Chapter 2

NP and NP completeness

“(if $\phi(n) \approx Kn^2$) 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 G"odel in a letter to John von Neumann, 1956

---

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 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 solutions can be efficiently 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 2.7.

This chapter also introduces $NP$-completeness, an important class of computational 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

As mentioned above, the complexity class \textbf{NP} will serve as our formal model for the class of problems having efficiently verifiable solutions: a decision problem / language is in \textbf{NP} if given an input $x$, we can easily verify that $x$ is a YES instance of the problem (or equivalently, $x$ is in the language) if we are given the polynomial-size solution for $x$, that certifies this fact. We will give several equivalent definitions for \textbf{NP}. The first one is as follows:

\textbf{Definition 2.1 (The class \textbf{NP})}

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

\textbf{Example 2.2}

Here are a few examples for decision problems in \textbf{NP}:

**Independent set:** (See Example 1.1 in the previous chapter.) 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 \textit{integer} numbers to $u_1, \ldots, u_m$ satisfying the inequalities. The certificate is the assignment.

\footnote{Some texts use the term \textit{witness} instead of certificate.}
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 $\pi$.

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 $t$.

2.1.1 Relation between NP and P

We have the following trivial relationships between NP and the classes P and DTIME($T(n)$) defined in the previous chapter:

Claim 2.3

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

Proof: ($P \subseteq NP$): Suppose $L \in P$ is decided in polynomial-time by a TM $N$. Then $L \in NP$ 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).

($NP \subseteq \bigcup_{c>1} \text{DTIME}(2^{n^c})$): If $L \in NP$ 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 if 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. ■

At the moment, we do not know of any stronger relation between NP and deterministic time classes than the trivial ones stated in Claim 2.3. 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 has failed to yield efficient algorithms for certain NP languages.

Example 2.4

Here is the current knowledge regarding the NP decision problems mentioned in Example 2.2: 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 [?]).
2.1. THE CLASS NP

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.

2.1.2 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 is that an NDTM has two transition functions \( \delta_0 \) and \( \delta_1 \). In addition
the NDTM has a special state we denote by \( q_{\text{accept}} \). 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 \( q_{\text{accept}} \)
on input \( x \). Otherwise— if every sequence of choices makes \( M \) halt without reaching \( q_{\text{accept}} \)— 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 \( q_{\text{accept}} \) within \( T(|x|) \) steps.

Definition 2.5
For every function \( T : \mathbb{N} \rightarrow \mathbb{N} \) and \( L \subseteq \{0,1\}^* \), we say that \( L \in \text{NTIME}(T(n)) \) 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 definition of \( NP \), the one that appears in most texts.

Theorem 2.6
\( \text{NP} = \cup_{c \in \mathbb{N}} \text{NTIME}(n^c) \)

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 \( q_{\text{accept}} \) 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
\( q_{\text{accept}} \) after using these nondeterministic choices. Thus \( L \in \text{NP} \) according to Definition 2.1.

Conversely, if \( L \in \text{NP} \) 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 \( \delta_0 \) correspond to
writing a 0 on the tape and transition \( \delta_1 \) 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 $q_{\text{accept}}$. Clearly, $N$ enters $q_{\text{accept}}$ 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 $L \in \text{NTIME}(n^c)$.

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 1. (In fact, using non-determinism we can even make the simulation by a universal TM slightly more efficient.)

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 NP ≠ 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 other language $A$? The crucial tool we use is the notion of a reduction (see Figure 2.1):

**Definition 2.7 (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") denoted by $A \leq_p B$ if there is a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$.

We say that $B$ is NP-hard if $A \leq_p B$ for every $A \in \text{NP}$. We say that $B$ is NP-complete if $B$ is NP-hard and $B \in \text{NP}$.

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 $L' = \{0,1\}^* \setminus L$ to strings in $L$. 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 of the following Theorem shows that this relation is transitive. (Later we will define other notions of reduction, and all will

---

2 Some texts call this notion “many-to-one reducibility” or “polynomial-time mapping reducibility”.

---
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.

**Theorem 2.8**

1. (Transitivity) If \( A \leq_p B \) and \( B \leq_p C \), then \( A \leq_p C \).

2. If language \( A \) is NP-hard and \( A \in P \) then \( P = NP \).

3. If language \( A \) is NP-complete then \( A \in P \) if and only if \( P = NP \).

**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 2.9**

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 \( M_\alpha \) denotes the TM represented by the string \( \alpha \).

Once you internalize the definition of NP, the proof of Theorem 2.9 is straightforward and so 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 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 found to be NP-complete.

\(^3\)Recall 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 \( k \).
2.3. THE COOK-LEVIN THEOREM: COMPUTATION IS LOCAL

2.3.1 Boolean formulae and the CNF form.

Some of the simplest examples of \( \text{NP} \)-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 (\( \land \)), NOT (\( \neg \)) and OR (\( \lor \)); see Appendix A for their definitions. For example, \((a \land b) \lor (a \land c) \lor (b \land c)\) is a Boolean formula that is \text{true} if and only if the majority of the variables \( a, b, c \) are \text{true}. If \( \varphi \) is a Boolean formula over variables \( u_1, \ldots, u_n \), and \( z \in \{0, 1\}^n \), then \( \varphi(z) \) denotes the value of \( \varphi \) when the variables of \( \varphi \) are assigned the values \( z \) (where we identify 1 with True and 0 with False). A formula \( \varphi \) is *satisfiable* if there exists some assignment \( z \) such that \( \varphi(z) \) is \text{true}. Otherwise, we say that \( \varphi \) is *unsatisfiable*.

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

\[
(u_1 \lor \bar{u}_2 \lor u_3) \land (u_2 \lor \bar{u}_3 \lor u_4) \land (\bar{u}_1 \lor u_3 \lor \bar{u}_4),
\]

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

More generally, a CNF formula has the form

\[
\bigwedge_i \left( \bigvee_j v_{ij} \right),
\]

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

2.3.2 The Cook-Levin Theorem

The following theorem provides us with our first natural \( \text{NP} \)-complete problems:

**Theorem 2.10 (Cook-Levin Theorem [?])**

Let \( \text{SAT} \) be the language of all satisfiable CNF formulae and \( 3\text{SAT} \) be the language of all satisfiable \( 3\text{CNF} \) formulae. Then,

1. \( \text{SAT} \) is \( \text{NP} \)-complete.
2. \( 3\text{SAT} \) is \( \text{NP} \)-complete.

**Remark 2.11**

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

Both \( \text{SAT} \) and \( 3\text{SAT} \) are clearly in \( \text{NP} \), since a satisfying assignment can serve as the certificate that a formula is satisfiable. Thus we only need to prove that they are \( \text{NP} \)-hard. We do so by first
proving that SAT is \textbf{NP}-hard and then showing that SAT is polynomial-time Karp reducible to 3SAT. This implies that 3SAT is \textbf{NP}-hard by the transitivity of polynomial-time reductions. Thus the following lemma is the key to the proof.

\textbf{Lemma 2.12} \\
\textit{SAT is \textbf{NP}-hard.}

Notice, to prove this we have to show how to reduce every \textbf{NP} language \( L \) to SAT, in other words give a polynomial-time transformation that turns any \( x \in \{0,1\}^* \) into a CNF formula \( \varphi_x \) such that \( x \in L \) iff \( \varphi_x \) is satisfiable. Since we know nothing about the language \( L \) except that it is in \textbf{NP}, this reduction has to rely just 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 2.12 we show how to express various conditions using CNF formulae.

\textbf{Example 2.13} \\
The formula \((a \lor b) \land (\overline{a} \lor \overline{b})\) is in CNF form. It is satisfied by only those values of \( a, b \) that are equal. Thus, the formula
\[
(x_1 \lor \overline{y}_1) \land (\overline{x}_1 \lor y_1) \land \cdots \land (x_n \lor \overline{y}_n) \land (\overline{x}_n \lor y_n)
\]
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 \( \lor \) or \( \land \), 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) \((\phi_1 \lor \overline{\phi}_2) \land (\overline{\phi}_1 \lor \phi_2)\).

In fact, CNF formulae of sufficient size can express \textit{every} Boolean condition, as shown by the following simple claim: (this fact is sometimes known as \textit{universality} of the operations \textsc{and}, \textsc{or} and \textsc{not})

\textbf{Claim 2.14} \\
For every Boolean function \( f : \{0,1\}^\ell \to \{0,1\} \) there is an \( \ell \)-variable CNF formula \( \varphi \) 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 \( \land/\lor \) symbols it contains.

\textbf{Proof Sketch:} For every \( v \in \{0,1\}^\ell \), it is not hard to see that 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 \( v = (1,1,0,1) \), the corresponding clause is \( \overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3 \lor \overline{x}_4 \).

We let \( \varphi \) be the \textsc{and} of all the clauses \( C_v \) for \( v \) such that \( f(v) = 0 \) (note that \( \varphi \) is indeed of size at most \( \ell 2^\ell \)). Then 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 2.14 only when the number of variables is some fixed constant.
2.3.4 Proof of Lemma 2.12

Let $L$ be an NP language and let $M$ be the polynomial time TM such that that for every $x \in \{0,1\}^*$, $x \in L \iff M(x, u) = 1$ 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 way to transform in polynomial-time every string $x \in \{0,1\}^*$ into a CNF formula $\varphi_x$ such that $x \in L$ iff $\varphi_x$ is satisfiable.

How can we construct such a formula $\varphi_x$? By Claim 2.14, the function that maps $u \in \{0,1\}^{p(|x|)}$ to $M(x, u)$ can be expressed as a CNF formula $\psi_x$ (i.e., $\psi_x(u) = M(x, u)$ for every $u \in \{0,1\}^{p(|x|)}$). Thus a string $u$ such that $M(x, u) = 1$ exists if and only if $\psi_x$ is satisfiable. But this is not useful for us, since the size of the formula $\psi_x$ obtained from Claim 2.14 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 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 these assumptions without loss of generality because for every $T(n)$-time TM $M$ there exists a two-tape oblivious TM $\hat{M}$ computing the same function in $O(T(n^2))$ time (see Remark 1.10 and Exercise 8 of Chapter 1). 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 $x \in L \iff \exists u \in \{0,1\}^{p(|x|)}$ s.t. $M(x, u) = 1$.

The advantage of assuming that $M$ is oblivious is that for any given input length, we can define functions $\text{inputpos}(i), \text{prev}(i)$ where $\text{inputpos}(i)$ denotes the location of the input tape head at the $i^{th}$ step and $\text{prev}(i)$ 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 $M$ on, say, the all-zeroes input.

Denote by $Q$ the set of $M$’s possible states and by $\Gamma$ its alphabet. The snapshot of $M$’s execution on some input $y$ at a particular step $i$ is the triple $(a, b, q) \in \Gamma \times \Gamma \times Q$ 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 $m \in \mathbb{N}$ and $y \in \{0,1\}^m$, the snapshot of $M$’s execution on input $y$ at the $i^{th}$ step depends on (1) its state in the $i - 1^{st}$ step and (2) the contents of the current cells of its input and work tapes. We write this relation as

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

where $\text{inputpos}(i)$ and $\text{prev}(i)$ are as defined earlier, $z_i$ is the encoding of the $i^{th}$ snapshot as a binary string of some length $c$, and $F$ is some function (derived from $M$’s transition function) that

\footnote{In 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 9 of Chapter 1. This oblivious machine may have more than two tapes, but the proof below easily generalizes to this case.}

\footnote{If $i$ is the first step that $M$ visits a certain location, then we define $\text{prev}(i) = 1$.}
2.3. THE COOK-LEVIN THEOREM: COMPUTATION IS LOCAL

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 $\langle 0, 1, q_7 \rangle$.

Let $n \in \mathbb{N}$ and $x \in \{0, 1\}^n$. We need to construct a CNF formula $\varphi_x$ such that $x \in L \iff \varphi_x \in \text{SAT}$. 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 $\langle \triangleright, \square, q_\text{start} \rangle$ where $\triangleright$ is the start symbol of the input tape, $\square$ is the blank symbol, and $q_\text{start}$ is the initial state of the TM
3. For every $i \in \{2, \ldots, T\}$, $z_i = F(z_{i-1}, z_{\text{inputpos}(i)}, z_{\text{prev}(i)})$.

4. The last string $z_T$ encodes a snapshot in which the machine halts and outputs 1.

The formula $\varphi_x$ 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 $x \in L \leftrightarrow \varphi_x \in \text{SAT}$ and so all that remains is to show that we can express $\varphi_x$ as a polynomial-sized CNF formula.

Condition 1 can be expressed as a CNF formula of size 4 (see Example 2.13). Conditions 2 and 4 each depend on $c$ variables and hence by Claim 2.14 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 $M$. ■

2.3.5 Reducing SAT to 3SAT.

Since both SAT and 3SAT are clearly in NP, Lemma 2.12 completes the proof that SAT is NP-complete. Thus all that is left to prove Theorem 2.10 is the following lemma:

**Lemma 2.15**

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

**Proof:** We will map a CNF formula $\varphi$ into a 3CNF formula $\psi$ such that $\psi$ is satisfiable if and only if $\varphi$ is. We demonstrate first the case that $\varphi$ is a 4CNF. Let $C$ be a clause of $\varphi$, say $C = u_1 \lor \overline{u}_2 \lor \overline{u}_3 \lor u_4$. We add a new variable $z$ to the $\varphi$ and replace $C$ with the pair of clauses $C_1 = u_1 \lor \overline{u}_2 \lor z$ and $C_2 = \overline{u}_3 \lor u_4 \lor z$. Clearly, if $u_1 \lor \overline{u}_2 \lor \overline{u}_3 \lor u_4$ is true then there is an assignment to $z$ that satisfies both $u_1 \lor \overline{u}_2 \lor z$ and $\overline{u}_3 \lor u_4 \lor z$ 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 $\varphi$ into an equivalent 3CNF formula $\psi$. ■

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.

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 9, Chapter 1) then for every \( x \in \{0, 1\}^* \), the size of the formula \( \varphi_x \) (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 \( \text{NP} \)-language \( L \) to \( \text{SAT} \) presented in Lemma 2.12 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 \( \text{NP} \)-complete problems (including all the ones mentioned in this chapter) have parsimonious Levin reductions from all the \( \text{NP} \) 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 \( \text{NP} \)-complete so much more interesting than the fact that, say, the language \( \text{TMSAT} \) of Theorem 2.9 is \( \text{NP} \)-complete. One answer is that 3SAT is useful for proving the \( \text{NP} \)-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.

2.4 The web of reductions

Cook and Levin had to show how every \( \text{NP} \) language can be reduced to \( \text{SAT} \). To prove the \( \text{NP} \)-completeness of any other language \( L \), we do not need to work as hard: it suffices to reduce \( \text{SAT} \) or 3SAT to \( L \). Once we know that \( L \) is \( \text{NP} \)-complete we can show that an \( \text{NP} \)-language \( L' \) is in fact \( \text{NP} \)-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 \( \text{NP} \)-complete. We now show the \( \text{NP} \)-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 2.16 (Independent set is \( \text{NP} \)-complete)**

Let \( \text{INDSET} = \{(G, k) : G \text{ has independent set of size } k\} \). Then \( \text{INDSET} \) is \( \text{NP} \)-complete.

**Proof:** Since \( \text{INDSET} \) is clearly in \( \text{NP} \), we only need to show that it is \( \text{NP} \)-hard, which we do by reducing 3SAT to \( \text{INDSET} \). Let \( \varphi \) 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 \( \varphi \). The vertices in cluster associated with a clause \( C \) correspond to the 7 possible partial 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 \( \overline{x}_2 \lor \overline{x}_5 \lor \overline{x}_7 \) then the 7 vertices in the cluster associated
Figure 2.4: Web of reductions between the \textbf{NP}-completeness problems described in this chapter and the exercises. Thousands more are known.
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 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 $u_1 = 1, u_2 = 0, u_3 = 0$ 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 $\varphi$ into $G$ can be done in polynomial time. We claim that $\varphi$ is satisfiable if and only if $G$ has a clique of size $m$. Indeed, suppose that $\varphi$ has a satisfying assignment $u$. Define a set $S$ of $m$ vertices as follows: for every clause $C$ of $\varphi$ 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 $\varphi$. 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 $\varphi$’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 2.17 (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 $u_1 \lor \overline{u}_2 \lor \overline{u}_3$ can be expressed as $u_1 + (1 - u_2) + (1 - u_3) \geq 1$. ■

**Theorem 2.18 (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 $\varphi$ into a graph $G$ such that $\varphi$ 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 $\phi$'s clauses $c_1, \ldots, c_m$, (2) a special starting vertex $v_{\text{start}}$ and ending vertex $v_{\text{end}}$ and (3) $n$ “chains” of $6m$ vertices corresponding to the $n$ variables of $\phi$. 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 $v_{\text{start}}$ 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 $v_{\text{end}}$.

In addition to these edges, for every clause $C$ of $\phi$, 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 $6m$ vertices, which is more than sufficient for this.

(\(\phi \in \text{SAT} \Rightarrow G \in \text{dHAMPATH}\).) Suppose that $\phi$ 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 $v_{\text{start}}$, travel through all the chains in order, and end at $v_{\text{end}}$. 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 \(v_{\text{start}}\) (as it has no incoming edges) and end at \(v_{\text{end}}\) (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 \(u_1, \ldots, u_n\) to \(\varphi\) 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 \(\varphi\). ■

In praise of reductions

Though originally invented as part of the theory of \(\text{NP}\)-completeness, the polynomial-time reduction (together with its first cousin, the randomized polynomial-time reduction defined in Section 7.9) has led to a rich understanding of complexity above and beyond \(\text{NP}\)-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.

Coping with \(\text{NP}\) hardness.

\(\text{NP}\)-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 \(\text{NP}\)-complete? On the outset, the situation looks bleak: if \(\text{P} \neq \text{NP}\) then there simply does not exist an efficient algorithm to solve such a problem. However, there may still be some hope: \(\text{NP}\) completeness only means that (assuming \(\text{P} \neq \text{NP}\)) 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 salespersons that need to visit various cities around your country. Does the fact that TSP is \(\text{NP}\)-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
2.5 Decision versus search

We have chosen to define the notion of \( \mathsf{NP} \) using \( \text{Yes}/\text{No} \) problems ("Is the given formula satisfiable?") as opposed to \textit{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 \( \mathsf{P} \neq \mathsf{NP} \) then neither one can be solved for an \( \mathsf{NP} \)-complete problem. However, it turns out that for \( \mathsf{NP} \)-complete problems they are equivalent in the sense that if the decision problem can be solved (and hence \( \mathsf{P} = \mathsf{NP} \)) then the search version of any \( \mathsf{NP} \) problem can also be solved in polynomial time.

\textbf{Theorem 2.19}

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

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

\textbf{Proof:} We start by showing the theorem for the case of \( \text{SAT} \). In particular we show that given an algorithm \( A \) that decides \( \text{SAT} \), we can come up with an algorithm \( B \) that on input a satisfiable \( \text{CNF} \) formula \( \varphi \) with \( n \) variables, finds a satisfying assignment for \( \varphi \) 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 \( \varphi \) is satisfiable. If so, we substitute \( x_1 = 0 \) and \( x_1 = 1 \) in \( \varphi \) (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 \( \varphi \).
To solve the search problem for an arbitrary \( \text{NP} \)-language \( L \), we use the fact that the reduction of Theorem 2.10 from \( L \) to \( \text{SAT} \) is actually a Levin reduction. This means that we have a polynomial-time computable function \( f \) such that not only \( x \in L \iff f(x) \in \text{SAT} \) 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 2.20**
The proof above shows that \( \text{SAT} \) is downward self-reducible, which means that given an algorithm that solves \( \text{SAT} \) on inputs of length smaller than \( n \) we can solve \( \text{SAT} \) on inputs of length \( n \). Using the Cook-Levin reduction, one can show that all \( \text{NP} \)-complete problems have a similar property, though we do not make this formal.

### 2.6 \( \text{coNP} \), \( \text{EXP} \) and \( \text{NEXP} \)

Now we define some related complexity classes that are very relevant to the study of the \( \text{P} \) versus \( \text{NP} \) question.

#### 2.6.1 \( \text{coNP} \)

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 2.21**
\[
\text{coNP} = \{ L : \overline{L} \in \text{P} \}.
\]

It is important to note that \( \text{coNP} \) is not the complement of the class \( \text{NP} \). In fact, they have a non-empty intersection, since every language in \( \text{P} \) is in \( \text{NP} \cap \text{coNP} \) (see Exercise 19). The following is an example of a \( \text{coNP} \) language: \( \text{SAT} = \{ \varphi : \varphi \text{ is not satisfiable} \} \). Students sometimes mistakenly convince themselves that \( \text{SAT} \) is in \( \text{NP} \). They have the following polynomial time NDTM in mind: on input \( \varphi \), the machine guesses an assignment. If this assignment does not satisfy \( \varphi \) then it accepts (i.e., goes into \( q_{\text{accept}} \) and halts) and if it does satisfy \( \varphi \) then the machine halts without accepting. This NDTM does not do the job: indeed it accepts every unsatisfiable \( \varphi \) 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 \( \text{coNP} \) (which is easily shown to be equivalent to the first, see Exercise 20):

**Definition 2.22 (\( \text{coNP} \), alternative definition)**

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

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

The key fact to note is the use of “\( \forall \)” in this definition where Definition 2.1 used \( \exists \).

We can define \( \text{coNP} \)-completeness in analogy to \( \text{NP} \)-completeness: a language is \( \text{coNP} \)-complete if it is in \( \text{coNP} \) and every \( \text{coNP} \) language is polynomial-time Karp reducible to it.
Example 2.23

In classical logic, tautologies are true statements. The following language is coNP-complete:

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

It is clearly in coNP by Definition 2.22 and so all we have to show is that for every \( L \in \text{coNP} \), \( L \leq p \text{TAUTOLOGY} \). But this is easy: just modify the Cook-Levin reduction from \( L \) (which is in NP) to SAT. For every input \( x \in \{0, 1\}^* \) that reduction produces a formula \( \varphi_x \) that is satisfiable iff \( x \in L \). Now consider the formula \( \neg \varphi_x \). It is in TAUTOLOGY iff \( x \in L \), and this completes the description of the reduction.

It is a simple exercise to check that if \( P = \text{NP} \) then \( \text{NP} = \text{coNP} = P \). Put in the contrapositive, if we can show that \( \text{NP} \neq \text{coNP} \) then we have shown \( P \neq \text{NP} \). Most researchers believe that \( \text{NP} \neq \text{coNP} \). The intuition is almost as strong as for the \( P \) versus \( \text{NP} \) 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 \( P \) and \( \text{NP} \).

**Definition 2.24**

\[
\text{EXP} = \bigcup_{c \geq 0} \text{DTIME}(2^{n^c}).
\]

\[
\text{NEXP} = \bigcup_{c \geq 0} \text{NTIME}(2^{n^c}).
\]

Because every problem in \( \text{NP} \) can be solved in exponential time by a brute force search for the certificate, \( P \subseteq \text{NP} \subseteq \text{EXP} \subseteq \text{NEXP} \). 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 2.25**

If \( \text{EXP} \neq \text{NEXP} \) then \( P \neq \text{NP} \).

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

\[
L_{\text{pad}} = \{ \langle x, 1^{2^{|x|}} \rangle : x \in L \}
\]

is in \( \text{NP} \). Here is an NDTM for \( L_{\text{pad}} \): given \( y \), first check if there is a string \( z \) such that \( y = \langle z, 1^{2^{|z|}} \rangle \). If not, output REJECT. If \( y \) is of this form, then run \( M \) on \( z \) for \( 2^{|z|} \) steps and output its answer. Clearly, the running time is polynomial in \( |y| \), and hence \( L_{\text{pad}} \in \text{NP} \). Hence if \( P = \text{NP} \) then \( L_{\text{pad}} \) is in \( P \). But if \( L_{\text{pad}} \) is in \( P \) then \( L \) is in \( \text{EXP} \): to determine whether an input \( x \) is in \( L \), we just pad the input and decide whether it is in \( L_{\text{pad}} \) using the polynomial-time machine for \( L_{\text{pad}} \).
Remark 2.26
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 resource bound $T'(n)$ “scale down” to resource bound $T(n)$.

Like $P$ and $NP$, most of the complexity classes studied later are also contained in both $EXP$ and $NEXP$.

2.7 More thoughts about $P$, $NP$, and all that

2.7.1 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 G"odel’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 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 $A$:

$$\text{THEOREMS} = \{(\varphi, 1^n) : \varphi \text{ has a formal proof of length } \leq n \text{ in system } A\}.$$
In fact, the exercises ask you to prove that this problem is \textbf{NP}-complete. Hence the \textbf{P} versus \textbf{NP} question is a \textit{rephrasing} of Gödel’s question (see quote at the beginning of the chapter), which asks whether or not there is an 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 \textbf{P} \neq \textbf{NP}.

\subsection*{2.7.4 What if \textbf{NP} = \textbf{coNP}?}

If \textbf{NP} = \textbf{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 \textbf{NP}-complete problem of finding whether or not a set of multivariate polynomials has a common
2.7. MORE THOUGHTS ABOUT P, NP, AND ALL THAT

root, in other words, deciding whether a system of equations of the following type has a solution:

\[ f_1(x_1, \ldots, x_n) = 0 \]
\[ f_2(x_1, \ldots, x_n) = 0 \]
\[ \vdots \]
\[ f_m(x_1, \ldots, x_n) = 0 \]

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 \( \text{coNP} \). 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 \( \text{coNP} = \text{NP} \)? 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 \( \text{NP} = \text{coNP} \) 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 \( \text{coNP} \) problem in the above discussion.

What have we learned?

- The class \( \text{NP} \) 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 \( \text{P} \). \( \text{NP} \) can also be defined using non-deterministic Turing machines.

- \( \text{NP} \)-complete problems are the hardest problems in \( \text{NP} \), in the sense that they have a polynomial-time algorithm if and only if \( \text{P} = \text{NP} \). Many natural problems that seemingly have nothing to do with Turing machines turn out to be \( \text{NP} \)-complete. One such example is the language 3SAT of satisfiable Boolean formulae in 3CNF form.

- If \( \text{P} = \text{NP} \) 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

In the 1950’s, Soviet scientists were aware of the undesirability of using exhaustive or brute force search, which they called \textit{perebor}, for combinatorial problems, and asked the question of whether
certain problems \emph{inherently} require such search (see \cite{?}). In the west the first published description of this issue is by Edmonds \cite{?}, 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 \textbf{NP} and \textbf{P}. Amazingly, in his 1956 letter to von Neumann we quoted above, Gödel essentially asks the question of \textbf{P} vs. \textbf{NP}, although there is no hint that he realized that one of the particular problems he mentions is \textbf{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 problem, and the letter was only discovered in the 1980’s.

In 1971 Cook published his seminal paper defining the notion of \textbf{NP}-completeness and showing that \textbf{SAT} is \textbf{NP} complete \cite{?}. Soon afterwards, Karp \cite{?} showed that 21 important problems are in fact \textbf{NP}-complete, generating tremendous interest in this notion. Meanwhile in the USSR Levin independently defined \textbf{NP}-completeness (although he focused on search problems) and showed that a variant of \textbf{SAT} is \textbf{NP}-complete. (Levin’s paper \cite{?} 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 \cite{?} for more on the history of \textbf{P} and \textbf{NP} and a full translation of Gödel’s remarkable letter.

The “TSP book” by Lawler et al. \cite{?} 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 \emph{Euclidean Steiner Tree} problem —today known to be \textbf{NP}-hard— in the early 19th century.

As mentioned above, the book by Garey and Johnson \cite{?} and the web site \cite{?} contain many more examples of \textbf{NP} complete problem. Also, Aaronson \cite{?} surveys various attempts to solve \textbf{NP} complete problems via “non-traditional” computing devices.

Even if \textbf{NP} \neq \textbf{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, \textbf{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 \emph{average-case} complexity. Also, Impagliazzo \cite{?} has an excellent survey on this topic.

\section*{Exercises}

§1 Prove the existence of a \emph{non-deterministic Universal TM} (analogously to the deterministic universal TM of Theorem 1.13). That is, prove that there exists a representation scheme of \textbf{NDTM}s, and an \textbf{NDTM} \textbf{NU} such that for every string \(\alpha\), and input \(x\), \textbf{NU}(x,\alpha) = M_{\alpha}(x).

(a) Prove that there exists such a universal \textbf{NDTM} \textbf{NU} such that if \(M_{\alpha}\) halts on \(x\) within \(T\) steps, then \textbf{NU} halts on \(x,\alpha\) within \(CT\log T\) steps (where \(C\) is a constant depending only on the machine represented by \(\alpha\)).

(b) Prove that there is such a universal \textbf{NDTM} that runs on these inputs for at most \(Ct\) steps.
Hint: A simulation in $O\left(\alpha t \log t\right)$ time can be obtained by a straightforward adaptation of the proof of Theorem 1.13. To do a more efficient simulation, the main idea is to first run a simulation uninformalized in the proof of Theorem 1.13. To do a simulation in time can be obtained by a

§2 Prove Theorem 2.

§3 Let HALT be the Halting language defined in Theorem 1.17. Show that HALT is NP-hard. Is it 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 $L_1, L_2 \in \text{NP}$. 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.

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

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. Prove that if this problem is in EXP then NEXP = EXP.

§10 Show that for every time constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \text{NTIME}(T(n))$ 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 2.12 is parsimonious.
2.7. MORE THOUGHTS ABOUT \( P, \ NP, \) AND ALL THAT

(b) Show a parsimonious reduction from \( \text{SAT} \) to \( 3\text{SAT}. \)

§12 The notion of polynomial-time reducibility used in Cook’s paper was somewhat different: a language \( A \) is \textit{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 overwitten 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 \( 3\text{SAT} \) is Cook-reducible to TAUTOLOGY.

§13 (Berman’s Theorem 1978) A language is called \textit{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 \( P = \NP \). (See Exercise 6 of Chapter 6 for a strengthening of this result.)

\textit{Hint}: If there is a \( n^c \) time reduction from \( 3\text{SAT} \) to a unary lan-

§14 In the \textsc{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 \textit{clique}). In the \textsc{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.

\textit{Hint}: Reduce from \textsc{indset}.

§15 In the \textsc{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 \textsc{exactly one 3sat} problem, we are given a 3CNF formula \( \varphi \) and need to decide if there exists a satisfying assignment \( u \) for \( \varphi \) such that every clause of \( \varphi \) has exactly one \textit{true} literal. In the \textsc{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 \textsc{exactly one3sat} and \textsc{subset sum} are \NP-complete.
2.7. MORE THOUGHTS ABOUT \( P \), \( NP \), AND ALL THAT

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 \( \text{SUBSET SUM} \) is that given a formula \( \varphi \), we map it to a \( \text{SUBSET SUM} \) instance by mapping each possible literal \( u_i \) to the number \( \sum_{j \in S_i} (2^n)^j \) where \( S_i \) is the set of clauses that the literal \( u_i \) satisfies, and setting the target \( T \) to be \( \sum_{m \in 1} (2^n)^j \). 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 \( \text{HAMPATH} \) of undirected graphs with Hamiltonian paths is \( NP \)-complete. Prove that the language \( \text{TSP} \) described in Example 2.2 is \( NP \)-complete. Prove that the language \( \text{HAMCYCLE} \) of undirected graphs that contain Hamiltonian cycle (a simple cycle involving all the vertices) is \( NP \)-complete.

§18 Let \( \text{quadeq} \) 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 \( \sum a_{i,j} u_i u_j = b \), where addition is modulo 2. Show that \( \text{quadeq} \) is \( NP \)-complete.

Hint: Reduce from \( \text{SAT} \)

§19 Prove that \( P \subseteq NP \cap coNP \).

§20 Prove that the Definitions 2.21 and 2.22 do indeed define the same class \( coNP \).

§21 Suppose \( L_1, L_2 \in NP \cap coNP \). Then show that \( L_1 \oplus L_2 \) is in \( NP \cap coNP \), where \( L_1 \oplus L_2 = \{ x : x \text{ is in exactly one of } L_1, L_2 \} \).

§22 Define the language \( UNARY \text{ SUBSET SUM} \) to be the variant of the \( \text{SUBSET SUM} \) problem of Exercise 16 where all numbers are represented by the unary representation (i.e., the number \( k \) is represented as \( 1^k \)). Show that \( UNARY \text{ SUBSET SUM} \) is in \( P \).

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

§23 Prove that if every unary \( NP \)-language is in \( P \) then \( EXP = NEXP \). (A language \( L \) is unary if it is a subset of \( \{1\}^* \), see Exercise 13.)