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 letter to John von Neumann, 1956; see [**?**]

[a] In 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 there is often a big difference between solving a problem from scratch and verifying a given solution. In the previous chapter we already encountered **P**, the class of decision problems that can be efficiently solved. In this chapter, we define the complexity class **NP** that aims to capture the set of problems whose solution can be easily *verified*. The famous **P** versus **NP** question asks whether or not the two are the same. The resolution of this conjecture will be of great practical, scientific and philosophical interest; see Section 2.6.

The chapter also introduces **NP**-*completeness*, an important class of combinatorial problems that are in **P** if and only if **P** = **NP**. Notions such as *reductions* and *completeness* encountered in this study motivate many other definitions encountered later in the book.

2.1 The class NP

We will give several equivalent definitions of **NP**. The first one is as follows: **NP** contains every language for which membership has a polynomial-time verifiable *certificate* (some texts call it a *witness*). An example is the independent set (INDSET) problem mentioned in Chapter (see Example 2.3 below).

DEFINITION 2.1 (THE CLASS **NP**)
For every $L \subseteq \{0,1\}^*$, we say that $L \in \mathbf{NP}$ if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M 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 $u \in \{0,1\}^{p(|x|)}$ satisfies $M(x,u) = 1$ then we say that u is a *certificate* for x (with respect to the language L and machine M).

We have the following trivial relationship.

THEOREM 2.2
$\mathbf{P} \subseteq \mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$.

PROOF: ($\mathbf{P} \subseteq \mathbf{NP}$): 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).

($\mathbf{NP} \subseteq \bigcup_{c>1} \mathbf{DTIME}(2^{n^c})$): 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 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 $c > 1$ then this machine runs in $2^{O(n^c)}$ time. Thus the theorem is proven. ∎

The question whether or not $\mathbf{P} = \mathbf{NP}$ is considered *the* central open question of complexity theory, and is also an important question in mathematics and science at large (see Section 2.6). Most researchers believe that $\mathbf{P} \neq \mathbf{NP}$ since years of effort has failed to yield efficient algorithms for certain **NP** languages.

EXAMPLE 2.3

It is usually convenient to think of languages as decision problems. Here are some examples of decision problems in **NP**:

Independent set: Given a graph G and a number k, decide if there is a k-size independent subset of G's vertices. The certificate for membership is the list of k vertices forming an independent set.

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 $A_1, \ldots, A_n$ 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 $u_1, \ldots, u_n$ (a linear inequality has the form $a_1u_1 + a_2u_2 + \ldots + a_nu_n \leq b$ for some coefficients $a_1, \ldots, a_n, b$), decide if there is an assignment of rational numbers to the variables $u_1, \ldots, u_n$ that satisfies all the inequalities. The certificate is the assignment.

Integer programming: Given a list of m linear inequalities with rational coefficients over n variables $u_1, \ldots, u_m$, find out if there is an assignment of *integer* numbers to $u_1, \ldots, u_n$ satisfying 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 $\pi : [n] \to [n]$ 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 factor M in the interval $[L, U]$. The certificate is the factor M.

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.

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 [**?**]). The composite numbers problem was only recently shown to be in **P** by Agrawal, Kayal and Saxena [**?**], who gave a beautiful algorithm to solve it. For the linear programming problem this is again highly non-trivial, and follows from the Ellipsoid algorithm of Khachiyan [**?**] (there are also faster algorithms, following Karmarkar's interior point paradigm [**?**]).

All the rest of the problems are not known to have a polynomial-time algorithm, although we have no proof that they are not in **P**. The *Independent Set*, *Traveling Salesperson*, *Subset Sum*, and *Integer Programming* problems are known to be **NP**-*complete*, which, as we will see in this chapter, 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.

2.2 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 is that an NDTM has *two* transition functions δ_0 and δ_1. In addition the NDTM has a special state we denote by q_{accept}. When an NDTM M computes, 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 a halting state or q_{accept} within $T(|x|)$ steps.

Definition 2.4
For every function $T : \mathbb{N} \to \mathbb{N}$ 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 definiton of **NP**, the one that appears in most texts.

Theorem 2.5
$\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NTIME}(n^c)$

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 $p : \mathbb{N} \to \mathbb{N}$ 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. Notice, 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, there exists a *universal* non-deterministic Turing machine, see Exercise 1. (In fact, using non-determinism we can even make the simulation by a universal TM slightly more efficient.)

2.3 Reducibility and NP-completeness

It turns out that the independent set is at least as hard as any other language in **NP**: if INDSET 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 **NP** $\neq$ **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 B is at least as hard as some language A? The crucial tool we use is the notion of a *reduction* (see Figure 2.1):

> Definition 2.6 (Reductions, **NP**-hardness and **NP**-completeness)
> We say that a language $A \subseteq \{0,1\}^*$ is *polynomial-time Karp reducible to a language* $B \subseteq \{0,1\}^*$ (sometimes shortened to just "polynomial-time reducible"[1]) denoted by $A \leq_p B$ if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in A \Leftrightarrow x \in B$.
> We say that B is **NP**-*hard* if $A \leq_p B$ for every $A \in$ **NP**. We say that B is **NP**-*complete* if B is **NP**-hard and $B \in$ **NP**.

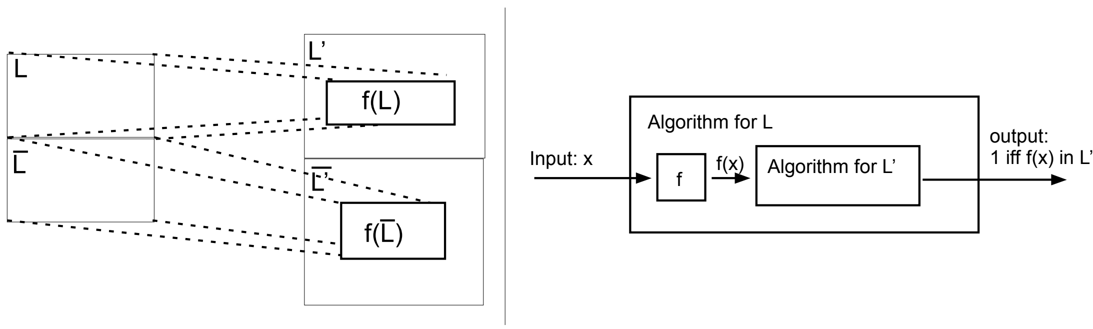

Figure 2.1: A Karp reduction from L to L' is a polynomial-time function f that maps strings in L to strings in L' and strings in $\overline{L} = \{0,1\}^* \setminus L$ to strings in $\overline{L'}$. It can be used to transform a polynomial-time TM M' that decides L into a polynomial-time TM M for L by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

Now we observe some properties of polynomial-time reductions. Part 1 shows that this relation is *transitive*. (Later we will define other notions of reduction, and all will satisfy transitivity.) Part 2 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.

[1]Some texts call this notion "many-to-one reducibility" or "polynomial-time mapping reducibility".

Theorem 2.7

1. (transitivity) *$A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.*

2. *If language A is* **NP***-hard and $A \in \mathbf{P}$ then $\mathbf{P} = \mathbf{NP}$.*

3. *If language A is* **NP***-complete then $A \in \mathbf{P}$ if and only if $\mathbf{P} = \mathbf{NP}$.*

Proof: The main observation is that if p, q are two functions that have polynomial growth then their composition $p(q(n))$ also has polynomial growth. We prove part 1 and leave the others as simple exercises.

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

Do **NP**-complete languages exist? It may not be clear that **NP** should possess a language that is as hard as any other language in the class. However, this does turn out to be the case:

Theorem 2.8
The following language is **NP***-complete:*

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

where $\llcorner M \lrcorner$ denotes the representation of the TM M according to the representation scheme of Theorem 1.6.

Theorem 2.8 is straightforward from the definition of **NP** and is left to the reader as Exercise 2. But TMSAT is not a very useful **NP**-complete problem since its definition is intimately tied to the notion of the Turing machine, and hence the fact that it is **NP**-complete does not provide much new insight.

2.3.1 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 combinatorial problems of practical interest are **NP**-complete. To date, thousands of computational problems in a variety of disciplines have been found to be **NP**-complete.

Some of the simplest examples of **NP**-complete problems come from propositional logic. A *Boolean formula* over the variables $u_1, \ldots, u_n$ consists of the variables and the logical operators AND ($\wedge$), NOT ($\neg$) and OR ($\vee$); see Appendix for their definitions. For example, $(a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$ is a Boolean formula that is TRUE if and only if the majority of the variables a, b, c are TRUE. If φ is a Boolean formula over variables $u_1, \ldots, u_n$, 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). The formula is *satisfiable* if there is an assignment to the variables that makes it evaluate to TRUE. Otherwise, we say φ is *unsatisfiable*.

DEFINITION 2.9 (CNF, kCNF)
A Boolean formula over variables $u_1, \ldots, u_n$ is in *CNF form* (shorthand for *Conjunctive Normal Form*) if it is an AND of OR's of variables or their negations. It has the form

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

where each v_{i_j} is a *literal* of φ, in other words either a variable u_k or to its negation $\overline{u}_k$. The terms $(\vee_j v_{i_j})$ are called the *clauses*. If all clauses contain at most k literals, the formula is a kCNF.

For example, the following is a CNF formula that is 3CNF.

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

DEFINITION 2.10 (SAT AND 3SAT)
SAT is the language of all satisfiable CNF formulae and 3SAT is the language of all satisfiable 3CNF formulae.

THEOREM 2.11 (COOK-LEVIN THEOREM [?, ?])

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

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

REMARK 2.12
An alternative proof of the Cook-Levin theorem, using the notion of *Boolean circuits*, is described in Section 6.7.

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 first proving that SAT is **NP**-hard and then showing that SAT is polynomial-time Karp reducible to 3SAT. This implies that 3SAT is **NP**-hard by the transitivity of polynomial-time reductions. Thus the following lemma is the key to the proof.

LEMMA 2.13
SAT is **NP**-*hard.*

Notice, to prove this we have to show how to reduce *every* **NP** language L to SAT, in other words give 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 just upon the definition of computation, and express it in some way using a boolean formula.

Expressiveness of boolean formulae

As a warmup for the proof of Lemma 2.13 we show how to express constraints using boolean formulae.

EXAMPLE 2.14
The formula $(a \vee \overline{b}) \wedge (\overline{a} \vee b)$ is in CNF form. It is satisfied by only those values of a, b 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 TRUE if and only if the strings $x, y \in \{0,1\}^n$ are equal to one another.

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 sufficient size can express *every* Boolean condition, as shown by the following simple claim: (this fact is sometimes known as *universality* of the operations AND, OR and NOT)

CLAIM 2.15
For every Boolean function $f : \{0,1\}^\ell \to \{0,1\}$ *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: For every $v \in \{0,1\}^\ell$, there exists a clause C_v 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 $\overline{u}_1 \vee \overline{u}_2 \vee u_3 \vee \overline{u}_4$. We let S be the set $\{v : f(v) = 1\}$ and set $\varphi = \wedge_{v\in S} C_v$. Then for every v such that $f(v) = 0$ it holds that $\varphi(v) = 0$ and for every u such that $f(u) = 1$, we have that $C_v(u) = 1$ for every $v \in S$ and hence $\varphi(u) = 1$. ■

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

Proof of Lemma 2.13

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 $p : \mathbb{N} \to \mathbb{N}$ is some polynomial. We show L is polynomial-time Karp reducible to SAT by describing a way to transform in polynomial-time every string $x \in \{0,1\}^*$ into a CNF formula φ_x such that $x \in L$ iff φ_x is satisfiable.

How can we construct such a formula φ_x? By Claim 2.15, the function that maps $u \in \{0,1\}^{p(|x|)}$ to $M(x,u)$ can be expressed as a CNF formula ψ_x (i.e., $\psi_x(u) = M(x,u)$ for every $u \in \{0,1\}^{p(|x|)}$). Thus a u such that $M(x,u) = 1$ exists if and only if ψ_x is satisfiable. But this is not useful for us, since the size of the formula ψ_x obtained from Claim 2.15 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).

We assume for notational simplicity that M only has two tapes: an input tape and a work/output tape. The proof carries over in the same way for any fixed number of tapes. We also assume that M is an *oblivious* TM in the sense that its head movement does not depend on the contents of its input tape. In particular, this means that M's computation takes the same time for all inputs of size n and for each time step i the location of M's heads at the i^{th} step depends only on i and M's input length. We can make this assumption since every TM can be easily simulated by an oblivious TM incurring only a polynomial overhead (e.g., by replacing each step of the computation with a left to right and back sweep of the machine's tapes; see Exercise 5 of Chapter 1).[2]

[2]In fact, with some more effort we even simulate a non-oblivious $T(n)$-time TM by an

The advantage of the obliviousness assumption is that there are polynomial-time computable functions $\texttt{inputpos}(i), \texttt{prev}(i)$ where $\texttt{inputpos}(i)$ denotes the location of the input tape head at step i and $\texttt{prev}(i)$ denotes the last step before i that M visited the same location on its work tape, see Figure 2.3. These values can be computed in polynomial-time by simulating the machine on say the input $(0^{|x|}, 0^{p(|x|)})$.

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). Each such snapshot can be encoded by a binary string of length c, where c is some constant depending on M's state and alphabet size (but independent of the input length).

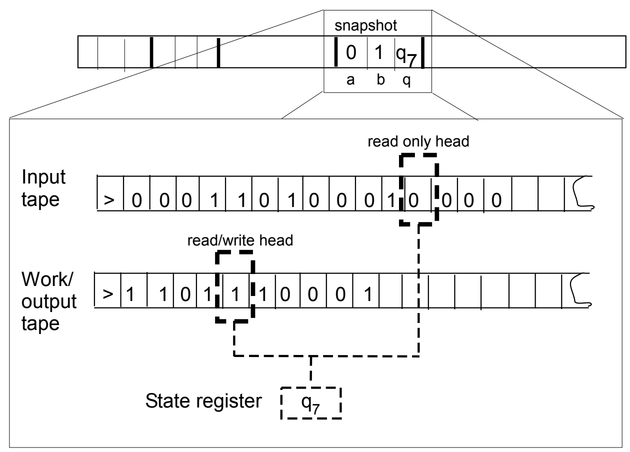

Figure 2.2: A snapshot of a TM contains the current state and symbols read by the TM at a particular step. If at the i^{th} step M reads the symbols $0, 1$ from its tapes and is in the state q_7 then the snapshot of M at the i^{th} step is $\langle 0, 1, q_7\rangle$.

For every $m \in \mathbb{N}$ and $y \in \{0,1\}^m$, the snapshot of M's execution on input y at the i^{th} step depends on its state in the $i-1^{st}$ step, and the contents of the current cells of its input and work tapes. Thus if we denote the encoding of the i^{th} snapshot as a length-c string by z_i, then z_i is a function of z_{i-1}, $y_{\texttt{inputpos}(i)}$, and $z_{\texttt{prev}(i)}$, where $\texttt{inputpos}(i), \texttt{prev}(i)$ are as defined earlier.

We write this as

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

oblivious TM running in $O(T(n)\log T(n))$-time, see Exercise 6 of Chapter 1.

where F is some function (derived from M's transition function) that maps $\{0,1\}^{2c+1}$ to $\{0,1\}^c$.[3]

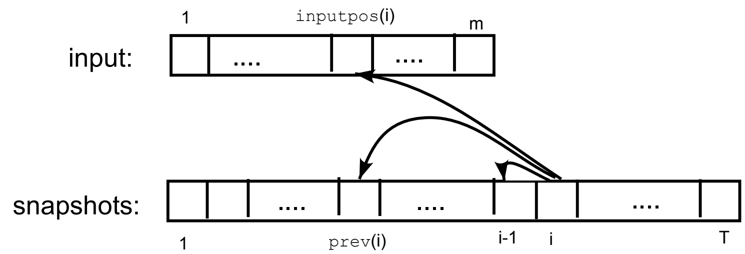

Figure 2.3: The snapshot of M at the i^{th} step depends on its previous state (contained in the snapshot at the $i-1^{st}$ step), and the symbols read from the input tape, which is in position $\texttt{inputpos}(i)$, and from the work tape, which was last written to in step $\texttt{prev}(i)$.

Let $n \in \mathbb{N}$ and $x \in \{0,1\}^n$. We need to construct a CNF formula φ_x such that $x \in L \Leftrightarrow \varphi_x \in \mathsf{SAT}$. Recall that $x \in L$ if and only if there exists some $u \in \{0,1\}^{p(n)}$ such that $M(y) = 1$ where $y = x \circ u$ (with $\circ$ denoting concatenation). Since the sequence of snapshots in M's execution completely determines its outcome, this happens if and only if there exists a string $y \in \{0,1\}^{n+p(n)}$ and a sequence of strings $z_1, \ldots, z_T \in \{0,1\}^c$ (where $T = T(n)$ is the number of steps M takes on inputs of length $n + p(n)$) satisfying the following four conditions:

1. The first n bits of y are equal to x.
2. The string z_1 encodes the initial snapshot of M (i.e., the triple $\langle \triangleright, \Box, q_{\mathsf{start}} \rangle$ where $\triangleright$ is the start symbol of the input tape, $\Box$ is the blank symbol, and q_{start} is the initial state of the TM M).
3. For every $i \in \{2, .., T\}$, $z_i = F(z_{i-1}, z_{\texttt{inputpos}(i)}, z_{\texttt{prev}(i)})$.
4. The last string z_T 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 $z \in \{0,1\}^{cT}$ and will verify that y, z satisfy the AND of these four conditions. Clearly $x \in L \Leftrightarrow \varphi_x \in \mathsf{SAT}$ and so all that remains is to show that we can express φ_x as a polynomial-sized CNF formula.

[3]If i is the first step in which M accesses a particular location of the input tape then we define $\texttt{prev}(i) = 1$ and we will always require that the snapshot z_1 contains the blank symbol $\Box$ for the work tape.

Condition 1 can be expressed as a CNF formula of size $2n$ (in fact, such a formula appears in Example 2.14). Conditions 2 and 4 each depend on c variables and hence by Claim 2.15 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 $T(3c+1)2^{3c+1}$. Hence the AND of all these conditions can be expressed as a CNF formula of size $d(n+T)$ 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.

Lemma 2.16
SAT $\leq_p$ 3SAT.

Proof: We will map a 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 $C = u_1 \vee \overline{u}_2 \vee \overline{u}_3 \vee u_4$. We add a new variable z to the φ and replace C with the pair of clauses $C_1 = u_1 \vee \overline{u}_2 \vee z$ and $C_2 = \overline{u}_3 \vee u_4 \vee \overline{z}$. Clearly, if $u_1 \vee \overline{u}_2 \vee \overline{u}_3 \vee u_4$ is true then there is an assignment to z that satisfies both $u_1 \vee \overline{u}_2 \vee z$ and $\overline{u}_3 \vee u_4 \vee \overline{z}$ 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. ■

2.3.2 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.

Also, it is worth pointing out that the proof 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 6, 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 10).

2. The reduction f from an **NP**-language L to SAT presented in Lemma 2.13 not only satisfied that $x \in L \Leftrightarrow f(x) \in$ 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 11). As we will see in this book, this fact is sometimes useful for certain applications.

Why 3SAT? The reader may wonder why is the fact that 3SAT is **NP**-complete so much more interesting than the fact that, say, the language TMSAT of Theorem 2.8 is **NP**-complete. One answer 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 answer has to do with history: propositional logic has had a central role in mathematical logic —in fact it was exclusively the language of classical logic (e.g. in ancient Greece). This historical resonance is one reason why Cook and Levin were interested in 3SAT in the first place. A third answer has to do with practical importance: it is a simple example of *constraint satisfaction problems*, which are ubiquitous in many fields including artificial intelligence.

2.3.3 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). See the classic book by Garey and Johnson [**?**] and the internet site [**?**] for more.

Theorem 2.17 (Independent set is **NP**-complete)
Let INDSET $= \{\langle G, k\rangle : G$ *has independent set of size* $k\}$. *Then* INDSET *is* **NP**-*complete.*

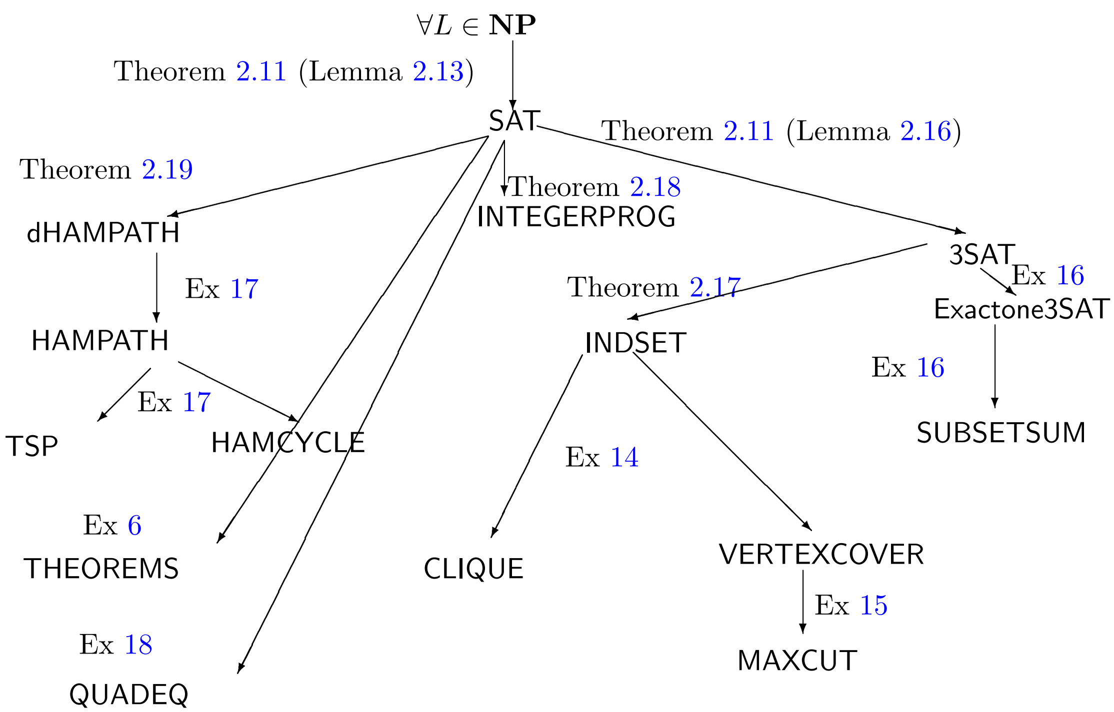

Figure 2.4: Web of reductions between the **NP**-completeness problems described in this chapter and the exercises. Thousands more are known.

PROOF: Since INDSET is clearly in **NP**, we only need to show that it is **NP**-hard, which we do by reducing 3SAT to INDSET. Let φ be a 3CNF formula on n variables with m clauses. We define a graph G of $7m$ vertices as follows: we associate a cluster of 7 vertices in G with each clause of φ. The vertices in cluster associated with a clause C correspond to the 7 possible assignments to the three variables C depends on (we call these *partial assignments*, since they only give values for some of the variables). For example, if C is $\overline{u_2} \vee \overline{u_5} \vee \overline{u_7}$ 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 $\langle a, b, c\rangle \neq \langle 1, 1, 1\rangle$. (If C depends on less than three variables we treat one of them as repeated and then 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 $u_1 = 1, u_2 = 0, u_3 = 0$ is inconsistent with the assignment $u_3 = 1, u_5 = 0, u_7 = 1$ because they share a variable (u_3) 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. We claim that φ is satisfiable if and only if G has a clique of size m. Indeed, suppose that φ has a satisfying assignment u. Define a set S of m 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.

On the other hand, 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 see that, surprisingly, the answer to the famous **NP** vs. **P** question depends on the seemingly mundane question of whether one can efficiently plan an optimal dinner party. Here are some more **NP**-completeness results:

Theorem 2.18 (Integer programming is **NP**-complete)
We say that a set of linear inequalities with rational coefficients over variables $u_1, \ldots, u_n$ is in IPROG *if there is an assignment of integer numbers in $\{0, 1, 2, \ldots\}$ to $u_1, \ldots, u_n$ that satisfies it. Then,* IPROG *is **NP**-complete.*

Proof: IPROG is clearly in **NP**. To reduce SAT to IPROG note that every CNF formula can be easily expressed as an integer program: first add the constraints $0 \leq u_i \leq 1$ for every i to ensure that the only feasible assignments to the variables are 0 or 1, then express every clause as an inequality. For example, the clause $u_1 \vee \overline{u}_2 \vee \overline{u}_3$ can be expressed as $u_1 + (1 - u_2) + (1 - u_3) \geq 1$.
■

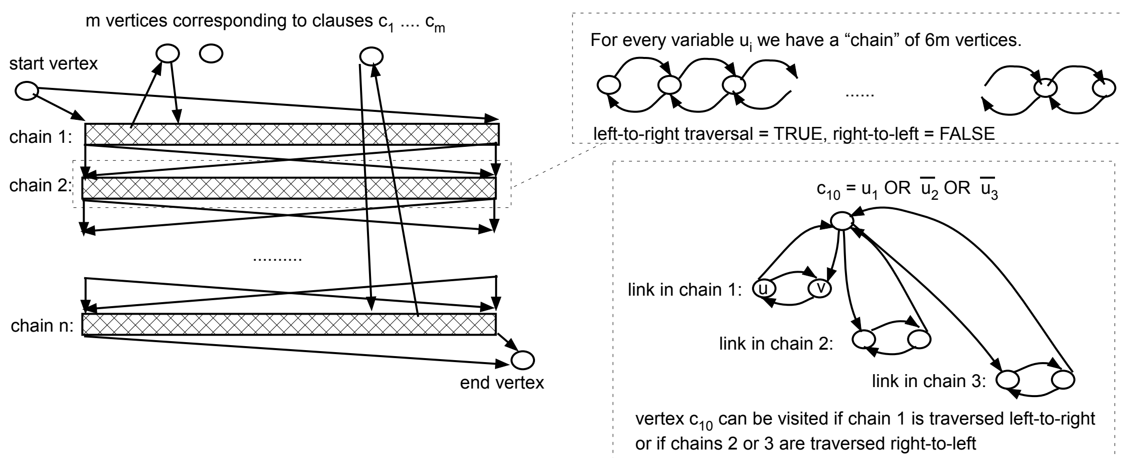

Figure 2.5: Reducing SAT to dHAMPATH. A formula φ with n variables and m clauses is mapped to a graph G that has m vertices corresponding to the clauses and n doubly linked chains, each of length $6m$, corresponding to the variables. Traversing a chain left to right corresponds to setting the variable 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 2.19 (Hamiltonian path is **NP**-complete)
Let dHAMPATH *denote the set of all directed graphs that contain a path visiting all of their vertices exactly once. Then* dHAMPATH *is **NP**-complete.*

Proof: Again, dHAMPATH is clearly in **NP**. To show it's **NP**-complete 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. path that visits all of G's vertices exactly once).

The reduction is described in Figure 2.5. The graph G has **(1)** m vertices for each of φ's clauses $c_1, \ldots, c_m$, **(2)** a special starting vertex v_{start} and ending vertex v_{end} and **(3)** n "chains" of $6m$ vertices corresponding to the n variables of φ. A chain is a set of vertices $v_1, \ldots, v_{6m}$ such that for every $i \in [6m-1]$, 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 $\overline{u}_j$ 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 $5m$ vertices, which is more than sufficient for this.

$\varphi \in \mathsf{SAT} \Rightarrow G \in \mathsf{dHAMPATH}$. Suppose that φ has a satisfying assignment $u_1, \ldots, u_n$. 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.

$G \in \mathsf{dHAMPATH} \Rightarrow \varphi \in \mathsf{SAT}$. 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 $u \to w$, where u is on a chain and w corresponds to a clause, then it must at the next step take the edge $w \to v$ 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 $u_1, \ldots, u_n$ 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, $u_1, \ldots, u_n$ 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 7.8) 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 lowerbounds 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 lowerbounds on Turing machines.

2.4 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"). However, it turns out that if $\mathbf{P} = \mathbf{NP}$ then the search versions of **NP** problems can also be solved in polynomial time.

THEOREM 2.20
*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 2.11 from L to SAT is actually a *Levin* reduction. This means that we have a polynomial-time computable function f such that not only $x \in L \Leftrightarrow f(x) \in \mathsf{SAT}$ 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. ■

Remark 2.21
The proof above 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 it on inputs of length n. Using the Cook-Levin reduction, one can show that all **NP**-complete problems have a similar property, though we do not make this formal.

2.5 coNP, EXP and NEXP

Now we define some related complexity classes that are very relevant to the study of the **P** versus **NP** question.

2.5.1 coNP

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

Definition 2.22
$\mathbf{coNP} = \left\{L : \overline{L} \in \mathbf{P}\right\}$.

It is important to note that **coNP** is *not* the complement of the class **NP**. In fact, they have a non-empty intersection, since every language in **P** is in $\mathbf{NP} \cap \mathbf{coNP}$ (see Exercise 19). The following is an example of a **coNP** language: $\overline{\mathsf{SAT}} = \{\varphi : \varphi \text{ is not satisfiable}\}$. Students sometimes mistakenly convince themselves that $\overline{\mathsf{SAT}}$ 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 20):

Definition 2.23 (**coNP**, alternative definition)
For every $L \subseteq \{0,1\}^*$, we say that $L \in \mathbf{coNP}$ if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$,

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

The key fact to note is the use of "$\forall$" 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 2.24
In classical logic, *tautologies* are true statements. The following language is **coNP**-complete:

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

It is clearly in **coNP** by Definition 2.23 and so all we have to show is that for every $L \in \mathbf{coNP}$, $L \leq_p \mathsf{TAUTOLOGY}$. But this is easy: just modify the Cook-Levin reduction from $\overline{L}$ (which is in **NP**) to SAT. For every input $x \in \{0,1\}^*$ that reduction produces a formula φ_x that is satisfiable iff $x \in \overline{L}$. Now consider the formula $\neg\varphi_x$. It is in $\mathsf{TAUTOLOGY}$ iff $x \in L$, and this completes the description of the reduction.

It is a simple exercise to check that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{NP} = \mathbf{coNP} = \mathbf{P}$. Put contrapositively, 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 **P** versus **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.

2.5.2 EXP and NEXP

The following two classes are exponential time analogues of **P** and **NP**.

Definition 2.25
$\mathbf{EXP} = \cup_{c \geq 0} \mathbf{DTIME}(2^{n^c})$.
$\mathbf{NEXP} = \cup_{c \geq 0} \mathbf{NTIME}(2^{n^c})$.

Because every problem in **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 2.26
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

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

is in **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 **P**. But if L_{pad} is in **P** then L is in **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}. ■

Remark 2.27
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. 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 resurce bound $T'(n)$ "scale down" to resource bound $T(n)$.

We note that most of the complexity classes studied later are, like **P** and **NP**, also contained in **EXP** or **NEXP**.

2.6 More thoughts about P, NP, and all that

2.6.1 The philosophical importance of NP

At a totally abstract level, the **P** versus **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 to students): recognizing the correctness of an answer is often easier than coming up with the answer. To give other analogies from life: 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 mentioned 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 salesman 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.

2.6.2 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 $\mathcal{A}$:

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

In fact, the exercises ask you to prove that this problem is **NP**-complete. Hence the **P** versus **NP** question is a *rephrasing* of Gödel's question (see quote at the beginning of the chapter), which asks whether or not there is a algorithm that finds mathematical proofs in time polynomial in the length of the proof.

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

2.6.3 What if P = 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, finding simple theories sometimes takes scientists centuries to solve. 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. (NB: 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 **P** = **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 10). 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.

2.6.4 What if NP = coNP?

If **NP** = **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, remember the **NP**-complete problem of finding whether or not a set of multivariate polynomials has a common root, in other words, deciding whether a system of equations of the following type has a solution:

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

where each f_i is a quadratic polynomial.

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 $g_1, g_2, \ldots, g_m$ 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 **coNP** = **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 **NP** = **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 would probably be even greater than the effect of Hilbert's Nullstellensatz. Of course, one can replace Nullstellensatz with any other **coNP** problem in the above discussion.

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 **P** =**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 **P** = **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.

Chapter notes and history

Sipser's survey [**?**] succinctly describes the evolution of the concepts of **P**, **NP** in various scholarly articles from the 1950s and 1960s. It also contains a translation of Gödel's remarkable letter, which was discovered among von Neumann's papers in the 1980s.

The "TSP book" by Lawler et al. [**?**] also has a similar chapter, and it traces interest in the Traveling Salesman Problem back to the 19th century. Furthermore, 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.

Even if **NP** $\neq$ **P**, this does not necessarily mean that all of the utopian applications mentioned in Section 2.6.3 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 15 for a more detailed study of *average-case* complexity.

Exercises

§1 Prove the existence of a *non-deterministic Universal TM* (analogously to the deterministic universal TM of Theorem 1.6). 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, if the NDTM M_α represented by α halts on x within at most t steps then $\mathcal{NU}(\alpha, t, x) = M_\alpha(x)$.

(a) Prove that there exists such a universal NDTM $\mathcal{NU}$ that on inputs α, t, x runs for at most $ct \log t$ 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 1.6. 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. Use can also use the fact that decreasing a counter from t to 0 can be done using $O(T)$ steps.

§2 Prove Theorem 2.

§3 Define the language H to consist of all the pairs $\langle \llcorner M \lrcorner, x\rangle$ such that $M(x)$ does not go into an infinite loop. Show that H is **NP**-hard. Is H **NP**-complete?

§4 We have defined a relation $\leq_p$ among languages. We noted that it is *reflexive* (that is, $A \leq_p A$ for all languages A) and *transitive* (that is, if $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$). Show that it is not commutative, namely, $A \leq_p B$ need not imply $B \leq_p A$.

§5 Suppose $L_1, L_2 \in$ **NP**. Then is $L_1 \cup L_2$ in **NP**? What about $L_1 \cap L_2$?

§6 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 mathmatical statement?

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

§7 Show that **NP** = **coNP** iff 3SAT and TAUTOLOGY are polynomial-time reducible to one another.

§8 Can you give a definition of **NEXP** without using NDTMs, analogous to the definition of **NP** in Definition 2.1? Why or why not?

§9 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.

§10 Show that for every time constructible $T : \mathbb{N} \to \mathbb{N}$, 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 $O(T(n) \log T(n))$?

§11 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 2.13 is parsimonious.

(b) Show a parsimonious reduction from SAT to 3SAT.

§12 The notion of polynomial-time reducibility used in Cook's paper was somewhat different: a language A is *polynomial-time Cook reducible* to a language B if there is a polynomial time TM M that, given an *oracle* for deciding B, can decide A. (An oracle for B 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 B, see Section **??**)

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

§13 (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 a unary language is **NP**-complete then $\mathbf{P} = \mathbf{NP}$. (See Exercise 6 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 2.20.

§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*). 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 $\{i, j\}$ of G, at least one of i or j is in S. Prove that both these problems are **NP**-complete.

Hint: reduce from INDSET.

§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 $\overline{S}$. Prove that this problem is **NP**-complete.

§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 $A_1, \ldots, A_n$ 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 One3SAT 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.

§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.

§18 Let quadeq be the language of all satisfiable sets of *quadratic equations* over 0/1 variables (a quadratic equations over $u_1, \ldots, u_n$ has the form form $\sum_{a_{i,j}} u_i u_j = b$), where addition is modulo 2. Show that quadeq is **NP**-complete.

Hint: Reduce from SAT

§19 Prove that $\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{coNP}$.

§20 Prove that the Definitions 2.22 and 2.23 do indeed define the same class **coNP**.

§21 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 $L_1 \oplus L_2 = \{x : \ x \text{ is in exactly one of } L_1, L_2\}$.