NP**-complete, generating tremendous interest in this notion. Meanwhile in the USSR Levin independently defined **NP**-completeness (although he focused on search problems) and showed that a variant of **SAT** is **NP**-complete. (Levin's paper [Lev73] was published in 1973, but he had been giving talks on his results since 1971, also in those days there was essentially zero communication between eastern and western scientists.) See Sipser's survey [Sip92] for more on the history of **P** and **NP** and a full translation of Gödel's remarkable letter.

The book by Garey and Johnson [GJ79] and the web site [CK00] contain many more examples of **NP** complete problems. Some such problems have been studied well before the invention of computers: the traveling salesperson problem has been studied in the 19th century (see [LLKS85]). Also, a recently discovered letter by Gauss to Schumacher shows that Gauss was thinking about methods to solve the famous *Euclidean Steiner Tree* problem —today known to be **NP**-hard— in the early 19th century. See also Wigderson's survey [Wig06] for more on the relations between **NP** and mathematics.

Aaronson [Aar05] surveys various attempts to solve **NP** complete problems via “non-traditional” computing devices.

Even if $\mathbf{NP} \neq \mathbf{P}$, this does not necessarily mean that all of the utopian applications mentioned in Section 2.7.3 are gone. It may be that, say, **3SAT** is hard to solve in the worst case on every input but actually very easy on the average, See Chapter 17 for a more detailed study of *average-case* complexity. Also, Impagliazzo [Imp95a] has an excellent survey on this topic.

An intriguing possibility is that it is simply impossible to resolve the **P** vs. **NP** question using the accepted axioms of mathematics: this has turned out to be the case with some other question, the most famous of which is Cantor's “Continuum Hypothesis”, see Aaronson's survey [Aar03].

Alon and Kilian (personal communication) showed that the condition that P is prime is crucial to ensure that the language **Factoring** of Example 2.3 does capture the factoring problem, since without this condition this language is **NP**-complete (for reasons having nothing to do with the hardness of factoring integers).

Exercises

- 2.1. Prove that changing the condition ($[\mathbf{np:eq:npdef}]$) of Definition 2.1 to

$$x \in L \Leftrightarrow \exists u \text{ s.t. } |u| \leq p(|x|) \text{ and } M(x, u) = 1 \quad (5)$$

does not change the class **NP**. That is, if we let **NP'** to be the complexity class obtained by replacing Condition ($[\mathbf{np:eq:npdef}]$) with Condition (5) in Definition 2.1 then $\mathbf{NP}' = \mathbf{NP}$.

- 2.2. Prove that the following languages are in **NP**:

Two coloring: $2\text{COL} = \{G : \text{graph } G \text{ has a coloring with 2 colors}\}$, where a coloring of G with c colors is an assignment of a number in $[c]$ to each vertex such that no adjacent vertices get the same number.

Three coloring: $3\text{COL} = \{G : \text{graph } G \text{ has a coloring with 3 colors}\}$.

Connectivity: $\text{CONN} = \{G : G \text{ is a connected graph}\}$.

- 2.3. Let **LINEQ** denote the set of satisfiable rational linear equations. That is, **LINEQ** consists of the set of all pairs $\langle A, \mathbf{b} \rangle$ where A is an $m \times n$ rational matrix and \mathbf{b} is an m dimensional rational vector, such that $A\mathbf{x} = \mathbf{b}$ for some n -dimensional vector \mathbf{x} . Prove that **LINEQ** is in

NP (the key is to prove that if there exists such a vector \mathbf{x} , then there exists an \mathbf{x} whose coefficients can be represented using a number of bits that is polynomial in the representation of A, \mathbf{b}).

(Note that **LINEQ** is actually in **P**: can you show this?.)

Hint: First show that for every rational matrix A , the determinant of A can always be represented using a number of bits that is polynomial in the representation of A . Then use Cramer's rule for expressing the solution of linear equations in terms of determinants.

- 2.4. [Pratt75] Let $\text{PRIMES} = \{ \lfloor n \rfloor : n \text{ is prime} \}$. Show that $\text{PRIMES} \in \mathbf{NP}$. You can use the following fact: a number n is prime iff for every prime factor q of $n-1$, there exists a number $a \in \{2, \dots, n-1\}$ satisfying $a^{n-1} = 1 \pmod{n}$ but $a^{(n-1)/q} \neq 1 \pmod{n}$.

Hint: The certificate that n is prime is the list of prime factors q_1, \dots, q_ℓ of $n-1$ along with the corresponding numbers a_1, \dots, a_ℓ and (recursive) primality certificates for q_1, \dots, q_ℓ .

- 2.5. Prove the existence of a *non-deterministic Universal TM* (analogously to the deterministic universal TM of Theorem 1.9). That is, prove that there exists a representation scheme of NDTMs, and an NDTM \mathcal{NU} such that for every string α , and input x , $\mathcal{NU}(x, \alpha) = M_\alpha(x)$.

- (a) Prove that there exists such a universal NDTM \mathcal{NU} such that if M_α halts on x within T steps, then \mathcal{NU} halts on x, α within $CT \log T$ steps (where C is a constant depending only on the machine represented by α).
- (b) Prove that there is such a universal NDTM that runs on these inputs for at most Ct steps.

Hint: A simulation in $O(|\alpha|t \log t)$ time can be obtained by a straightforward adaptation of the proof of Theorem 1.9. To do a more efficient simulation, the main idea is to first run a simulation of M without actually reading the contents of the work tapes, but rather simply non-deterministically guessing these contents, and writing those guesses down. Then, go over tape by tape and verify that all guesses were consistent.

- 2.6. Prove Parts 2 and 3 of Theorem 2.8.

- 2.7. Let **HALT** be the Halting language defined in Theorem 1.11. Show that **HALT** is **NP**-hard. Is it **NP**-complete?

- 2.8. We have defined a relation \leq_p among languages. We noted that it is *reflexive* (that is, $L \leq_p L$ for all languages L) and *transitive* (that is, if $L \leq_p L'$ and $L' \leq_p L''$ then $L \leq_p L''$). Show that it is not *symmetric*, namely, $L \leq_p L'$ need not imply $L' \leq_p L$.

- 2.9. Suppose $L_1, L_2 \in \mathbf{NP}$. Then is $L_1 \cup L_2$ in **NP**? What about $L_1 \cap L_2$?

- 2.10. Mathematics can be axiomatized using for example the *Zermelo Frankel* system, which has a finite description. Argue at a high level that the following language is **NP**-complete.

$\{ \langle \varphi, 1^n \rangle : \text{math statement } \varphi \text{ has a proof of size at most } n \text{ in the ZF system} \}.$

Hint: Why is this language in **NP**? Is Boolean satisfiability a mathematical statement?

The question of whether this language is in **P** is essentially the question asked by Gödel in the chapter's initial quote.

- 2.11. Show that for every time constructible $T : \mathbb{N} \rightarrow \mathbb{N}$, if $L \in \mathbf{NTIME}(T(n))$ then we can give a polynomial-time Karp reduction from L to 3SAT that transforms instances of size n into 3CNF formulae of size $O(T(n) \log T(n))$. Can you make this reduction also run in $O(T(n) \text{poly}(\log T(n)))$?
- 2.12. Recall that a reduction f from an **NP**-language L to an **NP**-languages L' is *parsimonious* if the number of certificates of f is equal to the number of certificates of $f(x)$.
- (a) Prove that the reduction from every **NP**-language L to SAT presented in the proof of Lemma 2.11 is parsimonious.
 - (b) Show a parsimonious reduction from SAT to 3SAT.
- 2.13. Cook [Coo71] used a somewhat different notion of reduction: a language L is *polynomial-time Cook reducible* to a language L' if there is a polynomial time TM M that, given an *oracle* for deciding L' , can decide L . An oracle for L' is a magical extra tape given to M , such that whenever M writes a string on this tape and goes into a special “invocation” state, then the string—in a single step!—gets overwritten by 1 or 0 depending upon whether the string is or is not in L' ; see Section 3.4 for a more precise definition.

Show that the notion of cook reducibility is transitive and that 3SAT is Cook-reducible to TAUTOLOGY.

- 2.14. In the CLIQUE problem we are given an undirected graph G and an integer K and have to decide whether there is a subset S of at least K vertices such that every two distinct vertices $u, v \in S$ have an edge between them (such a subset is called a *clique* of G). In the VERTEX COVER problem we are given an undirected graph G and an integer K and have to decide whether there is a subset S of at most K vertices such that for every edge \overline{ij} of G , at least one of i or j is in S (such a subset is called a *vertex cover* of G). Prove that both these problems are **NP**-complete.

Hint: reduce from INDSET.

- 2.15. In the MAX CUT problem we are given an undirected graph G and an integer K and have to decide whether there is a subset of vertices S such that there are at least K edges that have one endpoint in S and one endpoint in \bar{S} . Prove that this problem is **NP**-complete.
- 2.16. In the Exactly One 3SAT problem, we are given a 3CNF formula φ and need to decide if there exists a satisfying assignment u for φ such that every clause of φ has exactly one TRUE literal. In the SUBSET SUM problem we are given a list of n numbers A_1, \dots, A_n and a number T and need to decide whether there exists a subset $S \subseteq [n]$ such that $\sum_{i \in S} A_i = T$ (the problem size is the sum of all the bit representations of all numbers). Prove that both Exactly One 3SAT and SUBSET SUM are **NP**-complete.

Hint: For Exactly One 3SAT replace each occurrence of a literal v_i in a clause C by a new variable $z_{i,C}$ and clauses and auxiliary variables ensuring that if v_i is TRUE then $z_{i,C}$ is allowed to be either TRUE or FALSE but if v_i is false then $z_{i,C}$ must be FALSE. The approach for the reduction of Exactly One 3SAT to SUBSET SUM is that given a formula φ , we map it to a SUBSET SUM instance by mapping each possible literal u_i to the number $\sum_{j \in S_i} (2n)^j$ where S_i is the set of clauses that the literal u_i satisfies, and setting the target T to be $\sum_{j=1}^m (2n)^j$. An additional trick is required to ensure that the solution to the subset sum instance will not include two literals that correspond to a variable and its negation.

- 2.17. Prove that the language **HAMPATH** of *undirected* graphs with Hamiltonian paths is **NP**-complete. Prove that the language **TSP** described in Example 2.3 is **NP**-complete. Prove that the language **HAMCYCLE** of undirected graphs that contain Hamiltonian cycle (a simple cycle involving all the vertices) is **NP**-complete.
- 2.18. Let **QUADEQ** be the language of all satisfiable sets of *quadratic equations* over 0/1 variables (a quadratic equations over u_1, \dots, u_n has the form $\sum_{i,j \in [n]} a_{i,j} u_i u_j = b$) where addition is modulo 2. Show that **QUADEQ** is **NP**-complete.

Hint: Reduce from SAT

- 2.19. Let **REALQUADEQ** be the language of all satisfiable sets of quadratic equations over *real* variables. Show that **REALQUADEQ** is **NP**-complete.

Hint: You can express the constraint $x \in \{0, 1\}$ using the equation $x^2 = x$.

- 2.20. Let $3\text{COL} = \{G : \text{graph } G \text{ has a coloring with 3 colors}\}$. Prove that **3COL** is **NP**-complete.
- 2.21. In a typical auction of n items, the auctioneer will sell the i^{th} item to the person that gave it the highest bid. However, sometimes the items sold are related to one another (e.g., think of lots of land that may be adjacent to one another) and so people may be willing to pay a high price to get, say, the three items $\{2, 5, 17\}$, but only if they get all of them together. In this case, deciding what to sell to whom might not be an easy task. The **COMBINATORIAL AUCTION** problem is to decide, given numbers n, k , and a list of pairs $\{\langle S_i, x_i \rangle\}_{i=1}^m$ where S_i is a subset of $[n]$ and x_i is an integer, whether there exist disjoint sets $S_{i_1}, \dots, S_{i_\ell}$ such that $\sum_{j=1}^{\ell} x_{i_j} \geq k$. That is, if x_i is the amount a bidder is willing to pay for the set S_i , then the problem is to decide if the auctioneer can sell items and get a revenue of at least k , under the obvious condition that he can't sell the same item twice. Prove that **COMBINATORIAL AUCTION** is **NP**-complete.
- 2.22. Prove that $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.
- 2.23. Prove that Definitions 2.19 and 2.20 do indeed define the same class **coNP**.
- 2.24. Prove that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{NP} = \mathbf{coNP}$.
- 2.25. Show that $\mathbf{NP} = \mathbf{coNP}$ iff **3SAT** and **TAUTOLOGY** are polynomial-time reducible to one another.
- 2.26. Give a definition of **NEXP** without using NDTMs, analogous to Definition 2.1 of the class **NP**, and prove that the two definitions are equivalent.
- 2.27. We say that a language is **NEXP**-complete if it is in **NEXP** and every language in **NEXP** is polynomial-time reducible to it. Describe a **NEXP**-complete language L . Prove that if $L \in \mathbf{EXP}$ then $\mathbf{NEXP} = \mathbf{EXP}$.
- 2.28. Suppose $L_1, L_2 \in \mathbf{NP} \cap \mathbf{coNP}$. Then show that $L_1 \oplus L_2$ is in $\mathbf{NP} \cap \mathbf{coNP}$, where $L_1 \oplus L_2 = \{x : x \text{ is in exactly one of } L_1, L_2\}$.
- 2.29. (Berman's Theorem 1978) A language is called *unary* if every string in it is of the form 1^i (the string of i ones) for some $i > 0$. Show that if there exists an **NP**-complete unary language then $\mathbf{P} = \mathbf{NP}$. (See Exercise 6.9 of Chapter 6 for a strengthening of this result.)

Hint: If there is a n^c time reduction from **3SAT** to a unary language L , then this reduction can only map size n instances of **3SAT** to some string of the form 1^i where $i \leq n^c$. Use this observation to obtain a polynomial-time algorithm for **SAT** using the downward self reducibility argument of Theorem 2.18.

- 2.30. Define the language **UNARY SUBSET SUM** to be the variant of the **SUBSET SUM** problem of Exercise 2.16 where all numbers are represented by the *unary* representation (i.e., the number k is represented as 1^k). Show that **UNARY SUBSET SUM** is in **P**.

Hint: Start with an exponential-time recursive algorithm for **SUBSET SUM**, and show that in this case you can make it into a polynomial-time algorithm by storing previously computed values in a table.

- 2.31. Prove that if every *unary* **NP**-language is in **P** then **EXP** = **NEXP**. (A language L is unary iff it is a subset of $\{1\}^*$, see Exercise 2.29.)
- 2.32. Let $\Sigma_2\text{SAT}$ denote the following decision problem: given a quantified formula ψ of the form

$$\psi = \exists_{x \in \{0,1\}^n} \forall_{y \in \{0,1\}^m} \text{ s.t. } \varphi(x, y) = 1,$$

where φ is a CNF formula, decide whether ψ is true. That is, decide whether there exists an x such that for every y , $\varphi(x, y)$ is true. Prove that if **P** = **NP** then $\Sigma_2\text{SAT}$ is in **P**.

- 2.33. Suppose that you are given a graph G and a number K , and are told that either (i) the smallest vertex cover (see Exercise 2.14) of G is of size at most K or (ii) it is of size at least $3K$. Show a polynomial-time algorithm that can distinguish between these two cases. Can you do it with a smaller constant than 3? Since **VERTEX COVER** is **NP**-hard, why does this algorithm not show that **P** = **NP**?

