Introduction

As long as a branch of science offers an abundance of problems, so long it is alive; a lack of problems foreshadows extinction or the cessation of independent development.

– David Hilbert, 1900

The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? … I (would like to) show that there is no algorithm for multiplication computationally as simple as that for addition, and this proves something of a stumbling block.

– Alan Cobham, 1964

The notion of *computation* has existed in some form for thousands of years, in contexts as varied as routine account keeping and astronomy. Here are three examples of tasks that we may wish to solve using computation:

- Given two integer numbers, compute their product.
- Given a set of n linear equations over n variables, find a solution, if it exists.
- Given a list of acquaintances and a list of all pairs among them who do not get along, find the largest set of acquaintances you can invite to a dinner party such that every two invitees get along with one another.

Throughout history people had a notion of a process of producing an output from a set of inputs in a finite number of steps, and they thought of "computation" as "a person writing numbers on a scratch pad following certain rules."

One of the important scientific advances in the first half of the twentieth century was that the notion of "computation" received a much more precise definition. From this definition, it quickly became clear that computation can happen in diverse physical and mathematical systems—Turing machines, lambda calculus, cellular automata, pointer machines, bouncing billiards balls, Conway's *Game of life*, and so on. Surprisingly, all these forms of computation are equivalent—in the sense that each model is capable of implementing all computations that we can conceive of on any other model (see Chapter 1). This realization quickly led to the invention of the standard *universal electronic computer*, a piece of hardware that is capable of executing all possible programs. The computer's rapid adoption in society in the subsequent decades brought computation into every aspect of modern life and made computational issues important in

design, planning, engineering, scientific discovery, and many other human endeavors. Computer *algorithms*, which are methods of solving computational problems, became ubiquitous.

But computation is not "merely" a practical tool. It is also a major scientific concept. Generalizing from physical models such as cellular automata, scientists now view many natural phenomena as akin to computational processes. The understanding of reproduction in living things was triggered by the discovery of self-reproduction in computational machines. (In fact, a book by the physicist Schroedinger [Sch44] predicted the existence of a DNA-like substance in cells before Watson and Crick discovered it and was credited by Crick as an inspiration for that research.) Today, computational models underlie many research areas in biology and neuroscience. Several physics theories such as QED give a description of nature that is very reminiscent of computation, motivating some scientists to even suggest that the entire universe may be viewed as a giant computer (see Lloyd [Llo06]). In an interesting twist, such physical theories have been used in the past decade to design a model for *quantum computation*; see Chapter 10.

Computability versus complexity

After their success in defining computation, researchers focused on understanding what problems are *computable*. They showed that several interesting tasks are *inherently uncomputable*: No computer can solve them without going into infinite loops (i.e., never halting) on certain inputs. Though a beautiful topic, computability will not be our focus in this book. We discuss it briefly in Chapter 1 and refer the reader to standard texts [Sip96, HMU01, Koz97, Rog87] for more details. Instead, we focus on *computational complexity theory*, which focuses on issues of *computational efficiency*—quantifying the amount of computational resources required to solve a given task. In the next section, we describe at an informal level how one can quantify *efficiency*, and after that we discuss some of the issues that arise in connection with its study.

QUANTIFYING COMPUTATIONAL EFFICIENCY

To explain what we mean by *computational efficiency*, we use the three examples of computational tasks we mentioned earlier. We start with the task of multiplying two integers. Consider two different methods (or *algorithms*) to perform this task. The first is *repeated addition*: to compute $a \cdot b$, just add a to itself $b - 1$ times. The other is the *grade-school algorithm* illustrated in Figure I.1. Though the repeated addition algorithm is perhaps simpler than the grade-school algorithm, we somehow feel that

$$
\begin{array}{cccccc}
 & & & 5 & 7 & 7 \\
 & & & 4 & 2 & 3 \\
\hline
 & & 1 & 7 & 3 & 1 \\
 & 1 & 1 & 5 & 4 & \\
2 & 3 & 0 & 8 & & \\
\hline
2 & 4 & 4 & 0 & 7 & 1
\end{array}
$$

Figure I.1. Grade-school algorithm for multiplication. Illustrated for computing $577 \cdot 423$.

the latter is *better*. Indeed, it is much more efficient. For example, multiplying 577 by 423 using repeated addition requires 422 additions, whereas doing it with the grade-school algorithm takes 3 multiplications of a number by a single digit and 3 additions.

We will quantify the efficiency of an algorithm by studying how its number of *basic operations* scales as we increase the *size* of the input. For this discussion, let the basic operations be addition and multiplication of single digits. (In other settings, we may wish to throw in division as a basic operation.) The *size* of the input is the number of digits in the numbers. The number of basic operations used to multiply two n-digit numbers (i.e., numbers between 10^{n-1} and 10^n) is at most $2n^2$ for the grade-school algorithm and at least $n10^{n-1}$ for repeated addition. Phrased this way, the huge difference between the two algorithms is apparent: Even for 11-digit numbers, a pocket calculator running the grade-school algorithm would beat the best current supercomputer running the repeated addition algorithm. For slightly larger numbers even a fifth grader with pen and paper would outperform a supercomputer. We see that *the efficiency of an algorithm is to a considerable extent much more important than the technology used to execute it.*

Surprisingly enough, there is an even faster algorithm for multiplication that uses the *Fast Fourier Transform*. It was only discovered some 40 years ago and multiplies two n-digit numbers using $cn \log n \log\log n$ operations, where c is some absolute constant independent of n; see Chapter 16. We call such an algorithm an $O(n \log n \log\log n)$-step algorithm: see our notational conventions below. As n grows, this number of operations is significantly smaller than n^2.

For the task of solving linear equations, the classic *Gaussian elimination* algorithm (named after Gauss but already known in some form to Chinese mathematicians of the first century) uses $O(n^3)$ basic arithmetic operations to solve n equations over n variables. In the late 1960s, Strassen found a more efficient algorithm that uses roughly $O(n^{2.81})$ operations, and the best current algorithm takes $O(n^{2.376})$ operations; see Chapter 16.

The dinner party task also has an interesting story. As in the case of multiplication, there is an obvious and simple inefficient algorithm: Try all possible subsets of the n people from the largest to the smallest, and stop when you find a subset that does not include any pair of guests who don't get along. This algorithm can take as much time as the number of subsets of a group of n people, which is 2^n. This is highly unpractical—an organizer of, say, a 70-person party, would need to plan it at least a thousand years in advance, even if she has a supercomputer at her disposal. Surprisingly, we still do not know of a significantly better algorithm for this task. In fact, as we will see in Chapter 2, we have reasons to suspect that no efficient algorithm *exists*, because this task turns out to be equivalent to the *independent set* computational problem, which, together with thousands of other important problems, is **NP**-complete. The famous "**P** versus **NP**" question (Chapter 2) asks whether or not any of these problems has an efficient algorithm.

PROVING NONEXISTENCE OF EFFICIENT ALGORITHMS

We have seen that sometimes computational tasks turn out to have nonintuitive algorithms that are more efficient than algorithms used for thousands of years. It would

therefore be really interesting to prove for some computational tasks that the current algorithm is the *best*—in other words, no better algorithms exist. For instance, we could try to prove that the $O(n \log n \log \log n)$-step algorithm for multiplication cannot be improved upon (thus implying that multiplication is inherently more difficult than addition, which does have an $O(n)$-step algorithm). Or, we could try to prove that there is no algorithm for the dinner party task that takes fewer than $2^{n/10}$ steps. Trying to prove such results is a central goal of complexity theory.

How can we ever prove such a nonexistence result? There are infinitely many possible algorithms! So we have to *mathematically prove* that each one of them is less efficient that the known algorithm. This may be possible because computation is a mathematically precise notion. In fact, this kind of result (if proved) would fit into a long tradition of *impossibility results* in mathematics, such as the independence of Euclid's parallel postulate from the other basic axioms of geometry, or the impossibility of trisecting an arbitrary angle using a compass and straightedge. Such results count among the most interesting, fruitful, and surprising results in mathematics.

In complexity theory, we are still only rarely able to prove such nonexistence of algorithms. We do have important nonexistence results in some concrete computational models that are not as powerful as general computers, which are described in Part II of the book. Because we are still missing good results for general computers, one important source of progress in complexity theory is our stunning success in *interrelating* different complexity questions, and the rest of the book is filled with examples of these.

SOME INTERESTING QUESTIONS ABOUT COMPUTATIONAL EFFICIENCY

Now we give an overview of some important issues regarding computational complexity, all of which will be treated in greater detail in later chapters. An overview of mathematical background is given in Appendix A.

1. Computational tasks in a variety of disciplines such as the life sciences, social sciences, and operations research involve searching for a solution across a vast space of possibilities (e.g., the aforementioned tasks of solving linear equations and finding a maximal set of invitees to a dinner party). This is sometimes called *exhaustive search*, since the search *exhausts* all possibilities. Can this exhaustive search be replaced by a more *efficient* search algorithm?

   As we will see in Chapter 2, this is essentially the famous **P** vs. **NP** question, considered *the* central problem of complexity theory. Many interesting search problems are **NP**-complete, which means that if the famous conjecture $\mathbf{P} \neq \mathbf{NP}$ is true, then these problems do not have efficient algorithms; they are *inherently intractable*.
2. Can algorithms use randomness (i.e., coin tossing) to speed up computation?

   Chapter 7 introduces randomized computation and describes efficient *probabilistic algorithms* for certain tasks. But Chapters 19 and 20 show a surprising recent result giving strong evidence that randomness does *not* help speed up computation too much, in the sense that any probabilistic algorithm can be replaced with a *deterministic* algorithm (tossing no coins) that is almost as efficient.

3. Can hard problems become easier to solve if we allow the algorithms to err on a small number of inputs, or to only compute an *approximate* solution?

   Average-case complexity and *approximation algorithms* are studied in Chapters 11, 18, 19, and 22. These chapters also show fascinating connections between these questions, the power of randomness, different notions of mathematical proofs, and the theory of error correcting codes.
4. Can we derive any practical benefit from computationally hard problems? For example, can we use them to construct cryptographic protocols that are *unbreakable* (at least by any plausible adversary)?

   As described in Chapter 9, the security of digital cryptography is intimately related to the **P** vs. **NP** question (see Chapter 2) and average-case complexity (see Chapters 18).
5. Can we use the counterintuitive quantum mechanical properties of matter to build faster computers?

   Chapter 10 describes the fascinating notion of *quantum computers* that use quantum mechanics to speed up certain computations. Peter Shor has shown that, if ever built, quantum computers will be able to factor integers efficiently (thus breaking many current cryptosystems). However, currently there are many daunting obstacles to actually building such computers.
6. Do we need people to prove mathematical theorems, or can we generate mathematical proofs automatically? Can we check a mathematical proof without reading it completely? Do interactive proofs, involving a dialog between prover and verifier, have more power than standard "static" mathematical proofs?

   The notion of *proof*, central to mathematics, turns out to be central to computational complexity as well, and complexity has shed new light on the meaning of mathematical proofs. Whether mathematical proofs can be generated automatically turns out to depend on the **P** vs. **NP** question (see Chapter 2). Chapter 11 describes *probabilistically checkable proofs*. These are surprisingly robust mathematical proofs that can be checked simply by reading them in very few probabilistically chosen locations, in contrast to the traditional proofs that require line-by-line verification. Along similar lines we introduce the notion of *interactive proofs* in Chapter 8 and use them to derive some surprising results. Finally, *proof complexity*, a subfield of complexity studying the minimal proof length of various statements, is studied in Chapter 15.

At roughly 40 years, complexity theory is still an infant science, and many important results are less than 20 years old. We have few complete answers for any of these questions. In a surprising twist, computational complexity has also been used to prove some metamathematical theorems: They provide evidence of the difficulty of resolving some of the questions of . . . computational complexity; see Chapter 23.
We conclude with another quote from Hilbert's 1900 lecture:

> *Proofs of impossibility were effected by the ancients . . . [and] in later mathematics, the question as to the impossibility of certain solutions plays a preminent part. . . .*
>
> *In other sciences also one meets old problems which have been settled in a manner most satisfactory and most useful to science by the proof of their impossibility. . . . After seeking in vain for the construction of a perpetual motion machine, the relations were investigated which must subsist between the forces of nature if such a machine is to be impossible; and this inverted question led to the discovery of the law of the conservation of energy. . . .*

It is probably this important fact along with other philosophical reasons that gives rise to conviction . . . that every definite mathematical problem must necessarily be susceptible to an exact settlement, either in the form of an actual answer to the question asked, or by the proof of the impossibility of its solution and therewith the necessary failure of all attempts. . . . This conviction . . . is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorance.

CHAPTER 0 Notational conventions

We now specify some of the notations and conventions used throughout this book. We make use of some notions from discrete mathematics such as strings, sets, functions, tuples, and graphs. All of these are reviewed in Appendix A.

Standard notation

We let $\mathbb{Z} = \{0, \pm 1, \pm 2, \ldots\}$ denote the set of integers, and $\mathbb{N}$ denote the set of natural numbers (i.e., nonnegative integers). A number denoted by one of the letters i, j, k, ℓ, m, n is always assumed to be an integer. If $n \geq 1$, then $[n]$ denotes the set $\{1, \ldots, n\}$. For a real number x, we denote by $\lceil x \rceil$ the smallest $n \in \mathbb{Z}$ such that $n \geq x$ and by $\lfloor x \rfloor$ the largest $n \in \mathbb{Z}$ such that $n \leq x$. Whenever we use a real number in a context requiring an integer, the operator $\lceil \rceil$ is implied. We denote by $\log x$ the logarithm of x to the base 2. We say that a condition $P(n)$ holds for *sufficiently large* n if there exists some number N such that $P(n)$ holds for every $n > N$ (for example, $2^n > 100n^2$ for sufficiently large n). We use expressions such as $\sum_i f(i)$ (as opposed to, say, $\sum_{i=1}^{n} f(i)$) when the range of values i takes is obvious from the context. If u is a string or vector, then u_i denotes the value of the i^{th} symbol/coordinate of u.

Strings

If S is a finite set then a *string* over the alphabet S is a finite ordered tuple of elements from S. In this book we will typically consider strings over the *binary* alphabet $\{0, 1\}$. For any integer $n \geq 0$, we denote by S^n the set of length-n strings over S (S^0 denotes the singleton consisting of the empty tuple). We denote by S^* the set of all strings (i.e., $S^* = \cup_{n \geq 0} S^n$). If x and y are strings, then we denote their concatenation (the tuple that contains first the elements of x and then the elements of y) by $x \circ y$ or sometimes simply xy. If x is a string and $k \geq 1$ is a natural number, then x^k denotes the concatenation of k copies of x. For example, 1^k denotes the string consisting of k ones. The length of a string x is denoted by $|x|$.

Additional notation

If S is a distribution then we use $x \in_R S$ to say that x is a random variable that is distributed according to S; if S is a set then this denotes that x is distributed uniformly over the members of S. We denote by U_n the uniform distribution over $\{0, 1\}^n$. For two

length-n strings $x, y \in \{0,1\}^n$, we denote by $x \odot y$ their dot product modulo 2; that is $x \odot y = \sum_i x_i y_i \pmod 2$. In contrast, the inner product of two n-dimensional real or complex vectors $\mathbf{u}, \mathbf{v}$ is denoted by $\langle \mathbf{u}, \mathbf{v} \rangle$ (see Section A.5.1). For any object x, we use $\llcorner x \lrcorner$ (not to be confused with the floor operator $\lfloor x \rfloor$) to denote the representation of x as a string (see Section 0.1).

0.1 REPRESENTING OBJECTS AS STRINGS

The basic computational task considered in this book is *computing a function*. In fact, we will typically restrict ourselves to functions whose inputs and outputs are finite *strings of bits* (i.e., members of $\{0,1\}^*$).

Representation

Considering only functions that operate on bit strings is not a real restriction since simple encodings can be used to *represent* general objects—integers, pairs of integers, graphs, vectors, matrices, etc.—as strings of bits. For example, we can represent an integer as a string using the binary expansion (e.g., 34 is represented as 100010) and a graph as its adjacency matrix (i.e., an n vertex graph G is represented by an $n \times n$ 0/1-valued matrix A such that $A_{i,j} = 1$ iff the edge $\overline{ij}$ is present in G). We will typically avoid dealing explicitly with such low-level issues of representation and will use $\llcorner x \lrcorner$ to denote some canonical (and unspecified) binary representation of the object x. Often we will drop the symbols $\llcorner \lrcorner$ and simply use x to denote both the object and its representation.

Representing pairs and tuples

We use the notation $\langle x, y \rangle$ to denote the ordered pair consisting of x and y. A canonical representation for $\langle x, y \rangle$ can be easily obtained from the representations of x and y. For example, we can first encode $\langle x, y \rangle$ as the string $\llcorner x \lrcorner \# \llcorner y \lrcorner$ over the alphabet $\{0, 1, \#\}$ and then use the mapping $0 \mapsto 00, 1 \mapsto 11, \# \mapsto 01$ to convert this representation into a string of bits. To reduce notational clutter, instead of $\llcorner \langle x, y \rangle \lrcorner$ we use $\langle x, y \rangle$ to denote not only the pair consisting of x and y but also the representation of this pair as a binary string. Similarly, we use $\langle x, y, z \rangle$ to denote both the ordered triple consisting of x, y, z and its representation, and use similar notation for 4-tuples, 5-tuples, etc.

Computing functions with nonstring inputs or outputs

The idea of representation allows us to talk about computing functions whose inputs are not strings (e.g., functions that take natural numbers as inputs). In all these cases, we implicitly identify any function f whose domain and range are not strings with the function $g : \{0,1\}^* \to \{0,1\}^*$ that given a representation of an object x as input, outputs the representation of $f(x)$. Also, using the representation of pairs and tuples, we can also talk about computing functions that have more than one input or output.

0.2 DECISION PROBLEMS/LANGUAGES

An important special case of functions mapping strings to strings is the case of *Boolean* functions, whose output is a single bit. We identify such a function f with the subset $L_f = \{x : f(x) = 1\}$ of $\{0,1\}^*$ and call such sets *languages* or *decision problems* (we use these terms interchangeably).[1] We identify the computational problem of computing f (i.e., given x compute $f(x)$) with the problem of deciding the language L_f (i.e., given x, decide whether $x \in L_f$).

EXAMPLE 0.1

By representing the possible invitees to a dinner party with the vertices of a graph having an edge between any two people who don't get along, the dinner party computational problem from the introduction becomes the problem of finding a maximum sized *independent set* (set of vertices without any common edges) in a given graph. The corresponding language is:

$$\texttt{INDSET} = \{\langle G,k\rangle : \exists S \subseteq V(G) \text{ s.t. } |S| \geq k \text{ and } \forall u,v \in S, \overline{uv} \notin E(G)\}$$

An algorithm to solve this language will tell us, on input a graph G and a number k, whether there exists a conflict-free set of invitees, called an *independent set*, of size at least k. It is not immediately clear that such an algorithm can be used to actually find such a set, but we will see this is the case in Chapter 2. For now, let's take it on faith that this is a good formalization of this problem.

0.3 BIG-OH NOTATION

We will typically measure the computational efficiency of an algorithm as the number of a basic operations it performs as *a function of its input length*. That is, the efficiency of an algorithm can be captured by a function T from the set $\mathbb{N}$ of natural numbers to itself such that $T(n)$ is equal to the maximum number of basic operations that the algorithm performs on inputs of length n. However, this function T is sometimes overly dependant on the low-level details of our definition of a basic operation. For example, the addition algorithm will take about three times more operations if it uses addition of single digit *binary* (i.e., base 2) numbers as a basic operation, as opposed to *decimal* (i.e., base 10) numbers. To help us ignore these low-level details and focus on the big picture, the following well-known notation is very useful.

Definition 0.2 *(Big-Oh notation)* If f, g are two functions from $\mathbb{N}$ to $\mathbb{N}$, then we (1) say that $f = O(g)$ if there exists a constant c such that $f(n) \leq c \cdot g(n)$ for every sufficiently

[1] The word "language" is perhaps not an ideal choice to denote subsets of $\{0,1\}^*$, but for historical reasons this is by now standard terminology.

large n, (2) say that $f = \Omega(g)$ if $g = O(f)$, (3) say that $f = \Theta(g)$ is $f = O(g)$ and $g = O(f)$, (4) say that $f = o(g)$ if for every $\epsilon > 0$, $f(n) \leq \epsilon \cdot g(n)$ for every sufficiently large n, and (5) say that $f = \omega(g)$ if $g = o(f)$.

To emphasize the input parameter, we often write $f(n) = O(g(n))$ instead of $f = O(g)$, and use similar notation for $o, \Omega, \omega, \Theta$.

EXAMPLE 0.3

Here are some examples for use of big-Oh notation:

1. If $f(n) = 100n \log n$ and $g(n) = n^2$ then we have the relations $f = O(g)$, $g = \Omega(f)$, $f = o(g)$, $g = \omega(f)$.
2. If $f(n) = 100n^2 + 24n + 2 \log n$ and $g(n) = n^2$ then $f = O(g)$. We will often write this relation as $f(n) = O(n^2)$. Note that we also have the relation $g = O(f)$ and hence $f = \Theta(g)$ and $g = \Theta(f)$.
3. If $f(n) = \min\{n, 10^6\}$ and $g(n) = 1$ for every n then $f = O(g)$. We use the notation $f = O(1)$ to denote this. Similarly, if h is a function that tends to infinity with n (i.e., for every c it holds that $h(n) > c$ for n sufficiently large) then we write $h = \omega(1)$.
4. If $f(n) = 2^n$ then for every number $c \in \mathbb{N}$, if $g(n) = n^c$ then $g = o(f)$. We sometimes write this as $2^n = n^{\omega(1)}$. Similarly, we also write $h(n) = n^{O(1)}$ to denote the fact that h is bounded from above by some polynomial. That is, there exist a number $c > 0$ such that for sufficiently large n, $h(n) \leq n^c$. We'll sometimes also also write $h(n) = \text{poly}(n)$ in this case.

For more examples and explanations, see any undergraduate algorithms text such as [DPV06, KT06, CLRS01] or Section 7.1 in Sipser's book [Sip96].

EXERCISES

0.1. For each of the following pairs of functions f, g determine whether $f = o(g)$, $g = o(f)$ or $f = \Theta(g)$. If $f = o(g)$ then find the first number n such that $f(n) < g(n)$:

(a) $f(n) = n^2$, $g(n) = 2n^2 + 100\sqrt{n}$.
(b) $f(n) = n^{100}$, $g(n) = 2^{n/100}$.
(c) $f(n) = n^{100}$, $g(n) = 2^{n^{1/100}}$.
(d) $f(n) = \sqrt{n}$, $g(n) = 2^{\sqrt{\log n}}$.
(e) $f(n) = n^{100}$, $g(n) = 2^{(\log n)^2}$.
(f) $f(n) = 1000n$, $g(n) = n \log n$.

0.2. For each of the following recursively defined functions f, find a closed (nonrecursive) expression for a function g such that $f(n) = \Theta(g(n))$, and prove that this is the case. (*Note*: Below we only supply the recursive rule, you can assume that $f(1) = f(2) = \cdots = f(10) = 1$ and the recursive rule is applied for $n > 10$; in any case, regardless of how the base case is defined it won't make any difference to the answer. Can you see why?)

(a) $f(n) = f(n-1) + 10$.
(b) $f(n) = f(n-1) + n$.
(c) $f(n) = 2f(n-1)$.

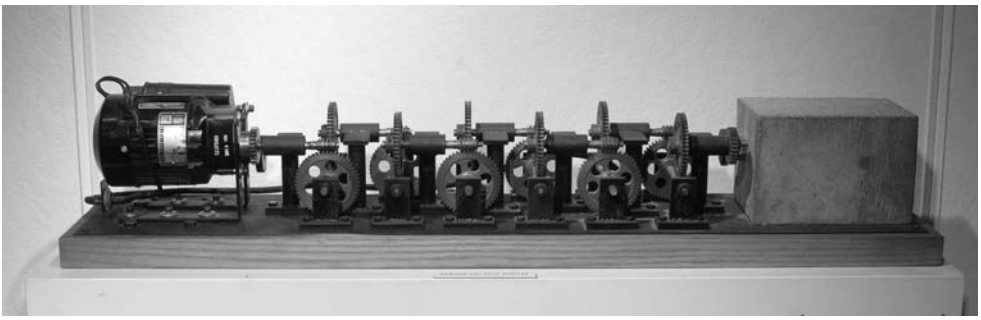

Figure 0.1. *Machine with Concrete* by Arthur Ganson. Reproduced with permission of the artist.

(d) $f(n) = f(n/2) + 10$.
(e) $f(n) = f(n/2) + n$.
(f) $f(n) = 2f(n/2) + n$.
(g) $f(n) = 3f(n/2)$.
(h) $f(n) = 2f(n/2) + O(n^2)$.

H531

0.3. The MIT museum contains a kinetic sculpture by Arthur Ganson called *Machine with Concrete* (see Figure 0.1). It consists of 13 gears connected to one another in a series such that each gear moves 50 times slower than the previous one. The fastest gear is constantly rotated by an engine at a rate of 212 rotations per minute. The slowest gear is fixed to a block of concrete and so apparently cannot move at all. Explain why this machine does not break apart.

PART ONE BASIC COMPLEXITY CLASSES

CHAPTER 1 The computational model—and why it doesn't matter

The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer. The human computer is supposed to be following fixed rules; he has no authority to deviate from them in any detail. We may suppose that these rules are supplied in a book, which is altered whenever he is put on to a new job. He has also an unlimited supply of paper on which he does his calculations.

– Alan Turing, 1950

[Turing] has for the first time succeeded in giving an absolute definition of an interesting epistemological notion, i.e., one not depending on the formalism chosen.

– Kurt Gödel, 1946

The problem of mathematically modeling computation may at first seem insurmountable: Throughout history people have been solving computational tasks using a wide variety of methods, ranging from intuition and "eureka" moments to mechanical devices such as abacus or sliderules to modern computers. Besides that, other organisms and systems in nature are also faced with and solve computational tasks every day using a bewildering array of mechanisms. How can you find a simple mathematical model that captures all of these ways to compute? The problem is even further exacerbated since in this book we are interested in issues of *computational efficiency*. Here, at first glance, it seems that we have to be very careful about our choice of a computational model, since even a kid knows that whether or not a new video game program is "efficiently computable" depends upon his computer's hardware.

Surprisingly enough, it turns out there there is a simple mathematical model that suffices for studying many questions about computation and its efficiency—the *Turing machine*. It suffices to restrict attention to this single model since it seems able to *simulate* all physically realizable computational methods with little loss of efficiency. Thus the set of "efficiently solvable" computational tasks is at least as large for the Turing machine as for any other method of computation. (One possible exception is the quantum computer model described in Chapter 10, but we do not currently know if it is physically realizable.)

In this chapter, we formally define Turing machines and survey some of their basic properties. Section 1.1 sketches the model and its basic properties. That section also gives an overview of the results of Sections 1.2 through 1.5 for the casual readers who

wish to skip the somewhat messy details of the model and go on to complexity theory, which begins with Section 1.6.

Since complexity theory is concerned with *computational efficiency*, Section 1.6 contains one of the most important definitions in this book: the definition of complexity class **P**, which aims to capture mathematically the set of all decision problems that can be efficiently solved. Section 1.6 also contains some discussion on whether or not the class **P** truly captures the informal notion of "efficient computation." The section also points out how throughout the book the definition of the Turing machine and the class **P** will be a starting point for definitions of many other models, including nondeterministic, probabilistic, and quantum Turing machines, Boolean circuits, parallel computers, decision trees, and communication games. Some of these models are introduced to study arguably realizable modes of physical computation, while others are mainly used to gain insights on Turing machine computations.

1.1 MODELING COMPUTATION: WHAT YOU REALLY NEED TO KNOW

Some tedious notation is unavoidable if one talks formally about Turing machines. We provide an intuitive overview of this material for casual readers who can then skip ahead to complexity questions, which begin with Section 1.6. Such a reader can always return to the skipped sections on the rare occasions in the rest of the book when we actually use details of the Turing machine model.

For thousands of years, the term "computation" was understood to mean application of mechanical rules to manipulate numbers, where the person/machine doing the manipulation is allowed a *scratch pad* on which to write the intermediate results. The Turing machine is a concrete embodiment of this intuitive notion. Section 1.2.1 shows that it can be also viewed as the equivalent of any modern programming language—albeit one with no built-in prohibition on its memory size.[1]

Here we describe this model informally along the lines of Turing's quote at the start of the chapter. Let f be a function that takes a string of bits (i.e., a member of the set $\{0,1\}^*$) and outputs either 0 or 1. An *algorithm* for computing f is a set of mechanical rules, such that by following them we can compute $f(x)$ given any input $x \in \{0,1\}^*$. The set of rules being followed is fixed (i.e., the same rules must work for all possible inputs) though each rule in this set may be applied arbitrarily many times. Each rule involves one or more of the following "elementary" operations:

1. Read a bit of the input.
2. Read a bit (or possibly a symbol from a slightly larger alphabet, say a digit in the set $\{0,\ldots,9\}$) from the scratch pad or working space we allow the algorithm to use.

Based on the values read,

1. Write a bit/symbol to the scratch pad.
2. Either stop and output 0 or 1, or choose a new rule from the set that will be applied next.

[1] Though the assumption of a potentially infinite memory may seem unrealistic at first, in the complexity setting it is of no consequence since we will restrict our study to machines that use at most a finite number of computational steps and memory cells any given input (the number allowed will depend upon the input size).

Finally, the *running time* is the number of these basic operations performed. We measure it in asymptotic terms, so we say a machine runs in time $T(n)$ if it performs at most $T(n)$ basic operations time on inputs of length n.

The following are simple facts about this model.

1. The model is robust to almost any tweak in the definition such as changing the alphabet from $\{0,1,\ldots,9\}$ to $\{0,1\}$, or allowing multiple scratchpads, and so on. The most basic version of the model can *simulate* the most complicated version with at most polynomial (actually quadratic) slowdown. Thus t steps on the complicated model can be simulated in $O(t^c)$ steps on the weaker model where c is a constant depending only on the two models. See Section 1.3.
2. An algorithm (i.e., a machine) can be represented as a bit string once we decide on some canonical encoding. Thus an algorithm/machine can be viewed as a possible *input* to another algorithm—this makes the boundary between *input*, *software*, and *hardware* very fluid. (As an aside, we note that this fluidity is the basis of a lot of computer technology.) We denote by M_α the machine whose representation as a bit string is α.
3. There is a *universal* Turing machine $\mathcal{U}$ that can *simulate* any other Turing machine given its bit representation. Given a pair of bit strings (x,α) as input, this machine simulates the behavior of M_α on input x. This simulation is very efficient: If the running time of M_α was $T(|x|)$, then the running time of $\mathcal{U}$ is $O(T(|x|)\log T(|x|))$. See Section 1.4.
4. The previous two facts can be used to easily prove the existence of functions that are not computable by any Turing machine; see Section 1.5. Uncomputability has an intimate connection to Gödel's famous Incompleteness Theorem; see Section 1.5.2.

1.2 THE TURING MACHINE

The *k-tape Turing machine* (TM) concretely realizes the above informal notion in the following way (see Figure 1.1).

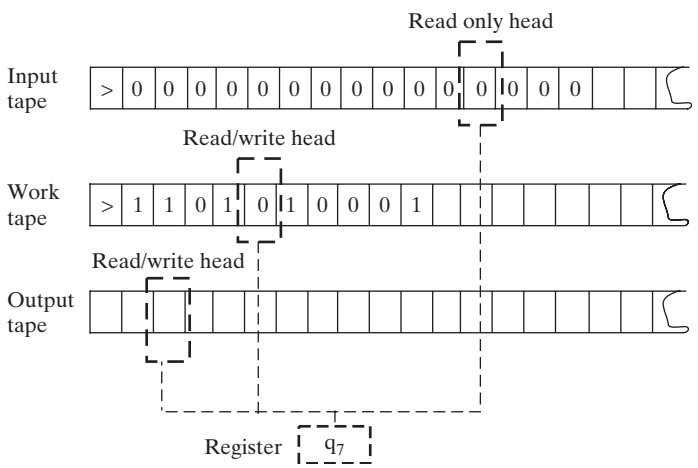

Figure 1.1. A snapshot of the execution of a three-tape Turing machine M with an input tape, a work tape, and an output tape.

Scratch pad

The scratch pad consists of k tapes. A *tape* is an infinite one-directional line of cells, each of which can hold a symbol from a finite set Γ called the *alphabet* of the machine. Each tape is equipped with a *tape head* that can potentially read or write symbols to the tape one cell at a time. The machine's computation is divided into discrete time steps, and the head can move left or right one cell in each step.

The first tape of the machine is designated as the *input* tape. The machine's head can only read symbols from that tape, not write them—a so-called read-only head. The $k-1$ read-write tapes are called *work tapes*, and the last one of them is designated as the *output tape* of the machine, on which it writes its final answer before halting its computation.

There also are variants of Turing machines with *random access memory*,[2] but it turns out that their computational powers are equivalent to standard Turing machines (see Exercise 1.9).

Finite set of operations/rules

The machine has a finite set of *states*, denoted Q. The machine contains a "register" that can hold a single element of Q; this is the "state" of the machine at that instant. This state determines its action at the next computational step, which consists of the following: (1) read the symbols in the cells directly under the k heads; (2) for the $k-1$ read-write tapes, replace each symbol with a new symbol (it has the option of not changing the tape by writing down the old symbol again); (3) change its register to contain another state from the finite set Q (it has the option not to change its state by choosing the old state again); and (4) move each head one cell to the left or to the right (or stay in place).

One can think of the Turing machine as a simplified modern computer, with the machine's tape corresponding to a computer's memory and the transition function and register corresponding to the computer's central processing unit (CPU). However, it's best to think of Turing machines as simply a formal way to describe algorithms. Even though algorithms are often best described by plain English text, it is sometimes useful to express them by such a formalism in order to argue about them mathematically. (Similarly, one needs to express an algorithm in a programming language in order to execute it on a computer.)

Formal definition. Formally, a TM M is described by a tuple (Γ, Q, δ) containing:

- A finite set Γ of the symbols that M's tapes can contain. We assume that Γ contains a designated "blank" symbol, denoted $\square$; a designated "start" symbol, denoted $\triangleright$; and the numbers 0 and 1. We call Γ the *alphabet* of M.
- A finite set Q of possible states M's register can be in. We assume that Q contains a designated start state, denoted $q_{\texttt{start}}$, and a designated halting state, denoted $q_{\texttt{halt}}$.

[2] *Random access* denotes the ability to access the ith symbol of the memory within a single step, without having to move a head all the way to the ith location. The name "random access" is somewhat unfortunate since this concept involves no notion of randomness—perhaps "indexed access" would have been better. However, "random access" is widely used, and so we follow this convention this book.

IF			THEN			
Input symbol read	Work/ output tape symbol read	Current state	Move input head	New work/ output tape symbol	Move work/ output tape	New state
⋮	⋮	⋮	⋮	⋮	⋮	⋮
a	b	q	**R**ight →	**b**'	**L**eft ←	**q**'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 1.2. The transition function of a two-tape TM (i.e., a TM with one input tape and one work/output tape).

- A function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$, where $k \geq 2$, describing the rules M use in performing each step. This function is called the *transition function* of M (see Figure 1.2.)

If the machine is in state $q \in Q$ and $(\sigma_1, \sigma_2, \ldots, \sigma_k)$ are the symbols currently being read in the k tapes, and $\delta(q, (\sigma_1, \ldots, \sigma_k)) = (q', (\sigma'_2, \ldots, \sigma'_k), z)$ where $z \in \{\mathtt{L}, \mathtt{S}, \mathtt{R}\}^k$, then at the next step the σ symbols in the last $k-1$ tapes will be replaced by the σ' symbols, the machine will be in state q', and the k heads will move Left, Right, or Stay in place, as given by z. (If the machine tries to move left from the leftmost position of a tape then it will stay in place.)

All tapes except for the input are initialized in their first location to the *start* symbol $\triangleright$ and in all other locations to the *blank* symbol $\square$. The input tape contains initially the start symbol $\triangleright$, a finite nonblank string x ("the input"), and the the blank symbol $\square$ on the rest of its cells. All heads start at the left ends of the tapes and the machine is in the special starting state $q_{\mathtt{start}}$. This is called the *start configuration* of M on input x. Each step of the computation is performed by applying the function δ as described previously. The special halting state $q_{\mathtt{halt}}$ has the property that once the machine is in $q_{\mathtt{halt}}$, the transition function δ does not allow it to further modify the tape or change states. Clearly, if the machine enters $q_{\mathtt{halt}}$, then it has *halted*. In complexity theory, we are typically only interested in machines that halt for every input in a finite number of steps.

EXAMPLE 1.1

Let PAL be the Boolean function defined as follows: for every $x \in \{0,1\}^*$, $\mathtt{PAL}(x)$ is equal to 1 if x is a *palindrome* and equal to 0 otherwise. That is, $\mathtt{PAL}(x) = 1$ if and only if x reads the same from left to right as from right to left (i.e., $x_1 x_2 \ldots x_n = x_n x_{n-1} \ldots x_1$). We now show a TM M that computes PAL within less than $3n$ steps.

Our TM M will use three tapes (input, work, and output) and the alphabet $\{\triangleright, \square, 0, 1\}$. It operates as follows:

1. Copy the input to the read-write work tape.
2. Move the input-tape head to the beginning of the input.
3. Move the input-tape head to the right while moving the work-tape head to the left. If at any moment the machine observes two different values, it halts and output 0.
4. Halt and output 1.

We now describe the machine more formally: The TM M uses five states denoted by $\{q_{\texttt{start}}, q_{\texttt{copy}}, q_{\texttt{left}}, q_{\texttt{test}}, q_{\texttt{halt}}\}$. Its transition function is defined as follows:

1. On state $q_{\texttt{start}}$: Move the input-tape head to the right, and move the work-tape head to the right while writing the start symbol $\triangleright$; change the state to $q_{\texttt{copy}}$. (Unless we mention this explicitly, the function does not change the output tape's contents or head position.)
2. On state $q_{\texttt{copy}}$:
   - If the symbol read from the input tape is not the blank symbol $\square$, then move both the input-tape and work-tape heads to the right, writing the symbol from the input tape on the work tape; stay in the state $q_{\texttt{copy}}$.
   - If the symbol read from the input tape is the blank symbol $\square$, then move the input-tape head to the left, while keeping the work-tape head in the same place (and not writing anything); change the state to $q_{\texttt{left}}$.
3. On state $q_{\texttt{left}}$:
   - If the symbol read from the input tape is not the start symbol $\triangleright$, then move the input head to the left, keeping the work-tape head in the same place (and not writing anything); stay in the state $q_{\texttt{left}}$.
   - If the symbol read from the input tape is the start symbol $\triangleright$, then move the input tape to the right and the work-tape head to the left (not writing anything); change to the state $q_{\texttt{test}}$.
4. On state $q_{\texttt{test}}$:
   - If the symbol read from the input tape is the blank symbol $\square$ and the symbol read from the work-tape is the start symbol $\triangleright$, then write 1 on the output tape and change state to $q_{\texttt{halt}}$.
   - Otherwise, if the symbols read from the input tape and the work tape are not the same, then write 0 on the output tape and change state to $q_{\texttt{halt}}$.
   - Otherwise, if the symbols read from the input tape and the work tape are the same, then move the input-tape head to the right and the work-tape head to the left; stay in the state $q_{\texttt{test}}$.

Clearly, fully specifying a Turing machine is tedious and not always informative. Even though it is useful to work out one or two examples for yourself (see Exercise 1.1), in the rest of this book we avoid such overly detailed descriptions and specify TMs in a high-level fashion. For readers with some programming experience, Example 1.2 should convince them that they know (in principle at least) how to design a Turing machine for any computational task for which they know how to write computer programs.

1.2.1 The expressive power of Turing machines

At first sight, it may be unclear that Turing machines do indeed encapsulate our intuitive notion of computation. It may be useful to work through some simple examples, such

as expressing the standard algorithms for addition and multiplication in terms of Turing machines computing the corresponding functions (see Exercise 1.1). Having done that, you may be ready for the next example; it outlines how you can translate a program in your favorite programming language into a Turing machine. (The reverse direction also holds: Most programming languages can simulate a Turing machine.)

EXAMPLE 1.2 (Simulating a general programming language using Turing machines)

(This example assumes some background in computing.)

We give a hand-wavy proof that for any program written in any of the familiar programming languages such as C or Java, there is an equivalent Turing machine. First, recall that programs in these programming languages can be translated (the technical term is *compiled*) into an equivalent *machine language* program. This is a sequence of instructions, each of one of a few simple types, for example, (a) read from memory into one of a finite number of registers, (b) write a register's contents to memory, (c) add the contents of two registers and store the result in a third, and (d) perform (c) but with other operations such as multiplication instead of addition. All these operations can be easily simulated by a Turing machine. The memory and registers can be implemented using the machine's tapes, while the instructions can be encoded by the machine's transition function. For example, it's not hard to design TM's that add or multiply two numbers. To simulate the computer's memory, a two-tape TM can use one tape for the simulated memory and the other tape to do binary-to-unary conversion that allows it, for a number i in binary representation, to read or modify the ith location of its first tape. We leave details to Exercise 1.8.

Exercise 1.10 asks you to give a more rigorous proof of such a simulation for a simple tailor-made programming language.

1.3 EFFICIENCY AND RUNNING TIME

Now we formalize the notion of running time. As every nontrivial computational task requires at least reading the entire input, we count the number of basic steps *as a function of the input length*.

Definition 1.3 *(Computing a function and running time)*
Let $f : \{0,1\}^* \to \{0,1\}^*$ and let $T : \mathbb{N} \to \mathbb{N}$ be some functions, and let M be a Turing machine. We say that *M computes f* if for every $x \in \{0,1\}^*$, whenever M is initialized to the start configuration on input x, then it halts with $f(x)$ written on its output tape. We say *M computes f in $T(n)$-time*[4] if its computation on every input x requires at most $T(|x|)$ steps.

[3] Formally we should write "T-time" instead of "$T(n)$-time," but we follow the convention of writing $T(n)$ to emphasize that T is applied to the input length.

EXAMPLE 1.4

It is easily checked that the Turing machine for palindrome recognition in Example 1.1 runs in $3n$ time.

Time-constructible functions

A function $T : \mathbb{N} \to \mathbb{N}$ is *time constructible* if $T(n) \geq n$ and there is a TM M that computes the function $x \mapsto \llcorner T(|x|) \lrcorner$ in time $T(n)$. (As usual, $\llcorner T(|x|) \lrcorner$ denotes the binary representation of the number $T(|x|)$.) Examples for time-constructible functions are n, $n \log n$, n^2, 2^n. Almost all functions encountered in this book will be time constructible, and we will restrict our attention to time bounds of this form. (Allowing time bounds that are not time constructible can lead to anomalous results.) The restriction $T(n) \geq n$ is to allow the algorithm time to read its input.

1.3.1 Robustness of our definition

Most of the specific details of our definition of Turing machines are quite arbitrary. It is a simple exercise to see that most changes to the definition do not yield a substantially different model, since our model can simulate any of these new models. In context of computational complexity, however, we have to verify not only that one model can simulate another, but also that it can do so efficiently. Now we state a few results of this type, which ultimately lead to the conclusion that the exact model is unimportant if we are willing to ignore polynomial factors in the running time. Variations on the model studied include restricting the alphabet Γ to be $\{0, 1, \square, \triangleright\}$, restricting the machine to have a single work tape, or allowing the tapes to be infinite in both directions. All results in this section are proved sketchily—completing these sketches into full proofs is a very good way to gain intuition on Turing machines, see Exercises 1.2, 1.3, and 1.4.

Claim 1.5 *For every $f : \{0,1\}^* \to \{0,1\}$ and time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using alphabet Γ, then it is computable in time $4 \log |\Gamma| T(n)$ by a TM $\tilde{M}$ using the alphabet $\{0, 1, \square, \triangleright\}$.* ◇

Proof Sketch: Let M be a TM with alphabet Γ, k tapes, and state set Q that computes the function f in $T(n)$ time. We describe an equivalent TM $\tilde{M}$ computing f with alphabet $\{0, 1, \square, \triangleright\}$, k tapes and a set Q' of states. The idea behind the transformation is simple: One can encode any member of Γ using $\log |\Gamma|$ bits.[5] Thus, each of $\tilde{M}$'s work tapes will simply encode one of M's tapes: For every cell in M's tape we will have $\log |\Gamma|$ cells in the corresponding tape of $\tilde{M}$ (see Figure 1.3).

To simulate one step of M, the machine $\tilde{M}$ will (1) use $\log |\Gamma|$ steps to read from each tape the $\log |\Gamma|$ bits encoding a symbol of Γ, (2) use its state register to store the symbols read, (3) use M's transition function to compute the symbols M writes and M's new state given this information, (4) store this information in its state register, and (5) use $\log |\Gamma|$ steps to write the encodings of these symbols on its tapes.

[5] Recall our conventions that log is taken to base 2, and noninteger numbers are rounded up when necessary.

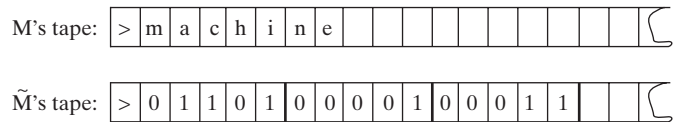

Figure 1.3. We can simulate a machine M using the alphabet $\{\triangleright, \square, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ by a machine M' using $\{\triangleright, \square, 0, 1\}$ via encoding every tape cell of M using five cells of M'.

One can verify that this can be carried out if $\tilde{M}$ has access to registers that can store M's state, k symbols in Γ, and a counter from 1 to $\log|\Gamma|$. Thus, there is such a machine $\tilde{M}$ utilizing no more than $c|Q||\Gamma|^{k+1}$ states for some absolute constant c. (In general, we can always simulate several registers using one register with a larger state space. For example, we can simulate three registers taking values in the sets A, B and C, respectively, with one register taking a value in the set $A \times B \times C$, which is of size $|A|\,|B|\,|C|$.)

It is not hard to see that for every input $x \in \{0,1\}^n$, if on input x the TM M outputs $f(x)$ within $T(n)$ steps, then $\tilde{M}$ will output the same value within less than $4\log|\Gamma|T(n)$ steps. ■

Now we consider the effect of restricting the machine to use a *single tape*—one read-write tape that serves as input, work, and output tape (this is the standard computational model in many undergraduate texts such as [Sip96]). We show that going from multiple tapes to a single tape can at most square the running time. This quadratic increase is inherent for some languages, including the palindrome recognition considered in Example 1.1; see the chapter notes.

Claim 1.6 *Define a single-tape Turing machine to be a TM that has only one read-write tape, that is used as input, work, and output tape. For every $f : \{0,1\}^* \to \{0,1\}$ and time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a TM M using k tapes, then it is computable in time $5kT(n)^2$ by a single-tape TM $\tilde{M}$.* ◇

Proof Sketch: Again the idea is simple: The TM $\tilde{M}$ encodes k tapes of M on a single tape by using locations $1, k+1, 2k+1, \ldots$ to encode the first tape, locations $2, k+2, 2k+2, \ldots$ to encode the second tape etc. (see Figure 1.4). For every symbol a in M's alphabet, $\tilde{M}$ will contain both the symbol a and the symbol $\hat{a}$. In the encoding of each tape, exactly one symbol will be of the "ˆ type," indicating that the corresponding head of M is positioned in that location (see Figure 1.4). $\tilde{M}$ will not touch the first $n+1$ locations of its tape (where the input is located) but rather start by taking $O(n^2)$ steps to copy the input bit by bit into the rest of the tape, while encoding it in the above way.

To simulate one step of M, the machine $\tilde{M}$ makes two sweeps of its work tape: First it sweeps the tape in the left-to-right direction and records to its register the k symbols that are marked by "ˆ". Then $\tilde{M}$ uses M's transition function to determine the new state, symbols, and head movements and sweeps the tape back in the right-to-left direction to update the encoding accordingly. Clearly, $\tilde{M}$ will have the same output as M. Also, since on n-length inputs M never reaches more than location $T(n)$ of any of its tapes, $\tilde{M}$

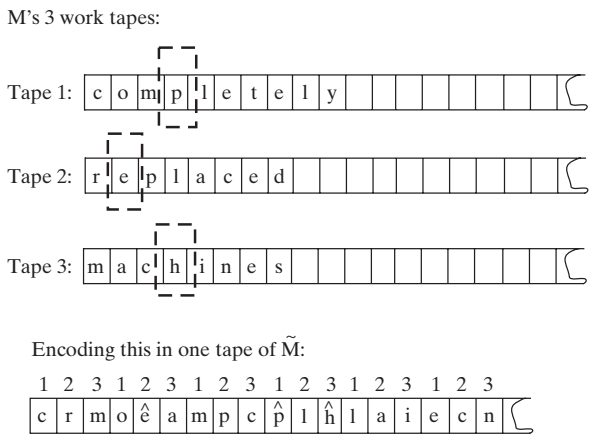

Figure 1.4. Simulating a machine M with three tapes using a machine $\tilde{M}$ with a single tape.

will never need to reach more than location $2n + kT(n) \leq (k+2)T(n)$ of its work tape, meaning that for each of the at most $T(n)$ steps of M, $\tilde{M}$ performs at most $5 \cdot k \cdot T(n)$ work (sweeping back and forth requires about $4 \cdot k \cdot T(n)$ steps, and some additional steps may be needed for updating head movement and book keeping). ■

Remark 1.7 *(Oblivious Turing machines)*
With a bit of care, one can ensure that the proof of Claim 1.6 yields a TM $\tilde{M}$ with the following property: Its head movements do not depend on the input but only depend on the input length. That is, every input $x \in \{0,1\}^*$ and $i \in \mathbb{N}$, the location of each of M's heads at the ith step of execution on input x is only a function of $|x|$ and i. A machine with this property is called *oblivious*, and the fact that every TM can be simulated by an oblivious TM will simplify some proofs later on (see Exercises 1.5 and 1.6 and the proof of Theorem 2.10).

Claim 1.8 *Define a bidirectional TM to be a TM whose tapes are infinite in both directions. For every $f : \{0,1\}^* \to \{0,1\}^*$ and time-constructible $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a bidirectional TM M, then it is computable in time $4T(n)$ by a standard (unidirectional) TM $\tilde{M}$.* ◇

Proof Sketch: The idea behind the proof is illustrated in Figure 1.5. If M uses alphabet Γ, then $\tilde{M}$ will use the alphabet Γ^2 (i.e., each symbol in $\tilde{M}$'s alphabet corresponds to a pair of symbols in M's alphabet). We encode a tape of M that is infinite in both direction using a standard (infinite in one direction) tape by "folding" it in an arbitrary location, with each location of $\tilde{M}$'s tape encoding two locations of M's tape. At first, $\tilde{M}$ will ignore the second symbol in the cell it reads and act according to M's transition function. However, if this transition function instructs $\tilde{M}$ to go "over the edge" of its tape, then instead it will start ignoring the first symbol in each cell and use only the second symbol. When it is in this mode, it will translate left movements into right movements and vice versa. If it needs to go over the edge again, then it will go back to reading the first symbol of each cell and translating movements normally. ■

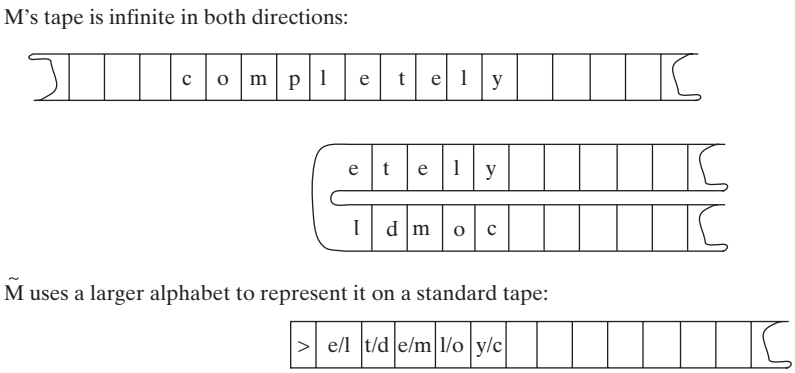

Figure 1.5. To simulate a machine M with alphabet Γ that has tapes infinite in both directions, we use a machine $\tilde{M}$ with alphabet Γ^2 whose tapes encode the "folded" version of M's tapes.

Other changes that do not have a very significant effect include having two or three dimensional tapes, allowing the machine *random access* to its tape, and making the output tape *write only* (see Exercises 1.7 and 1.9; the texts [Sip96, HMU01] contain more examples). In particular none of these modifications will change the class **P** of polynomial-time computable decision problems defined in Section 1.6.

1.4 MACHINES AS STRINGS AND THE UNIVERSAL TURING MACHINE

It is almost obvious that we can represent a Turing machine as a string: Just write the description of the TM on paper, and encode this description as a sequence of zeros and ones. This string can be given as input to another TM. This simple observation is actually profound since it blurs the distinction between *software*, *hardware*, and *data*. Historically speaking, it motivated the invention of the *general-purpose* electronic computer, which is a single machine that can be adapted to any arbitrary task by loading it with an appropriate program (software).

Because we will use this notion of representing TMs as strings quite extensively, it may be worthwhile to spell out our representation a bit more concretely. Since the behavior of a Turing machine is determined by its transition function, we will use the list of all inputs and outputs of this function (which can be easily encoded as a string in $\{0,1\}^*$) as the encoding of the Turing machine.[6] We will also find it convenient to assume that our representation scheme satisfies the following properties:

1. Every string in $\{0,1\}^*$ represents *some* Turing machine.
   This is easy to ensure by mapping strings that are not valid encodings into some canonical trivial TM, such as the TM that immediately halts and outputs zero on any input.

[6] Note that the size of the alphabet, the number of tapes, and the size of the state space can be deduced from the transition function's table. We can also reorder the table to ensure that the special states $q_{\texttt{start}}, q_{\texttt{halt}}$ are the first two states of the TM. Similarly, we may assume that the symbols $\triangleright, \square, 0, 1$ are the first four symbols of the machine's alphabet.

2. Every TM is represented by infinitely many strings.
   This can be ensured by specifying that the representation can end with an arbitrary number of 1s, that are ignored. This has a somewhat similar effect to the *comments* mechanism of many programming languages (e.g., the $/*\cdots*/$ construct in C, C++, and Java) that allows to add superfluous symbols to any program. Though this may seem like a nitpicky assumption, it will simplify some proofs.

We denote by $\llcorner M \lrcorner$ the TM M's representation as a binary string. If α is a string then M_α denotes the TM that α represents. As is our convention, we will also often use M to denote both the TM and its representation as a string. Exercise 1.11 asks you to fully specify a representation scheme for Turing machines with the above properties.

1.4.1 The universal Turing machine

Turing was the first to observe that general-purpose computers are possible, by showing a *universal* Turing machine that can *simulate* the execution of every other TM M given M's description as input. Of course, since we are so used to having a universal computer on our desktops or even in our pockets, today we take this notion for granted. But it is good to remember why it was once counterintuitive. The parameters of the universal TM are fixed—alphabet size, number of states, and number of tapes. The corresponding parameters for the machine being simulated could be much larger. The reason this is not a hurdle is, of course, the ability to use *encodings*. Even if the universal TM has a very simple alphabet, this suffices to represent the other machine's state and transition table on its tapes and then follow along in the computation step by step.

Now we state a computationally efficient version of Turing's construction due to Hennie and Stearns [HS66]. To give the essential idea we first prove a slightly relaxed variant where the term $T \log T$ below is replaced with T^2. But since the efficient version is needed a few times in the book, a full proof is also given at the end of the chapter (see Section 1.7).

Theorem 1.9 *(Efficient universal Turing machine)*
There exists a TM $\mathcal{U}$ such that for every $x, \alpha \in \{0,1\}^*$, $\mathcal{U}(x,\alpha) = M_\alpha(x)$, where M_α denotes the TM represented by α.
Moreover, if M_α halts on input x within T steps then $\mathcal{U}(x,\alpha)$ halts within $CT \log T$ steps, where C is a number independent of $|x|$ and depending only on M_α's alphabet size, number of tapes, and number of states.

A common exercise in programming courses is to write an *interpreter* for a particular programming language using the same language. (An interpreter takes a program P as input and outputs the result of executing the program P.) Theorem 1.9 can be considered a variant of this exercise.

Proof of relaxed version of Theorem 1.9: Our universal TM $\mathcal{U}$ is given an input x, α, where α represents some TM M, and needs to output $M(x)$. A crucial observation is that we may assume that M (1) has a single work tape (in addition to the input and output tape) and (2) uses the alphabet $\{\triangleright, \square, 0, 1\}$. The reason is that $\mathcal{U}$ can transform a representation of every TM M into a representation of an equivalent TM $\tilde{M}$ that satisfies

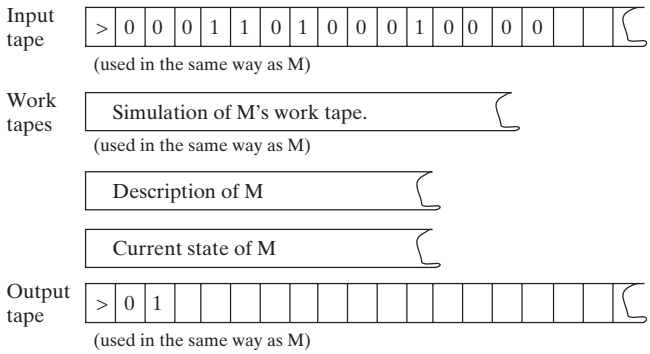

Figure 1.6. The universal TM $\mathcal{U}$ has in addition to the input and output tape, three work tapes. One work tape will have the same contents as the simulated machine M, another tape includes the description M (converted to an equivalent one-work-tape form), and another tape contains the current state of M.

these properties as shown in the proofs of Claims 1.5 and 1.6. Note that these transformations may introduce a quadratic slowdown (i.e., transform M from running in T time to running in $C'T^2$ time where C' depends on M's alphabet size and number of tapes).

The TM $\mathcal{U}$ uses the alphabet $\{\triangleright, \square, 0, 1\}$ and three work tapes in addition to its input and output tape (see Figure 1.6). $\mathcal{U}$ uses its input tape, output tape, and one of the work tapes in the same way M uses its three tapes. In addition, $\mathcal{U}$ will use its first extra work tape to store the table of values of M's transition function (after applying the transformations of Claims 1.5 and 1.6 as noted earlier), and its other extra work tape to store the current state of M. To simulate one computational step of M, $\mathcal{U}$ scans the table of M's transition function and the current state to find out the new state, symbols to be written and head movements, which it then executes. We see that each computational step of M is simulated using C steps of $\mathcal{U}$, where C is some number depending on the size of the transition function's table.

This high-level description can be turned into an exact specification of the TM $\mathcal{U}$, though we leave this to the reader. To work out the details, it may help to think first how to program these steps in your favorite programming language and then try to transform this into a description of a Turing machine. ■

Universal TM with time bound

It is sometimes useful to consider a variant of the universal TM $\mathcal{U}$ that gets a number T as an extra input (in addition to x and α), and outputs $M_\alpha(x)$ if and only if M_α halts on x within T steps (otherwise outputting some special failure symbol). By adding a time counter to $\mathcal{U}$, the proof of Theorem 1.9 can be easily modified to give such a universal TM. The time counter is used to keep track of the number of steps that the computation has taken so far.

1.5 UNCOMPUTABILITY: AN INTRODUCTION

It may seem "obvious" that every function can be computed, given sufficient time. However, this turns out to be false: There exist functions that cannot be computed

within any finite number of steps! This section gives a brief introduction to this fact and its ramifications. Though this material is not strictly necessary for the study of complexity, it forms the intellectual background for it.

The next theorem shows the existence of uncomputable functions. (In fact, it shows the existence of such functions whose range is {0, 1}, i.e. *languages*; such uncomputable functions with range {0, 1} are also known as *undecidable languages*.) The theorem's proof uses a technique called *diagonalization*, which is useful in complexity theory as well; see Chapter 3.

Theorem 1.10
There exists a function $\mathtt{UC} : \{0,1\}^* \to \{0,1\}$ *that is not computable by any TM.* ◇

Proof: The function UC is defined as follows: For every $\alpha \in \{0,1\}^*$, if $M_\alpha(\alpha) = 1$, then $\mathtt{UC}(\alpha) = 0$; otherwise (if $M_\alpha(\alpha)$ outputs a different value or enters an infinite loop), $\mathtt{UC}(\alpha) = 1$.

Suppose for the sake of contradiction that UC is computable and hence there exists a TM M such that $M(\alpha) = \mathtt{UC}(\alpha)$ for every $\alpha \in \{0,1\}^*$. Then, in particular, $M(\llcorner M \lrcorner) = \mathtt{UC}(\llcorner M \lrcorner)$. But this is impossible: By the definition of UC,

$$\mathtt{UC}(\llcorner M \lrcorner) = 1 \Leftrightarrow M(\llcorner M \lrcorner) \neq 1$$

■

Figure 1.7 demonstrates why this proof technique is called *diagonalization*.

1.5.1 The Halting problem (first encounter with reductions)

The reader may well ask why should we care whether or not the function UC described above is computable—who would want to compute such a contrived function anyway?

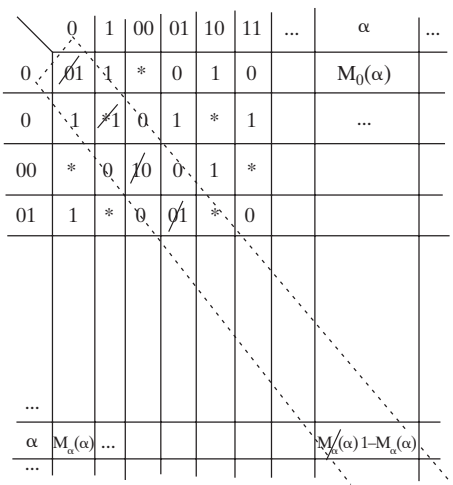

Figure 1.7. Suppose we order all strings in lexicographic order, and write in a table the value of $M_\alpha(x)$ for all strings α, x, where M_α denotes the TM represented by the string α and we use $\star$ to denote the case that $M_\alpha(x)$ is not a value in {0, 1} or that M_α does not halt on input x. Then, function UC is defined by "negating" the diagonal of this table. Since the rows of the table represent *all* TMs, we conclude that UC cannot be computed by any TM.

We now show a more natural uncomputable function. The function HALT takes as input a pair $\langle \alpha, x \rangle$ and outputs 1 if and only if the TM M_α represented by α halts on input x within a finite number of steps. This is definitely a function we want to compute: Given a computer program and an input, we'd certainly like to know if the program is going to enter an infinite loop on this input. If computers could compute HALT, the task of designing bug-free software and hardware would become much easier. Unfortunately, we now show that computers cannot do this, even if they are allowed to run an arbitrarily long time.

Theorem 1.11
HALT *is not computable by any TM.* ◇

Proof: Suppose, for the sake of contradiction, that there was a TM M_{HALT} computing HALT. We will use M_{HALT} to show a TM M_{UC} computing UC, contradicting Theorem 1.10.

The TM M_{UC} is simple: On input α, M_{UC} runs $M_{\mathsf{HALT}}(\alpha, \alpha)$. If the result is 0 (meaning that M_α does not halt on α), then M_{UC} outputs 1. Otherwise, M_{UC} uses the universal TM $\mathcal{U}$ to compute $b = M_\alpha(\alpha)$. If $b = 1$, then M_{UC} outputs 0; otherwise, it outputs 1.

Under the assumption that $M_{\mathsf{HALT}}(\alpha, \alpha)$ outputs $\mathsf{HALT}(\alpha, \alpha)$ within a finite number of steps, the TM $M_{\mathsf{UC}}(\alpha)$ will output $\mathsf{UC}(\alpha)$. ■

The proof technique employed to show Theorem 1.11 is called a *reduction*. We showed that computing UC is *reducible* to computing HALT—we showed that if there were a hypothetical algorithm for HALT then there would be one for UC. We will see many reductions in this book, often used (as is the case here) to show that a problem B is at least as hard as a problem A by showing an algorithm that could solve A given a procedure that solves B.

There are many other examples of interesting uncomputable (also known as *undecidable*) functions, see Exercise 1.12. There are even uncomputable functions whose formulation has seemingly nothing to do with Turing machines or algorithms. For example, the following problem cannot be solved in finite time by any TM: Given a set of polynomial equations with integer coefficients, find out whether these equations have an integer solution (i.e., whether there is an assignment of integers to the variables that satisfies the equations). This is known as the problem of solving Diophantine equations, and in 1900 Hilbert mentioned finding an algorithm for solving it (which he presumed to exist) as one of the top 23 open problems in mathematics. The chapter notes mention some good sources for more information on computability theory.

1.5.2 Gödel's Theorem

In the year 1900, David Hilbert, the preeminent mathematician of his time, proposed an ambitious agenda to base all of mathematics on solid axiomatic foundations, so that eventually all true statements would be rigorously proven. Mathematicians such as Russell, Whitehead, Zermelo, and Fraenkel, proposed axiomatic systems in the ensuing decades, but nobody was able to prove that their systems are simultaneously *complete* (i.e., prove all true mathematical statements) and *sound* (i.e., prove no false statements). In 1931, Kurt Gödel shocked the mathematical world by showing that this ongoing effort is doomed to fail—for every sound system $\mathcal{S}$ of axioms and rules

of inference, there exist true number theoretic statements that cannot be proven in $\mathcal{S}$. Below we sketch a proof of this result. Note that we this discussion does not address Gödel's more powerful result, which says that any sufficiently powerful axiomatization of mathematics cannot prove its own consistency. Proving that is also not too hard given the ideas below.

Gödel's work directly motivated the work of Turing and Church on computability. Our presentation reverses this order: We use uncomputability to sketch a proof of Gödel's result. The main observation is the following: In any sufficiently powerful axiomatic system, for any input $\langle \alpha, x \rangle$ we can write a mathematical statement $\phi_{\langle \alpha, x \rangle}$ that is true iff $\mathtt{HALT}(\langle \alpha, x \rangle) = 1$. (A sketch of this construction appears below.) Now if the system is complete, it must prove at least one of $\phi_{\langle \alpha, x \rangle}$ or $\neg\phi_{\langle \alpha, x \rangle}$, and if it is sound, it cannot prove both. So if the system is both complete and sound, the following algorithm for the Halting problem is guaranteed to terminate in finite time for all inputs. "*Given input $\langle \alpha, x \rangle$, start enumerating all strings of finite length, and check for each generated string whether it represents a proof in the axiomatic system for either $\phi_{\langle \alpha, x \rangle}$ or $\neg\phi_{\langle \alpha, x \rangle}$. If one waits long enough, a proof of one of the two statements will appear in the enumeration, at which point the correct answer* 1 *or* 0 *is revealed, which you then output.*" (Note that this procedure implicitly uses the simple fact that proofs in axiomatic systems can be easily verified by a Turing machine, since each step in the proof has to follow mechanically from previous steps by applying the axioms.)

Now we sketch the construction of the desired statement $\phi_{\langle \alpha, x \rangle}$. Assume the axiomatic system has the ability to express statements about the natural number using the operators plus ($+$) and times ($\times$), equality and comparison relations ($=, >, <$), and logical operators such as AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). The language also includes the quantifiers for-all ($\forall$) and exists ($\exists$) and the constant 1 (we can get any other constant c by adding 1 to itself c times). For example, the formal expression for "x divides y" will be $\mathtt{DIVIDES}(x, y) = \exists_k \ : \ y = x \times k$, and the expression for "y is prime" will be $\mathtt{PRIME}(y) = \ \forall_x (x{=}1) \vee (x{=}y) \vee \neg\mathtt{DIVIDES}(x, y)$ (where $\mathtt{DIVIDES}(x, y)$ is shorthand for the corresponding expression).

We can encode strings (and hence also Turing machines and their inputs and tapes) as numbers. Then one notes that a basic operation of the Turing machine only influences one (or a few, if the machine has mutiple tapes) of bits on its tape, which can be viewed as a simple arithmetic operation on the string/number representing the tape contents. With some work, one obtains an expression $\varphi_{\alpha,x}(t)$ that is true if and only if the TM M_α halts on input x within t steps. Hence, M_α halts on x if and only if $\exists_t \varphi_{\alpha,x}(t)$ is true, which is the desired mathematical statement. We leave the details as Exercise 1.13.

Note that this construction also implies, as first pointed out by Turing, that the set of true mathematical statements is undecidable, which showed that Hilbert's famous *Entscheidungsproblem* has no solution. (Hilbert had asked for a "mechanical procedure"—now interpreted as "algorithmic procedure"—for deciding truth of mathematical statements.)

1.6 THE CLASS P

A *complexity class* is a set of functions that can be computed within given resource bounds. We will now introduce our first complexity class. For reasons of technical

convenience, throughout most of this book, we will pay special attention to Boolean functions, namely those that have only one bit of output. These functions define *decision problems* or *languages*. We say that a machine *decides* a language $L \subseteq \{0,1\}^*$ if it computes the function $f_L : \{0,1\}^* \to \{0,1\}$, where $f_L(x) = 1 \Leftrightarrow x \in L$.

Definition 1.12 *(The class* **DTIME***)* Let $T : \mathbb{N} \to \mathbb{N}$ be some function. A language L is in $\mathbf{DTIME}(T(n))$ iff there is a Turing machine that runs in time $c \cdot T(n)$ for some constant $c > 0$ and decides L. ◇

The D in the notation **DTIME** refers to "deterministic." The Turing machine introduced in this chapter is more precisely called the *deterministic* Turing machine since for any given input x, the machine's computation can proceed in exactly one way. Later we will see other types of Turing machines, including nondeterministic and probabilistic TMs.

Now we try to make the notion of "efficient computation" precise. We equate this with *polynomial* running time, which means it is at most n^c for some constant $c > 0$. The following class captures this notion, where **P** stands for "polynomial."

Definition 1.13 *(The class* **P***)*
$\mathbf{P} = \cup_{c \geq 1} \mathbf{DTIME}(n^c)$.

Thus, we can phrase the question from the introduction as to whether the dinner party problem has an efficient algorithm as follows: *"Is* `INDSET` *in* **P***?"*, where `INDSET` is the language defined in Example 0.1.

EXAMPLE 1.14 *(Graph connectivity)*

In the *graph connectivity* problem, we are given a graph G and two vertices s, t in G. We have to decide if s is connected to t in G. This problem is in **P**. The algorithm that shows this uses *depth-first search*, a simple idea taught in undergraduate courses. The algorithm explored the graph edge-by-edge starting from s, marking visited edges. In subsequent edges, it also tries to explore all unvisited edges that are adjacent to previously visited edges. After at most $\binom{n}{2}$ steps, all edges are either visited or will never be visited.

See Exercise 1.14 for more examples of languages in **P**.

EXAMPLE 1.15

We give some examples to emphasize a couple of points about the definition of the class **P**. First, the class contains only decision problems. Thus we cannot say, for example, that "integer multiplication is in **P**." Instead, we may say that its decision version is in **P**, namely, the following language:

$$\{\langle x, i\rangle : \text{The } i\text{th bit of } xy \text{ is equal to } 1\}$$

Second, the running time is a function of the number of *bits* in the input. Consider the problem of solving a system of linear equations over the rational numbers. In other

words, given a pair $\langle A, \mathbf{b} \rangle$ where A is an $m \times n$ rational matrix and $\mathbf{b}$ is an m dimensional rational vector, find out if there exists an n-dimensional vector $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$. The standard Gaussian elimination algorithm solves this problem in $O(n^3)$ *arithmetic operations*. But on a Turing machine, each arithmetic operation has to be done in the gradeschool fashion, bit by laborious bit. Thus, to prove that this decision problem is in **P**, we have to verify that Gaussian elimination (or some other algorithm for the problem) runs on a Turing machine in time that is polynomial in the number of bits required to represent $a_1, a_2, \ldots, a_n$. That is, in the case of Gaussian elimination, we need to verify that all the intermediate numbers involved in the computation can be represented by polynomially many bits. Fortunately, this does turn out to be the case (for a related result, see Exercise 2.3).

1.6.1 Why the model may not matter

We defined the classes of "computable" languages and **P** using Turing machines. Would they be different if we had used a different computational model? Would these classes be different for some advanced alien civilization, which has discovered computation but with a different computational model than the Turing machine?

We already encountered variations on the Turing machine model, and saw that the weakest one can simulate the strongest one with quadratic slow down. Thus *polynomial* time is the same on all these variants, as is the set of computable problems.

In the few decades after Church and Turing's work, many other models of computation were discovered, some quite bizarre. It was easily shown that the Turing machine can simulate all of them with at most polynomial slowdown. Thus, the analog of **P** on these models is no larger than that for the Turing machine.

Most scientists believe the **Church-Turing (CT) thesis**, which states that every physically realizable computation device—whether it's based on silicon, DNA, neurons or some other alien technology—can be simulated by a Turing machine. This implies that the set of *computable* problems would be no larger on any other computational model that on the Turing machine. (The CT thesis is not a theorem, merely a belief about the nature of the world as we currently understand it.)

However, when it comes to *efficiently* computable problems, the situation is less clear. The **strong form of the CT thesis** says that every physically realizable computation model can be simulated by a TM *with polynomial overhead* (in other words, t steps on the model can be simulated in t^c steps on the TM, where c is a constant that depends upon the model). If true, it implies that the class **P** defined by the aliens will be the same as ours. However, this strong form is somewhat controversial, in particular because of models such as *quantum computers* (see Chapter 10), which do not appear to be efficiently simulatable on TMs. However, it is still unclear if quantum computers can be physically realized.

1.6.2 On the philosophical importance of P

The class **P** is felt to capture the notion of decision problems with "feasible" decision procedures. Of course, one may argue whether $\mathbf{DTIME}(n^{100})$ really represents "feasible" computation in the real world since n^{100} is prohibitively huge even for moderate

values of n. However, in practice, whenever we show that a problem is in **P**, we usually find an n^3 or n^5 time algorithm (with reasonable constants), and not an n^{100} time algorithm. (It has also happened a few times that the first polynomial-time algorithm for a problem had high complexity, say n^{20}, but soon somebody simplified it to say an n^5 time algorithm.)

Note that the class **P** is useful only in a certain context. Turing machines are a crude model if one is designing algorithms that must run in a fraction of a second on the latest PC (in which case one must carefully account for fine details about the hardware). However, if the question is whether any subexponential algorithms exist for, say, the language INDSET of Example 0.1, then even an n^{20} time algorithm would be a fantastic breakthrough.

P is also a natural class from the viewpoint of a programmer. Suppose a programmer is asked to invent the definition of an "efficient" computation. Presumably, she would agree that a computation that runs in linear or quadratic time is efficient. Next, since programmers often write programs that call other programs (or subroutines), she might find it natural to consider a program efficient if it performs only efficient computations and calls subroutines that are efficient. The resulting notion of "efficient computations" obtained turns out to be exactly the class **P** [Cob64].

1.6.3 Criticisms of P and some efforts to address them

Now we address some possible criticisms of the definition of **P** and some related complexity classes that address these.

Worst-case exact computation is too strict. The definition of **P** only considers algorithms that compute the function on *every* possible input. Critics point out that not all possible inputs arise in practice, and our algorithms only need to be efficient on the types of inputs that do arise. This criticism is partly answered using *average-case complexity* and by defining an analog of **P** in that context; see Chapter 18. We also note that quantifying "real-life" distributions is tricky.

Similarly, in context of computing functions such as the size of the largest independent set in the graph, users are often willing to settle for *approximate* solutions. Chapters 11 and 22 contain a rigorous treatment of the complexity of approximation.

Other physically realizable models. We already mentioned the strong form of the Church-Turing thesis, which posits that the class **P** is not larger for any physically realizable computational model. However, some subtleties need discusssion.

(a) *Issue of precision.* TM's compute with discrete symbols, whereas physical quantities may be real numbers in $\mathbb{R}$. Thus one can conceive of computational models based upon physics phenomena that may be able to operate over real numbers. Because of the precision issue, a TM can only approximately simulate such computations. It seems though that TMs do not suffer from an inherent handicap (though a few researchers disagree). After all, real-life devices suffer from noise, and physical quantities can only be measured up to finite precision. Thus physical processes could not involve arbitrary precision, and the simulating TM can therefore simulate them using finite precision. Even so, in Chapter 16 we also consider a modification of the TM model that allows

computations in $\mathbb{R}$ as a basic operation. The resulting complexity classes have fascinating connections with the standard classes.

(b) *Use of randomness.* The TM as defined is *deterministic.* If randomness exists in the world, one can conceive of computational models that use a source of random bits (i.e., coin tosses). Chapter 7 considers Turing machines that are allowed to also toss coins, and studies the complexity class **BPP**, which is the analog of **P** for those machines. However, we will see in Chapters 19 and 20 the intriguing possibility that randomized computation may be no more powerful than deterministic computation.

(c) *Use of quantum mechanics.* A more clever computational model might use some of the counterintuitive features of quantum mechanics. In Chapter 10, we define the complexity class **BQP**, which generalizes **P** in such a way. We will see problems in **BQP** that are currently not known to be in **P** (though there is no known proof that $\mathbf{BQP} \neq \mathbf{P}$). However, it is not yet clear whether the quantum model is truly physically realizable. Also quantum computers currently seem able to efficiently solve only very few problems that are not known to be in **P**. Hence some insights gained from studying **P** may still apply to quantum computers.

(d) *Use of other exotic physics, such as string theory.* So far it seems that many such physical theories yield the same class **BQP**, though much remains to be understood.

Decision problems are too limited. Some computational problems are not easily expressed as decision problems. Indeed, we will introduce several classes in the book to capture tasks such as computing non-Boolean functions, solving search problems, approximating optimization problems, interaction, and more. Yet the framework of decision problems turn out to be surprisingly expressive, and we will often use it in this book.

1.6.4 Edmonds's quote

We conclude this section with a quote from Edmonds [Edm65], who in his celebrated paper on a polynomial-time algorithm for the maximum matching problem, explained the meaning of such a result as follows:

> *For practical purposes computational details are vital. However, my purpose is only to show as attractively as I can that there is an efficient algorithm. According to the dictionary, "efficient" means "adequate in operation or performance." This is roughly the meaning I want — in the sense that it is conceivable for maximum matching to have no efficient algorithm.*
>
> *There is an obvious finite algorithm, but that algorithm increases in difficulty exponentially with the size of the graph. It is by no means obvious whether or not there exists an algorithm whose difficulty increases only algebraically with the size of the graph.*
>
> *When the measure of problem-size is reasonable and when the sizes assume values arbitrarily large, an asymptotic estimate of . . . the order of difficulty of an algorithm is theoretically important. It cannot be rigged by making the algorithm artificially difficult for smaller sizes.*
>
> *One can find many classes of problems, besides maximum matching and its generalizations, which have algorithms of exponential order but seemingly none better. . . . For practical purposes the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite.*

It would be unfortunate for any rigid criterion to inhibit the practical development of algorithms which are either not known or known not to conform nicely to the criterion. Many of the best algorithmic idea known today would suffer by such theoretical pedantry. . . . However, if only to motivate the search for good, practical algorithms, it is important to realize that it is mathematically sensible even to question their existence. For one thing the task can then be described in terms of concrete conjectures.

1.7 PROOF OF THEOREM 1.9: UNIVERSAL SIMULATION IN $O(T \log T)$-TIME

We now show how to prove Theorem 1.9 as stated. That is, we show a universal TM $\mathcal{U}$ such that given an input x and a description of a TM M that halts on x within T steps, $\mathcal{U}$ outputs $M(x)$ within $O(T \log T)$ time (where the constants hidden in the O notation may depend on the parameters of the TM M being simulated).

The general structure of $\mathcal{U}$ will be as in Section 1.4.1. $\mathcal{U}$ will use its input and output tape in the same way M does and will also have extra work tapes to store M's transition table and current state and to encode the contents of M's work tapes. The main obstacle we need to overcome is that we cannot use Claim 1.6 to reduce the number of M's work tapes to one, since Claim 1.6 introduces too much overhead in the simulation. Therefore, we will show a different way to encode all of M's work tapes in a single tape of $\mathcal{U}$, which we call the *main* work tape of $\mathcal{U}$.

Let k be the number of tapes that M uses (apart from its input and output tapes) and Γ its alphabet. Following the proof of Claim 1.5, we may assume that $\mathcal{U}$ uses the alphabet Γ^k (as this can be simulated with a overhead depending only on $k, |\Gamma|$). Thus we can encode in each cell of $\mathcal{U}$'s main work tape k symbols of Γ, each corresponding to a symbol from one of M's tapes. This means that we can think of $\mathcal{U}$'s main work tape not as a single tape but rather as k *parallel tapes*; that is, we can think of $\mathcal{U}$ as having k tapes with the property that in each step all their read-write heads go in unison either one location to the left, one location to the right, or stay in place. While we can easily encode the contents of M's k work tapes in $\mathcal{U}$'s k parallel tapes, we still have to deal with the fact that M's k read-write heads can each move independently to the left or right, whereas $\mathcal{U}$'s parallel tapes are forced to move together. Paraphrasing the famous saying, our strategy to handle this is: *"If the head cannot go to the tape locations then the locations will go to the head."*

That is, since we can not move $\mathcal{U}$'s read-write head in different directions at once, we simply move the parallel tapes "under" the head. To simulate a single step of M, we shift all the nonblank symbols in each of these parallel tapes until the head's position in these parallel tapes corresponds to the heads' positions of M's k tapes. For example, if $k = 3$ and in some particular step M's transition function specifies the movements L, R, R, then $\mathcal{U}$ will shift all the nonblank entries of its first parallel tape one cell to the right, and shift the nonblank entries of its second and third tapes one cell to the left (see Figure 1.8). $\mathcal{U}$ can easily perform such shifts using an additional "scratch" work tape.

The approach above is still not good enough to get $O(T \log T)$-time simulation. The reason is that there may be as many as T nonblank symbols in each parallel tape, and so each shift operation may cost $\mathcal{U}$ as much as T operations per each step of M, resulting in $\Theta(T^2)$-time simulation. We will deal with this problem by encoding the information

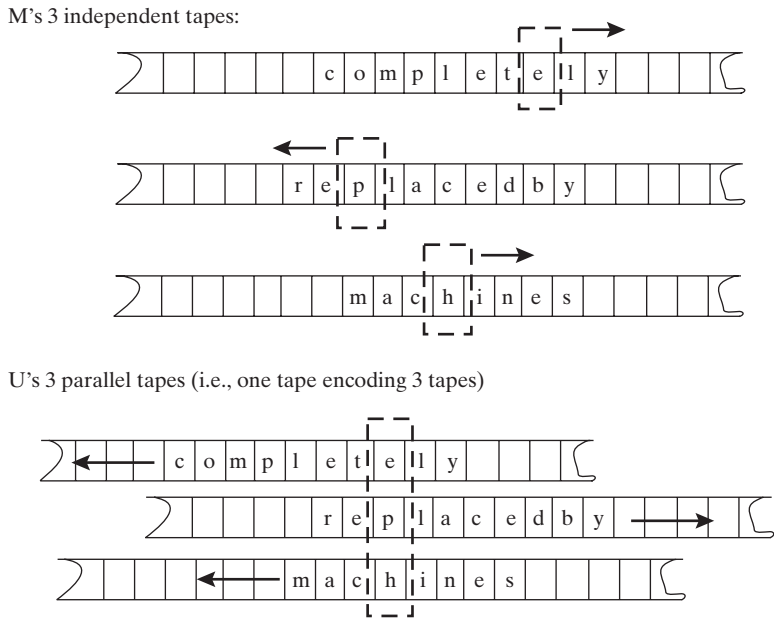

Figure 1.8. Packing k tapes of M into one tape of $\mathcal{U}$. We consider $\mathcal{U}$'s single work tape to be composed of k parallel tapes, whose heads move in unison, and hence we shift the contents of these tapes to simulate independent head movement.

on the tapes in a way that allows us to amortize the work of performing a shift. We will ensure that we do not need to move all the nonblank symbols of the tape in each shift operation. Specifically, we will encode the information in a way that allows half of the shift operations to be performed using $2c$ steps, for some constant c, a quarter of them using $4c$ steps, and more generally 2^{-i} fraction of the operations will take $2^i c$ steps, leading to simulation in roughly $cT \log T$ time (see below). (This kind of analysis is called *amortized analysis* and is widely used in algorithm design.)

Encoding M's tapes on $\mathcal{U}$'s tape

To allow more efficient shifts we encode the information using "buffer zones": Rather than having each of $\mathcal{U}$'s parallel tapes correspond exactly to a tape of M, we add a special kind of blank symbol $\boxtimes$ to the alphabet of $\mathcal{U}$'s parallel tapes with the semantics that this symbol is ignored in the simulation. For example, if the nonblank contents of M's tape are 010, then this can be encoded in the corresponding parallel tape of $\mathcal{U}$ not just by 010 but also by $0\boxtimes 01$ or $0\boxtimes\boxtimes 1\boxtimes 0$ and so on.

For convenience, we think of $\mathcal{U}$'s parallel tapes as infinite in both the left and right directions (this can be easily simulated with minimal overhead: see Claim 1.8). Thus, we index their locations by $0, \pm 1, \pm 2, \ldots$. Normally we keep $\mathcal{U}$'s head on location 0 of these parallel tapes. We will only move it temporarily to perform a shift when, following our general approach, we simulate a left head movement by shifting the tape to the right and vice versa. At the end of the shift, we return the head to location 0.

We split each of $\mathcal{U}$'s parallel tapes into *zones* that we denote by $R_0, L_0, R_1, L_1, \ldots$ (we'll only need to go up to $R_{\log T}, L_{\log T}$). The cell at location 0 is not

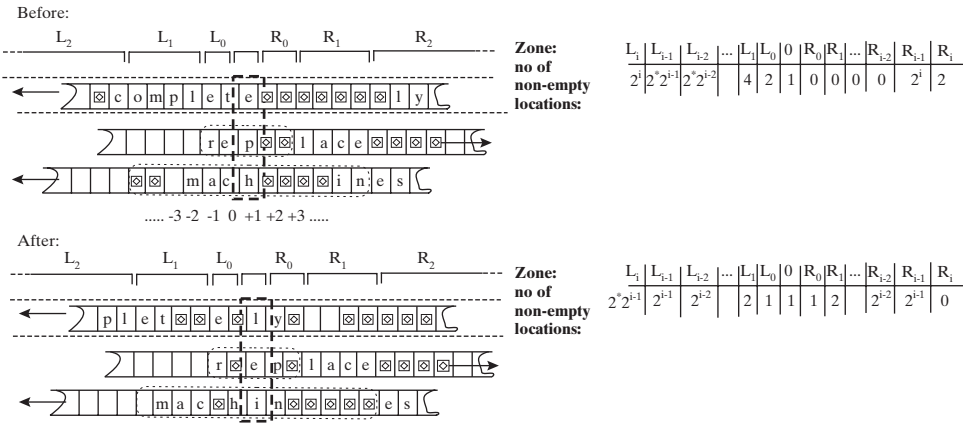

Figure 1.9. Performing a shift of the parallel tapes. The left shift of the first tape involves zones $R_0, L_0, R_1, L_1, R_2, L_2$, the right shift of the second tape involves only R_0, L_0, while the left shift of the third tape involves zones R_0, L_0, R_1, L_1. We maintain the invariant that each zone is either empty, half-full, or full and that the total number of nonempty cells in $R_i \cup L_i$ is $2 \cdot 2^i$. If before the left shift zones $L_0, \ldots, L_{i-1}$ were full and L_i was half-full (and so $R_0, \ldots, R_{i-1}$ were full and R_i half-full), then after the shift zones $R_0, L_0, \ldots, R_{i-1}, L_{i-1}$ will be half-full, L_i will be full and R_i will be empty.

at any zone. Zone R_0 contains the two cells immediately to the right of location C (i.e., locations $+1$ and $+2$), while Zone R_1 contains the four cells $+3, +4, +5, +6$. Generally, for every $i \geq 1$, Zone R_i contains the $2 \cdot 2^i$ cells that are to the right of Zone R_{i-1} (i.e., locations $[2^{i+1} - 1, \ldots, 2^{i+2} - 2]$). Similarly, Zone L_0 contains the two cells indexed by -1 and -2, and generally Zone L_i contains the cells $[-2^{i+2} + 2, \cdots, -2^{i+1} + 1]$. We shall always maintain the following invariants:

- Each of the zones is either *empty*, *full*, or *half-full* with non-⊠ symbols. That is, the number of symbols in zone R_i that are not ⊠ is either 0, 2^i, or $2 \cdot 2^i$ and the same holds for L_i. (We treat the ordinary □ symbol the same as any other symbol in Γ, and in particular a zone full of □'s is considered full.)
  We assume that initially all the zones are half-full. We can ensure this by filling half of each zone with ⊠ symbols in the first time we encounter it.
- The total number of non-⊠ symbols in $R_i \cup L_i$ is $2 \cdot 2^i$. That is, either R_i is empty and L_i is full, or R_i is full and L_i is empty, or they are both half-full.
- Location 0 always contains a non-⊠ symbol.

Performing a shift

The advantage in setting up these zones is that now when performing the shifts, we do not always have to move the entire tape, but we can restrict ourselves to only using some of the zones. We illustrate this by showing how $\mathcal{U}$ performs a left shift on the first of its parallel tapes (see also Figure 1.9):

1. $\mathcal{U}$ finds the smallest i_0 such that R_{i_0} is not empty. Note that this is also the smallest i_0 such that L_{i_0} is not full. We call this number i_0 the *index* of this particular shift.
2. $\mathcal{U}$ puts the leftmost non-⊠ symbol of R_{i_0} in position 0 and shifts the remaining leftmost $2^{i_0} - 1$ non-⊠ symbols from R_{i_0} into the zones $R_0, \ldots, R_{i_0-1}$ filling up exactly half the

symbols of each zone. Note that there is exactly room to perform this since all the zones $R_0, \ldots, R_{i_0-1}$ were empty and indeed $2^{i_0} - 1 = \sum_{j=0}^{i_0-1} 2^j$.

3. $\mathcal{U}$ performs the symmetric operation to the left of position 0. That is, for j starting from $i_0 - 1$ down to 0, $\mathcal{U}$ iteratively moves the $2 \cdot 2^j$ symbols from L_j to fill half the cells of L_{j+1}. Finally, $\mathcal{U}$ moves the symbol originally in position 0 (modified appropriately according to M's transition function) to L_0.
4. At the end of the shift, all of the zones $R_0, L_0, \ldots, R_{i_0-1}, L_{i_0-1}$ are half-full, R_{i_0} has 2^{i_0} fewer non-⊠ symbols, and L_i has 2^i additional non-⊠ symbols. Thus, our invariants are maintained.
5. The total cost of performing the shift is proportional to the total size of all the zones involved $R_0, L_0, \ldots, R_{i_0}, L_{i_0}$. That is, $O(\sum_{j=0}^{i_0} 2 \cdot 2^j) = O(2^{i_0})$ operations.

After performing a shift with index i the zones $L_0, R_0, \ldots, L_{i-1}, R_{i-1}$ are half-full, which means that it will take at least $2^i - 1$ left shifts before the zones $L_0, \ldots, L_{i-1}$ become empty or at least $2^i - 1$ right shifts before the zones $R_0, \ldots, R_{i-1}$ become empty. In any case, once we perform a shift with index i, the next $2^i - 1$ shifts of that particular parallel tape will all have index less than i. This means that for every one of the parallel tapes, at most a $1/2^i$ fraction of the total number of shifts have index i. Since we perform at most T shifts, and the highest possible index is $\log T$, the total work spent in shifting $\mathcal{U}$'s k parallel tapes in the course of simulating T steps of M is

$$O(k \cdot \sum_{i=1}^{\log T} \frac{T}{2^{i-1}} 2^i) = O(T \log T) \,. \blacksquare$$

What have we learned?

- There are many equivalent ways to mathematically model computational processes; we use the standard Turing machine formalization.
- Turing machines can be represented as strings. There is a *universal* TM that can simulate (with small overhead) any TM given its representation.
- There exist functions, such as the Halting problem, that cannot be computed by any TM regardless of its running time.
- The class **P** consists of all decision problems that are solvable by Turing machines in polynomial time. We say that problems in **P** are efficiently solvable.
- Low-level choices (number of tapes, alphabet size, etc..) in the definition of Turing machines are immaterial, as they will not change the definition of **P**.

CHAPTER NOTES AND HISTORY

Although certain algorithms have been studied for thousands of years, and some forms of computing devices were designed before the twentieth century (most notably Charles Babbage's difference and analytical engines in the mid 1800s), it seems fair to say that the foundations of modern computer science were only laid in the 1930s.

In 1931, Kurt Gödel shocked the mathematical world by showing that certain true statements about the natural numbers are *inherently unprovable*, thereby shattering an ambitious agenda set in 1900 by David Hilbert to base all of mathematics on solid axiomatic foundations. In 1936, Alonzo Church defined a model of computation called λ-calculus (which years later inspired the programming language LISP) and showed the existence of functions *inherently uncomputable* in this model [Chu36]. A few months later, Alan Turing independently introduced his Turing machines and showed functions inherently uncomputable by such machines [Tur36]. Turing also introduced the idea of the *universal* Turing machine that can be loaded with arbitrary programs. The two models turned out to be equivalent, but in the words of Church himself, Turing machines have "the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately." The anthology [Dav65] contains many seminal papers on computability. Part II of Sipser's book [Sip96] is a good gentle introduction to this theory, while the books [Rog87, HMU01, Koz97] go into a bit more depth. These books also cover *automata theory*, which is another area of the theory of computation not discussed in the current book. This book's Web site contains some additional links for information on both these topics.

During World War II, Turing designed mechanical code-breaking devices and played a key role in the effort to crack the German "Enigma" cipher, an achievement that had a decisive effect on the war's progress (see the biographies [Hod83, Lea05]).[7] After World War II, efforts to build electronic universal computers were undertaken in both sides of the Atlantic. A key figure in these efforts was John von Neumann, an extremely prolific scientist who was involved in everything from the Manhattan project to founding game theory in economics. To this day, essentially all digital computers follow the "von-Neumann architecture" he pioneered while working on the design of the EDVAC, one of the earliest digital computers [vN45].

As computers became more prevalent, the issue of efficiency in computation began to take center stage. Cobham [Cob64] defined the class **P** and suggested it may be a good formalization for efficient computation. A similar suggestion was made by Edmonds ([Edm65], see earlier quote) in the context of presenting a highly nontrivial polynomial-time algorithm for finding a maximum matching in general graphs. Hartmanis and Stearns [HS65] defined the class $\mathbf{DTIME}(T(n))$ for every function T and proved the slightly relaxed version of Theorem 1.9 we showed in this chapter (the version we stated and prove below was given by Hennie and Stearns [HS66]). They also coined the name "computational complexity" and proved an interesting "speed-up theorem": If a function f is computable by a TM M in time $T(n)$ then for every constant $c \geq 1$, f is computable by a TM $\tilde{M}$ (possibly with larger state size and alphabet size than M) in time $T(n)/c$. This speed-up theorem is another justification for ignoring constant factors in the definition of $\mathbf{DTIME}(T(n))$. Blum [Blu67] has given an axiomatic formalization of complexity theory that does not explicitly mention Turing machines.

We have omitted a discussion of some of the "bizarre conditions" that may occur when considering time bounds that are not time-constructible, especially "huge" time

[7] Unfortunately, Turing's wartime achievements were kept confidential during his lifetime, and so did not keep him from being forced by British courts to take hormones to "cure" his homosexuality, resulting in his suicide in 1954.

bounds (i.e., function $T(n)$ that are much larger than exponential in n). For example, there is a non-time-constructible function $T : \mathbb{N} \to \mathbb{N}$ such that every function computable in time $T(n)$ can also be computed in the much shorter time $\log T(n)$. However, we will not encounter non-time-constructible time bounds in this book.

The result that PAL requires $\Omega(n^2)$ steps to compute on TM's using a single read-write tape is from [Maa84], see also Exercise 13.3. We have stated that algorithms that take less than n steps are not very interesting as they do not even have time to read their input. This is true for the Turing machine model. However, if one allows *random access* to the input combined with *randomization* then many interesting computational tasks can actually be achieved in *sublinear* time. See [Fis04] for a survey of this line of research.

EXERCISES

1.1. Let f be the *addition* function that maps the representation of a pair of numbers x, y to the representation of the number $x + y$. Let g be the *multiplication* function that maps $\langle x, y \rangle$ to $\llcorner x \cdot y \lrcorner$. Prove that both f and g are computable by writing down a full description (including the states, alphabet, and transition function) of the corresponding Turing machines.

H531

1.2. Complete the proof of Claim 1.5 by writing down explicitly the description of the machine $\tilde{M}$.

1.3. Complete the proof of Claim 1.6.

1.4. Complete the proof of Claim 1.8.

1.5. Define a TM M to be *oblivious* if its head movements do not depend on the input but only on the input length. That is, M is oblivious if for every input $x \in \{0,1\}^*$ and $i \in \mathbb{N}$, the location of each of M's heads at the ith step of execution on input x is only a function of $|x|$ and i. Show that for every time-constructible $T\colon \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$, then there is an oblivious TM that decides L in time $O(T(n)^2)$. Furthermore, show that there is such a TM that uses only *two tapes*: one input tape and one work/output tape.

H531

1.6. Show that for every time-constructible $T\colon \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{DTIME}(T(n))$, then there is an oblivious TM that decides L in time $O(T(n) \log T(n))$.

H531

1.7. Define a *two-dimensional* Turing machine to be a TM where each of its tapes is an infinite grid (and the machine can move not only Left and Right but also Up and Down). Show that for every (time-constructible) $T\colon \mathbb{N} \to \mathbb{N}$ and every Boolean function f, if g can be computed in time $T(n)$ using a two-dimensional TM then $f \in \mathbf{DTIME}(T(n)^2)$.

1.8. Let LOOKUP denote the following function: on input a pair $\langle x, i \rangle$ (where x is a binary string and i is a natural number), LOOKUP outputs the ith bit of x or 0 if $|x| < i$. Prove that LOOKUP $\in \mathbf{P}$.

1.9. Define a *RAM Turing machine* to be a Turing machine that has *random access memory*. We formalize this as follows: The machine has an infinite array A that is initialized to all blanks. It accesses this array as follows. One of the machine's work tapes is designated as the *address tape*. Also the machine has two special alphabet symbols denoted by `R` and `W` and an additional state we denote by q_{access}. Whenever the machine enters q_{access}, if its address tape contains $\llcorner i \lrcorner$`R` (where $\llcorner i \lrcorner$ denotes the binary representation of i) then the value $A[i]$ is written in the cell next to the `R` symbol. If its tape contains $\llcorner i \lrcorner$`W`σ (where σ is some symbol in the machine's alphabet) then $A[i]$ is set to the value σ.

Show that if a Boolean function f is computable within time $T(n)$ (for some time-constructible T) by a RAM TM, then it is in $\mathbf{DTIME}(T(n)^2)$.

1.10. Consider the following simple programming language. It has a single infinite array `A` of elements in $\{0, 1, \square\}$ (initialized to $\square$) and a single integer variable `i`. A program in this language contains a sequence of lines of the following form:

$$\textit{label}\texttt{: If A[i] equals } \sigma \texttt{ then } \textit{cmds}$$

where $\sigma \in \{0, 1, \square\}$ and *cmds* is a list of one or more of the following commands: (1) `Set A[i] to` τ where $\tau \in \{0, 1, \square\}$, (2) `Goto` *label*, (3) `Increment i by one`, (4) `Decrement i by one`, and (5) `Output` b `and halt`, where $b \in \{0, 1\}$. A program is executed on an input $x \in \{0, 1\}^n$ by placing the ith bit of x in `A[i]` and then running the program following the obvious semantics.

Prove that for every functions $f : \{0, 1\}^* \to \{0, 1\}$ and (time-constructible) $T : \mathbb{N} \to \mathbb{N}$, if f is computable in time $T(n)$ by a program in this language, then $f \in \mathbf{DTIME}(T(n))$.

1.11. Give a full specification of a representation scheme of Turing machines as binary string strings. That is, show a procedure that transforms any TM M (e.g., the TM computing the function `PAL` described in Example 1.1) into a binary string $\llcorner M \lrcorner$. It should be possible to recover M from $\llcorner M \lrcorner$, or at least recover a functionally equivalent TM (i.e., a TM $\tilde{M}$ computing the same function as M with the same running time).

1.12. A *partial* function from $\{0, 1\}^*$ to $\{0, 1\}^*$ is a function that is not necessarily defined on all its inputs. We say that a TM M computes a partial function f if for every x on which f is defined, $M(x) = f(x)$ and for every x on which f is not defined M gets into an infinite loop when executed on input x. If $\mathcal{S}$ is a set of partial functions, we define $f_{\mathcal{S}}$ to be the Boolean function that on input α outputs 1 iff M_α computes a partial function in $\mathcal{S}$. *Rice's Theorem* says that for every nontrivial $\mathcal{S}$ (a set that is not the empty set nor the set of all partial functions computable by some Turing machine), the function $f_{\mathcal{S}}$ is not computable.

(a) Show that Rice's Theorem yields an alternative proof for Theorem 1.11 by showing that the function `HALT` is not computable.

(b) Prove Rice's Theorem.

H531

1.13. It is known that there is some constant C such that for every $i > C$ there is a prime larger than i^3 but smaller than $(i+1)^3$ [Hoh30, Ing37]. For every $i \in \mathbb{N}$, let p_i denote the smallest prime between $(i + C)^3$ and $(i + C + 1)^3$. We say that a number n

encodes a string $x \in \{0,1\}^*$, if for every $i \in \{1,\ldots,|x|\}$, p_i divides n if and only if $x_i = 1$.[8]

(a) Show (using the operators described in Section 1.5.2) a logical expression $\texttt{BIT}(n,i)$ that is true if and only if p_i divides n.

(b) Show a logical expression $\texttt{COMPARE}(n,m,i,j)$ that is true if and only if the strings encoded by the numbers n and m agree between the ith and jth position.

(c) A *configuration* of a TM M is the contents of all its input tapes, its head location, and the state of its register. That is, it contains all the information about M at a particular moment in its execution. Show that such a configuration can be represented by a binary string. (You may assume that M is a single-tape TM as in Claim 1.6.)

(d) For a TM M and input $x \in \{0,1\}^*$, show an expression $\texttt{INIT}_{M,x}(n)$ that is true if and only if n encodes the initial configuration of M on input x.

(e) For a TM M show an expression $\texttt{HALT}_M(n)$ that is true if and only if n encodes a configuration of M after which M will halt its execution.

(f) For a TM M, show an expression $\texttt{NEXT}(n,m)$ that is true if and only if n,m encode configurations x,y of M such that y is the configuration that is obtained from x by a single computational step of M.

(g) For a TM M, show an expression $\texttt{VALID}_M(m,t)$ that is true if and only m a tuple of t configurations $x_1,\ldots,x_t$ such that x_{i+1} is the configuration obtained from x_i in one computational step of M.

(h) For a TM M and input $x \in \{0,1\}^*$, show an expression $\texttt{HALT}_{M,x}(t)$ that is true if and only if M halts on input x within t steps.

(i) Let $\texttt{TRUE-EXP}$ denote the function that on input (a string representation of) a number-theoretic statement φ (composed in the preceding formalism), outputs 1 if φ is true, and 0 if φ is false. Prove that $\texttt{TRUE-EXP}$ is uncomputable.

1.14. Prove that the following languages/decision problems on graphs are in **P**. (You may pick either the adjacency matrix or adjacency list representation for graphs; it will not make a difference. Can you see why?)

(a) $\texttt{CONNECTED}$—The set of all connected graphs. That is, $G \in \texttt{CONNECTED}$ if every pair of vertices u,v in G are connected by a path.

(b) $\texttt{TRIANGLEFREE}$—The set of all graphs that do not contain a triangle (i.e., a triplet u,v,w of connected distinct vertices).

(c) $\texttt{BIPARTITE}$—The set of all bipartite graphs. That is, $G \in \texttt{BIPARTITE}$ if the vertices of G can be partitioned to two sets A,B such that all edges in G are from a vertex in A to a vertex in B (there is no edge between two members of A or two members of B).

(d) $\texttt{TREE}$—The set of all trees. A graph is a *tree* if it is connected and contains no cycles. Equivalently, a graph G is a tree if every two distinct vertices u,v in G are connected by exactly one simple path (a path is simple if it has no repeated vertices).

1.15. Recall that normally we assume that numbers are represented as string using the *binary* basis. That is, a number n is represented by the sequence $x_0,x_1,\ldots,x_{\log n}$

[8] Technically speaking under this definition a number can encode more than one string. This will not be an issue, though we can avoid it by first encoding the string x as a $2|x|$ bit string y using the map $0 \mapsto 00, 1 \mapsto 11$ and then adding the sequence 01 at the end of y.

such that $n = \sum_{i=0}^{n} x_i 2^i$. However, we could have used other encoding schemes. If $n \in \mathbb{N}$ and $b \geq 2$, then *the representation of n in base b*, denoted by $\llcorner n \lrcorner_b$ is obtained as follows: First, represent n as a sequence of digits in $\{0, \ldots, b-1\}$, and then replace each digit $d \in \{0, \ldots, d-1\}$ by its binary representation. The *unary* representation of n, denoted by $\llcorner n \lrcorner_{\text{unary}}$ is the string 1^n (i.e., a sequence of n ones).

(a) Show that choosing a different base of representation will make no difference to the class **P**. That is, show that for every subset S of the natural numbers, if we define $L_S^b = \{ \llcorner n \lrcorner_b : n \in S \}$, then for every $b \geq 2$, $L_S^b \in \mathbf{P}$ iff $L_S^2 \in \mathbf{P}$.

(b) Show that choosing the unary representation may make a difference by showing that the following language is in **P**:

$$\text{UNARYFACTORING} = \{ \langle \llcorner n \lrcorner_{\text{unary}}, \llcorner \ell \lrcorner_{\text{unary}}, \llcorner k \lrcorner_{\text{unary}} \rangle : \text{there is a prime } j \in (\ell, k) \text{ dividing } n \}$$

It is not known to be in **P** if we choose the binary representation (see Chapters 9 and 10). In Chapter 3 we will see that there is a problem that is *proven* to be in **P** when choosing the unary representation but not in **P** when using the binary representation.

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 Gödel in a letter to John von Neumann, 1956

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

– Jack Edmonds, 1966

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

– Richard Karp, 1972

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

This chapter studies the computational analog of the preceding phenomenon. In Section 2.1, we define a complexity class **NP** that aims to capture the set of problems whose solutions can be efficiently *verified*. By contrast, the class **P** of the previous chapter contains decision problems that can be efficiently *solved.* The famous **P** versus **NP** question asks whether or not the two classes are the same.

In Section 2.2, we introduce the important phenomenon of **NP**-*complete* problems, which are in a precise sense the "hardest problems" in **NP**. The number of real-life problems that are known to be **NP**-complete now runs into the thousands. Each of them has a polynomial algorithm if and only if $\mathbf{P} = \mathbf{NP}$. The study of **NP**-completeness

[*] In modern terminology, if SAT has a quadratic time algorithm

involves *reductions*, a basic notion used to relate the computational complexity of two different problems. This notion and its various siblings will often reappear in later chapters (e.g., in Chapters 7, 17, and 18). The framework of ideas introduced in this chapter motivates much of the rest of this book.

The implications of $\mathbf{P} = \mathbf{NP}$ are mind-boggling. As already mentioned, **NP** problems seem to capture some aspects of "creativity" in problem solving, and such creativity could become accessible to computers if $\mathbf{P} = \mathbf{NP}$. For instance, in this case computers would be able to quickly find proofs for every true mathematical statement for which a proof exists. We survey this "$\mathbf{P} = \mathbf{NP}$ Utopia" in Section 2.7.3. Resolving the **P** versus **NP** question is truly of great practical, scientific, and philosophical interest.

2.1 THE CLASS NP

Now we formalize the intuitive notion of *efficiently verifiable solutions* by defining a complexity class **NP**. In Chapter 1, we said that problems are "efficiently solvable" if they can be solved by a Turing machine in polynomial time. Thus, it is natural to say that solutions to the problem are "efficiently verifiable" if they can be verified in polynomial time. Since a Turing machine can only read one bit in a step, this means also that the presented solution has to be not too long—at most polynomial in the length of the input.

Definition 2.1 *(The class* **NP***)*
A language $L \subseteq \{0,1\}^*$ is in **NP** if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M (called the *verifier* for L) such that for every $x \in \{0,1\}^*$,

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

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

Some texts use the term *witness* instead of certificate. Clearly, $\mathbf{P} \subseteq \mathbf{NP}$ since the polynomial $p(|x|)$ is allowed to be 0 (in other words, u can be an empty string).

EXAMPLE 2.2 (INDSET ∈ NP)

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

encoded using $O(k \log n)$ bits, where n is the number of vertices in G. Thus, u is a string of at most $O(n \log n)$ bits, which is polynomial in the size of the representation of G.

EXAMPLE 2.3

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

Traveling salesperson: Given a set of n nodes, $\binom{n}{2}$ numbers $d_{i,j}$ denoting the distances between all pairs of nodes, and a number k, decide if there is a closed circuit (i.e., a "salesperson tour") that visits every node exactly once and has total length at most k. The certificate is the sequence of nodes in such a 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 such a 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 $a_1u_1 + a_2u_2 + \cdots + a_nu_n \leq b$ 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 (see Exercise 2.4).

0/1 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 zeroes and ones to $u_1, \ldots, u_n$ satisfying all the inequalities. The certificate is the assignment.

Graph isomorphism: Given two $n \times n$ adjacency matrices M_1, M_2, decide if M_1 and M_2 define the same graph, up to renaming of vertices. The certificate is the permutation $\pi : [n] \to [n]$ 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 prime factor p in the interval $[L, U]$. The certificate is the factor p.[1]

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

In the preceding list, the **connectivity**, **composite numbers**, and **linear programming** problems are known to be in **P**. For connectivity, this follows from the simple and well-known breadth-first search algorithm (see any algorithms text such as [KT06, CLRS01]). The composite numbers problem was only recently shown to be in **P** (see the beautiful algorithm of [AKS04]). For the linear programming problem, this is again highly nontrivial and follows from the Ellipsoid algorithm of Khachiyan [Kha79].

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

[1] There is a polynomial-time algorithm to check primality [AKS04]. We can also show that **Factoring** is in **NP** by using the primality certificate of Exercise 2.5.

2.1.1 Relation between NP and P

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

Claim 2.4 *Let* $\mathbf{EXP} = \bigcup_{c>1}$. *Then* $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$. ◇

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

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

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

2.1.2 Nondeterministic Turing machines

The class **NP** can also be defined using a variant of Turing machines called *nondeterministic* Turing machines (abbreviated NDTM). In fact, this was the original definition, and the reason for the name **NP**, which stands for *nondeterministic polynomial time*. The only difference between an NDTM and a standard TM (as defined in Section 1.2) is that an NDTM has *two* transition functions δ_0 and δ_1, and a special state denoted by $q_{\texttt{accept}}$. 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. For every input x, we say that $M(x) = 1$ if there *exists* some sequence of these choices (which we call the *nondeterministic choices* of M) that would make M reach $q_{\texttt{accept}}$ on input x. Otherwise—if *every* sequence of choices makes M halt without reaching $q_{\texttt{accept}}$—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 nondeterministic choices, M reaches either the halting state or $q_{\texttt{accept}}$ within $T(|x|)$ steps.

Definition 2.5 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 $c \cdot T(n)$-time NDTM M such that for every $x \in \{0,1\}^*$, $x \in L \Leftrightarrow M(x) = 1$. ◇

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

Theorem 2.6 $\mathbf{NP} = \cup_{c \in \mathbb{N}} \mathbf{NTIME}(n^c)$. ◇

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 $q_{\texttt{accept}}$ on input x. We can use this sequence as a *certificate* for x. This certificate has length $p(|x|)$ and can be verified in polynomial time by a *deterministic* machine, which simulates the action of N using these nondeterministic choices and verifies that it would have entered $q_{\texttt{accept}}$ 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 nondeterministic 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 $q_{\texttt{accept}}$. Clearly, N enters $q_{\texttt{accept}}$ 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 TMs, NDTMs can be easily represented as strings, and there exists a *universal* nondeterministic Turing machine (see Exercise 2.6). In fact, using nondeterminism, we can even make the simulation by a universal TM slightly more efficient.

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

2.2 REDUCIBILITY AND NP-COMPLETENESS

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

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

Definition 2.7 *(Reductions,* **NP***-hardness and* **NP***-completeness)* A language $L \subseteq \{0,1\}^*$ is *polynomial-time Karp reducible to a language* $L' \subseteq \{0,1\}^*$ (sometimes shortened to just "polynomial-time reducible"), denoted by $L \leq_p L'$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $x \in L$ if and only if $f(x) \in L'$.

We say that L' is **NP**-*hard* if $L \leq_p L'$ for every $L \in \mathbf{NP}$. We say that L' is **NP**-*complete* if L' is **NP**-hard and $L' \in \mathbf{NP}$.

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

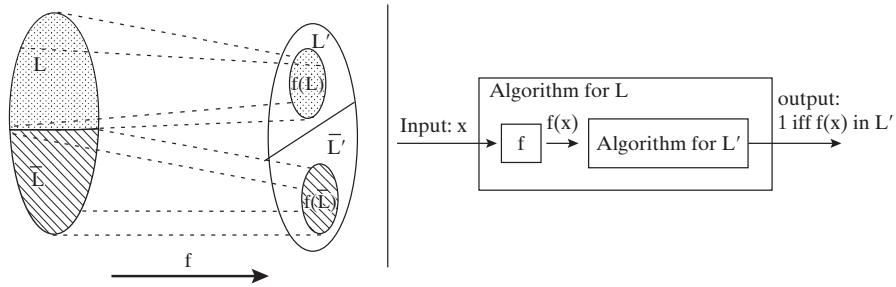

Figure 2.1. A Karp reduction from L to L' is a polynomial-time function f that maps strings in L to strings in L' and strings in $\overline{L} = \{0,1\}^* \setminus L$ to strings in $\overline{L'}$. It can be used to transform a polynomial-time TM M' that decides L' into a polynomial-time TM M for L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

The important (and easy to verify) property of polynomial-time reducibility is that if $L \leq_p L'$ and $L' \in \mathbf{P}$ then $L \in \mathbf{P}$—see Figure 2.1. This is why we say in this case that L' is *at least as hard* as L, as far as polynomial-time algorithms are concerned. Note that $\leq_p$ is a *relation* among languages, and part 1 of Theorem 2.8 shows that this relation is *transitive*. Later we will define other notions of reduction, and many will satisfy transitivity. Part 2 of the theorem suggests the reason for the term **NP**-hard—namely, an **NP**-hard languages is *at least as hard* as any other **NP** language. Part 3 similarly suggests the reason for the term **NP**-complete: to study the **P** versus **NP** question it suffices to study whether any **NP**-complete problem can be decided in polynomial time.

Theorem 2.8

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

Proof: The main observation underlying all three parts is that if p, q are two functions that grow at most as n^c and n^d, respectively, then composition $p(q(n))$ grows as at most n^{cd}, which is also polynomial. We now prove part 1 and leave the others as simple exercises.

If f_1 is a polynomial-time reduction from L to L' and f_2 is a reduction from L' to L'', then the mapping $x \mapsto f_2(f_1(x))$ is a polynomial-time reduction from L to L'' since $f_2(f_1(x))$ takes polynomial time to compute given x. Finally, $f_2(f_1(x)) \in L''$ iff $f_1(x) \in L'$, which holds iff $x \in L$. ■

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

Theorem 2.9 *The following language is* **NP***-complete:*

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

where M_α denotes the (deterministic) TM represented by the string α.[2] ◇

[2] Recall that 1^k denotes the string consisting of k bits, each of them 1. Often in complexity theory we include the string 1^k in the input to allow a polynomial TM to run in time polynomial in k.

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

$$\langle \llcorner M \lrcorner, x, 1^{p(|x|)}, 1^{q(m)} \rangle \in \text{TMSAT} \Leftrightarrow$$
$$\exists_{u \in \{0,1\}^{p(|x|)}} \text{ s.t. } M(x,u) \text{ outputs 1 within } q(m) \text{ steps} \Leftrightarrow x \in L$$

■

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

2.3 THE COOK-LEVIN THEOREM: COMPUTATION IS LOCAL

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

2.3.1 Boolean formulas, CNF, and SAT

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 ($\wedge$), OR ($\vee$), and NOT ($\neg$). For example, $(u_1 \wedge u_2) \vee (u_2 \wedge u_3) \vee (u_3 \wedge u_1)$ is a Boolean formula. If φ is a Boolean formula over variables $u_1, \ldots, u_n$, 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). A formula φ is *satisfiable* if there exists some assignment z such that $\varphi(z)$ is TRUE. Otherwise, we say that φ is *unsatisfiable*.

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

A Boolean formula over variables $u_1, \ldots, u_n$ 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: (here and elsewhere, $\bar{u}_i$ denotes $\neg u_i$)

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

More generally, a CNF formula has the form

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

where each v_{i_j} is either a variable u_k or its negation $\bar{u}_k$. The terms v_{i_j} are called the *literals* of the formula and the terms $(\vee_j v_{i_j})$ are called its *clauses*. A kCNF is a CNF formula in which all clauses contain at most k literals. We denote by SAT the language of all satisfiable CNF formulae and by 3SAT the language of all satisfiable 3CNF formulae.[3]

2.3.2 The Cook-Levin Theorem

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

Theorem 2.10 *(Cook-Levin Theorem* [Coo71, Lev73]*)*

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

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

Lemma 2.11 SAT *is* **NP***-hard.* ◇

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

2.3.3 Warmup: Expressiveness of Boolean formulas

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

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

EXAMPLE 2.12 *(Expressing equality of strings)*

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

$$(x_1 \vee \overline{y}_1) \wedge (\overline{x}_1 \vee y_1) \wedge \cdots \wedge (x_n \vee \overline{y}_n) \wedge (\overline{x}_n \vee y_n)$$

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

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

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

Claim 2.13 *(Universality of AND, OR, NOT) 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 Sketch: For every $v \in \{0,1\}^\ell$, it is not hard to see that there exists a clause $C_v(z_1, z_2, \ldots, z_\ell)$ in ℓ variables such that $C_v(v) = 0$ and $C_v(u) = 1$ for every $u \neq v$. For example, if $v = \langle 1,1,0,1 \rangle$, the corresponding clause is $\overline{z}_1 \vee \overline{z}_2 \vee z_3 \vee \overline{z}_4$.

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

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

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

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

2.3.4 Proof of Lemma 2.11

Let L be an **NP** language. By definition, there is polynomial time TM M such 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 polynomial-time transformation $x \to \varphi_x$ from strings to CNF formulae such that $x \in L$ iff φ_x is satisfiable. Equivalently,

$$\varphi_x \in \mathsf{SAT} \text{ iff } \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x \circ u) = 1 \tag{2.1}$$

(where $\circ$ denotes concatenation).[4]

[4] Because the length $p(|x|)$ of the second input u is easily computable, we can represent the pair $\langle x, u \rangle$ simply by $x \circ u$, without a need to use a "marker symbol" between x and u.

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

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

We can make these assumptions without loss of generality because for every $T(n)$-time TM M there exists a two-tape oblivious TM $\tilde{M}$ computing the same function in $O(T(n)^2)$ time (see Remark 1.7 and Exercise 1.5).[5] Thus in particular, if L is in **NP**, then there exists a two-tape *oblivious* polynomial-time TM M and a polynomial p such that

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \text{ s.t. } M(x \circ u) = 1 \tag{2.2}$$

Note that because M is oblivious, we can run it on the trivial input $(x, 0^{p(|x|)})$ to determine the precise head position of M during its computation on every other input of the same length. We will use this fact later on.

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 ith step (see Figure 2.2). Clearly the snapshot can be encoded as a binary string. Let c denote the length of this string, which is some constant depending upon $|Q|$ and $|\Gamma|$.

For every $y \in \{0,1\}^*$, the snapshot of M's execution on input y at the ith step depends on (a) its state in the $(i-1)$st step and (b) the contents of the current cells of its input and work tapes.

The insight at the heart of the proof concerns the following thought exercise. Suppose somebody were to claim the existence of some u satisfying $M(x \circ u) = 1$ and, as evidence, present you with the sequence of snapshots that arise from M's execution on $x \circ u$. How can you tell that the snapshots present a valid computation that was actually performed by M?

Clearly, it suffices to check that for each $i \leq T(n)$, the snapshot z_i is correct given the snapshots for the previous $i-1$ steps. However, since the TM can only read/modify one bit at a time, to check the correctness of z_i it suffices to look at only *two* of the previous snapshots. Specifically, to check z_i we need to only look at the following: $z_{i-1}, y_{\texttt{inputpos}(i)}, z_{\texttt{prev}(i)}$ (see Figure 2.3). Here y is shorthand for $x \circ u$;

[5] In fact, with some more effort, we even simulate a nonoblivious $T(n)$-time TM by an oblivious TM running in $O(T(n)\log T(n))$-time; see Exercise 1.6. This oblivious machine may have more than two tapes, but the following proof below easily generalizes to this case.

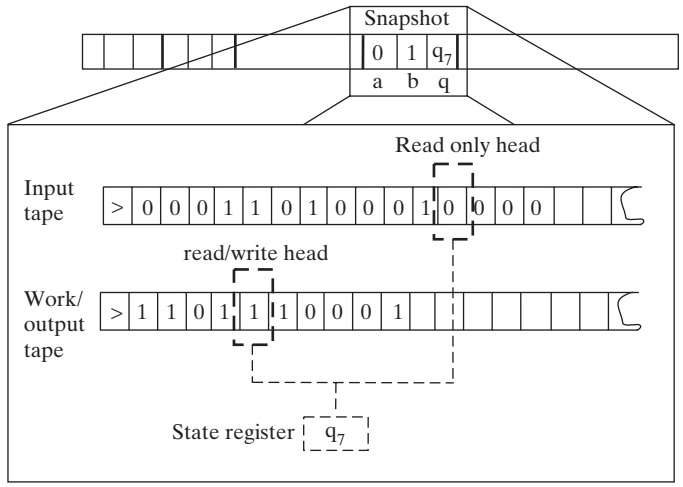

Figure 2.2. A snapshot of a TM contains the current state and symbols read by the TM at a particular step. If at the ith step M reads the symbols $0, 1$ from its tapes and is in the state q_7, then the snapshot of M at the ith step is $\langle 0, 1, q_7 \rangle$.

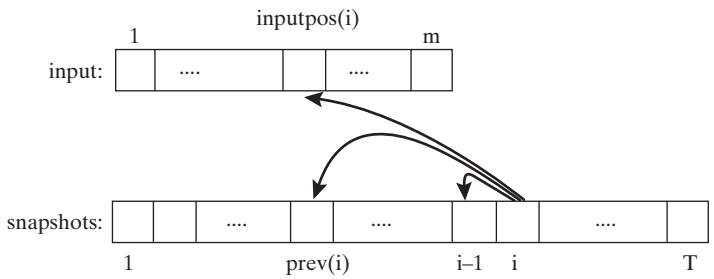

Figure 2.3. The snapshot of M at the ith step depends on its previous state (contained in the snapshot at the $(i-1)$st step), and the symbols read from the input tape, which is in position $\texttt{inputpos}(i)$, and from the work tape, which was last written to in step $\texttt{prev}(i)$.

$\texttt{inputpos}(i)$ denotes the location of M's input tape head at the ith step (recall that the input tape is read-only, so it contains $x \circ u$ throughout the computation); and $\texttt{prev}(i)$ is the last step before i when M's head was in the same cell on its work tape that it is on during step i.[6] The reason this small amount of information suffices to check the correctness of z_i is that the contents of the current cell have not been affected between step $\texttt{prev}(i)$ and step i.

In fact, since M is a deterministic TM, for every triple of values to $z_{i-1}, y_{\texttt{inputpos}(i)}, z_{\texttt{prev}(i)}$, there is at most one value of z_i that is correct. Thus there is some function F (derived from M's transition function) that maps $\{0,1\}^{2c+1}$ to $\{0,1\}^c$ such that a correct z_i satisfies

$$z_i = F(z_{i-1}, z_{\texttt{prev}(i)}, y_{\texttt{inputpos}(i)}) \tag{2.3}$$

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

Because M is oblivious, the values $\texttt{inputpos}(i)$ and $\texttt{prev}(i)$ do not depend on the particular input y. Also, as previously mentioned, these indices can be computed in polynomial-time by simulating M on a trivial input.

Now we turn the above thought exercise into a reduction. Recall that by (2.2), an input $x \in \{0,1\}^n$ is in L if and only if $M(x \circ u) = 1$ for some $u \in \{0,1\}^{p(n)}$. The previous discussion shows this latter condition occurs if and only if there exists a string $y \in \{0,1\}^{n+p(n)}$ and a sequence of strings $z_1, \ldots, z_{T(n)} \in \{0,1\}^c$ (where $T(n)$ is the number of steps M takes on inputs of length $n + p(n)$) satisfying the following four conditions:

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

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

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

2.3.5 Reducing SAT to 3SAT

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

Lemma 2.14 $\mathsf{SAT} \leq_p \mathsf{3SAT}$. ◇

Proof: We give a transformation that maps each CNF formula φ into a 3CNF formula ψ such that ψ is satisfiable if and only if φ is. We demonstrate first the case that φ is a 4CNF. Let C be a clause of φ, say $C = u_1 \vee \overline{u}_2 \vee \overline{u}_3 \vee u_4$. 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. ■

2.3.6 More thoughts on the Cook-Levin Theorem

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

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

1. We can reduce the size of the output formula φ_x if we use the efficient simulation of a standard TM by an oblivious TM (see Exercise 1.6), which manages to keep the simulation overhead logarithmic. Then for every $x \in \{0,1\}^*$, the size of the formula φ_x (and the time to compute it) is $O(T \log T)$, where T is the number of steps the machine M takes on input x (see Exercise 2.12).
2. The reduction f from an **NP**-language L to SAT presented in Lemma 2.11 not only satisfied that $x \in L \Leftrightarrow f(x) \in$ 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 modify the proof slightly (see Exercise 2.13) so that it actually supplies us with 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. As we will see later in Chapter 17, this fact is useful in studying the complexity of counting the *number* of certificates for an instance of an **NP** problem.

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

2.4 THE WEB OF REDUCTIONS

Cook and Levin had to show how *every* **NP** language can be reduced to SAT. To prove the **NP**-completeness of any other language L, we do not need to work as hard: by Theorem 2.8 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).

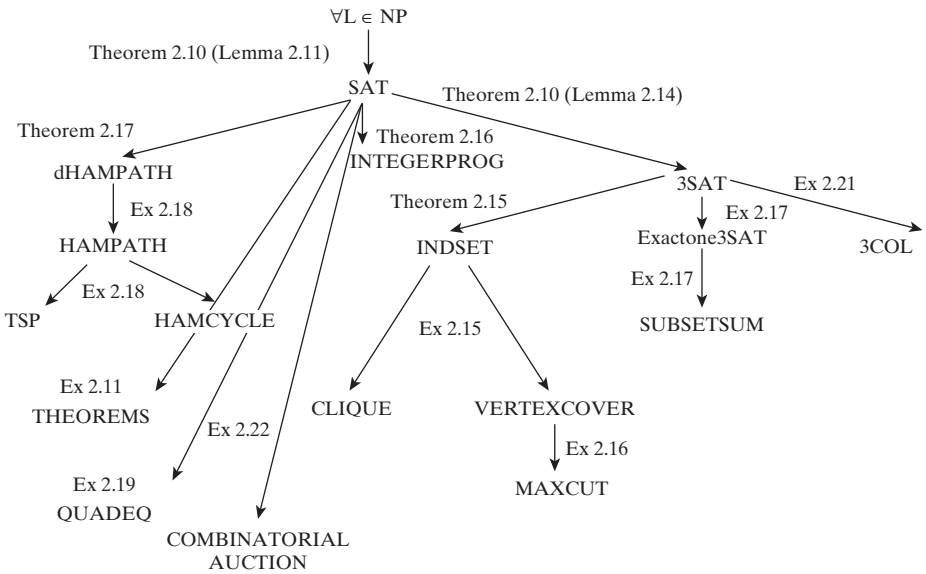

Figure 2.4. Web of reductions between the **NP**-completeness problems described in this chapter and the exercises. Thousands more are known.

Recall the problem of planning a dinner party where every pair of guests is on speaking terms, formalized in Example 0.1 as the language

$$\texttt{INDSET} = \{\langle G, k\rangle : G \text{ has independent set of size } k\}$$

Theorem 2.15 `INDSET` *is* **NP***-complete.* ◇

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

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

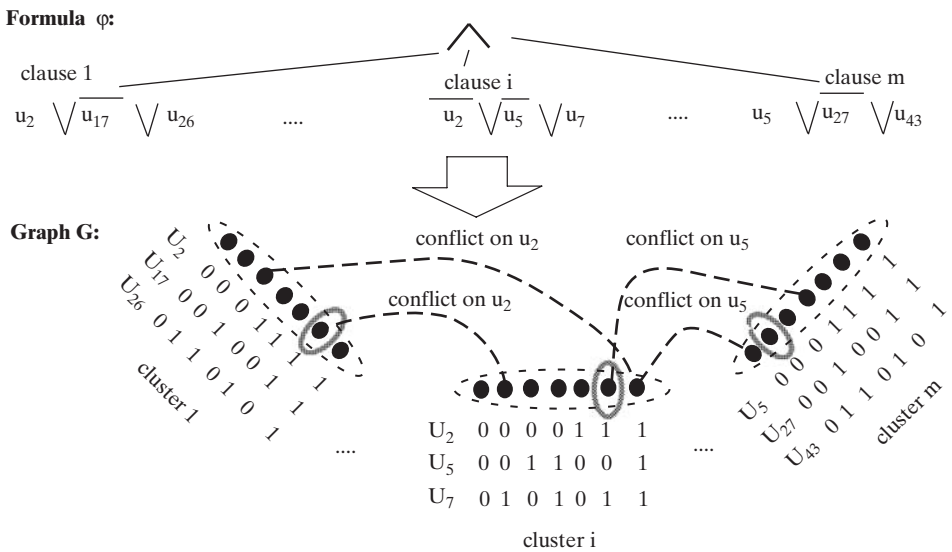

Figure 2.5. We transform a 3*CNF* formula φ with m clauses into a graph G with $7m$ vertices as follows: each clause C is associated with a cluster of 7 vertices corresponding to the 7 possible satisfying assignments to the variables C depends on. We put edges between any two vertices in the same cluster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

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

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

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

Theorem 2.16 `0/1 IPROG` *is* **NP***-complete.* ◇

PROOF: `0/1 IPROG` is clearly in **NP** since the assignment can serve as the certificate. To reduce `SAT` to `0/1 IPROG`, note that every CNF formula can be easily expressed as

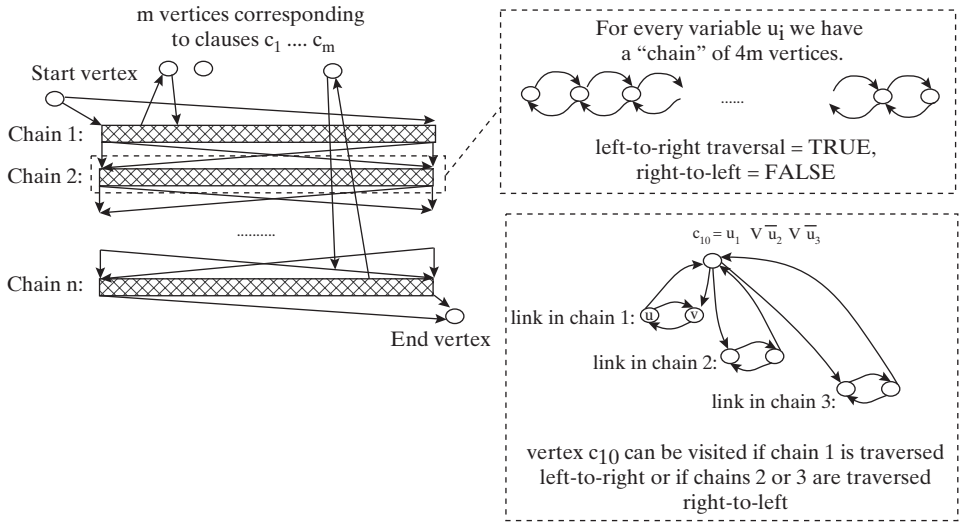

Figure 2.6. Reducing SAT to dHAMPATH. A formula φ with n variables and m clauses is mapped to a graph G that has m vertices corresponding to the clauses and n doubly linked chains, each of length $4m$, corresponding to the variables. Traversing a chain left to right corresponds to setting the variable 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.

an integer program by expressing every clause as an inequality. For example, the clause $u_1 \vee \overline{u}_2 \vee \overline{u}_3$ can be expressed as $u_1 + (1 - u_2) + (1 - u_3) \geq 1$. ■

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

Theorem 2.17 dHAMPATH *is* **NP***-complete.* ◇

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

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

We put edges from the starting vertex v_{start} to the two extreme points of the first chain. We also put edges from the extreme points of the jth 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 nth 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 jth 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 $4m$ vertices, which is more than sufficient for this. We now prove that $\varphi \in \text{SAT} \Leftrightarrow G \in \text{dHAMPATH}$:

($\varphi \in \text{SAT} \Rightarrow G \in \text{dHAMPATH}$): Suppose that φ has a satisfying assignment $u_1, \ldots, u_n$. 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 jth chain in left-to-right order if $u_j = 1$ and travels it in right-to-left order if $u_j = 0$. This path visits all the vertices except for those corresponding to clauses. Yet, if u is a satisfying assignment then the path can be easily modified to visit all the vertices corresponding to clauses: For each clause C, there is at least one literal that is true, and we can use one link on the chain corresponding to that literal to "skip" to the vertex v_C and continue on as before.

($G \in \text{dHAMPATH} \Rightarrow \varphi \in \text{SAT}$): Suppose that G has an Hamiltonian path P. We first note that the path P must start in v_{start} (as it has no incoming edges) and end at v_{end} (as it has no outgoing edges). Furthermore, we claim that P needs to traverse all the chains in order and, within each chain, traverse it either in left-to-right order or right-to-left order. This would be immediate if the path did not use the edges from a chain to the vertices corresponding to clauses. The claim holds because if a Hamiltonian path takes the edge $u \to w$, 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 (i.e., will find every outgoing edge already taken) 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 jth 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 7.6) has led to a rich understanding of complexity above and beyond **NP**-completeness. Much of complexity theory and cryptography today (thus, many chapters of this book) consists of using reductions to make connections between disparate complexity theoretic conjectures. Why do complexity theorists excel at reductions but not at actually proving lower bounds on Turing machines? Maybe human creativity is more adaptible to gadget-making and algorithm-design (after all, a reduction is merely an algorithm to transform one problem into another) than to proving lower bounds on Turing machines.

2.5 DECISION VERSUS SEARCH

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

the sense that if the decision problem can be solved (and hence **P** = **NP**), then the search version of any **NP** problem can also be solved in polynomial time.

Theorem 2.18 *Suppose that* **P** = **NP**. *Then, for every* **NP** *language L and a verifier TM M for L (as per Definition 2.1), there is a polynomial-time TM B that on input* $x \in L$ *outputs a certificate for x (with respect to the language L and TM M).* ◇

Proof: We need to show that if **P** = **NP**, then for every polynomial-time TM M and polynomial $p(n)$, there is a polynomial-time TM B with the following property: for every $x \in \{0,1\}^n$, if there is $u \in \{0,1\}^{p(n)}$ such that $M(x,u) = 1$ (i.e., a certificate that x is in the language verified by M) then $|B(x)| = p(n)$ and $M(x, B(x)) = 1$.

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 first substitute $x_1 = 0$ and then $x_1 = 1$ in φ (this transformation, which simplifies and shortens the formula a little and leaves 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 (at least one of them is). Say the first is satisfiable. Then we fix $x_1 = 0$ from now on and continue with the simplified formula. Continuing this way, we end up fixing all n variables while ensuring that each intermediate formula is satisfiable. Thus the final assignment to the variables satisfies φ.

To solve the search problem for an arbitrary **NP**-language L, we use the fact that the reduction of Theorem 2.10 from L to SAT is actually a *Levin* reduction. This means that we have a polynomial-time computable function f such that not only $x \in L \Leftrightarrow f(x) \in \mathsf{SAT}$ 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. ■

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

2.6 coNP, EXP, AND NEXP

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

2.6.1 coNP

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

Definition 2.19 $\mathbf{coNP} = \left\{ L : \overline{L} \in \mathbf{NP} \right\}$. ◇

coNP is *not* the complement of the class **NP**. In fact, **coNP** and **NP** have a nonempty intersection, since every language in **P** is in **NP** ∩ **coNP** (see Exercise 2.23). The following is an example of a **coNP** language: $\overline{\text{SAT}} = \{\varphi : \varphi \text{ is not satisfiable}\}$. Students sometimes mistakenly convince themselves that $\overline{\text{SAT}}$ 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 speaking we prefer the following definition of **coNP** (which is easily shown to be equivalent to the first, see Exercise 2.24).

Definition 2.20 *(**coNP**, alternative definition)* For every $L \subseteq \{0,1\}^*$, we say that $L \in$ **coNP** if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time TM M such that for every $x \in \{0,1\}^*$,

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

Note the use of the "∀" quantifier in this definition where Definition 2.1 used ∃.

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

EXAMPLE 2.21

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

$$\text{TAUTOLOGY} = \{\varphi : \varphi \text{ is a } \textit{tautology}\text{—a Boolean formula that is satisfied by every assignment}\}$$

It is clearly in **coNP** by Definition 2.20, and so all we have to show is that for every $L \in$ **coNP**, $L \leq_p \text{TAUTOLOGY}$. 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 TAUTOLOGY iff $x \in L$, and this completes the description of the reduction.

It's not hard to see that if **P** = **NP**, then **NP** = **coNP** = **P** (Exercise 2.25). Put in the contrapositive: If we can show that **NP** ≠ **coNP**, then we have shown **P** ≠ **NP**. Most researchers believe that **NP** ≠ **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.

2.6.2 EXP and NEXP

In Claim 2.4, we encountered the class **EXP** = $\cup_{c\geq 1}$**DTIME**(2^{n^c}), which is the exponential-time analog of **P**. The exponential-time analog of **NP** is the class **NEXP**, defined as $\cup_{c\geq 1}$**NTIME**(2^{n^c}).

As was seen earlier, 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 2.22 *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

$$L_{\text{pad}} = \left\{ \langle x, 1^{2^{|x|^c}} \rangle : x \in L \right\} \tag{2.4}$$

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 0 (i.e., halt without going to the state $q_{\texttt{accept}}$). If y is of this form, then simulate 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}. ■

The *padding* technique used in this proof, whereby we transform a language by "padding" every string in a language with a string of (useless) symbols, is also used in several other results in complexity theory (see, e.g., Section 14.4.1). In many settings, it can be used to show that equalities between complexity classes "scale up"; that is, if two different types 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**, many complexity classes studied in this book are 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): Recognizing the correctness of an answer is often much easier than coming up with the answer. Appreciating a Beethoven sonata is far easier than composing the sonata; verifying the solidity of a design for a suspension bridge is easier (to a civil engineer anyway!) than coming up with a good design; verifying the proof of a theorem is easier than coming up with a proof itself (a fact referred to in Gödel's letter quoted at the start of the chapter), and so forth. In such cases, coming up with the

right answer seems to involve *exhaustive search* over an exponentially large set. The **P** versus **NP** question asks whether exhaustive search can be avoided in general. It seems obvious to most people—and the basis of many false proofs proposed by amateurs—that exhaustive search cannot be avoided. Unfortunately, turning this intuition into a proof has proved difficult.

2.7.2 NP and mathematical proofs

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

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

Gödel's quote from 1956 at the start of this chapter asks whether this problem can be solved in say quadratic time. He observes that this is a finite version of Hilbert's *Entscheidungsproblem*, which asked for an algorithmic decision procedure for checking whether a given mathematical statement has a proof (with no upper bound specified on the length of the proof). He points out that if THEOREMS can be solved in quadtratic time, then the undecidability of the *Entscheidungsproblem* would become less depressing, since we are usually only interested in theorems whose proof is not too long (say, fits in a few books).

Exercise 2.11 asks you to prove that THEOREMS is **NP**-complete. Hence the **P** versus **NP** question is a *rephrasing* of Gödel's question, which asks whether or not there is a algorithm that finds mathematical proofs in time polynomial in the length of the proof.

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

2.7.3 What if 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 computational Utopia. Mathematicians could be replaced by efficient theorem-discovering programs (a fact pointed out in Kurt Gödel's 1956 letter and recovered two 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. Artificial intelligence (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. Very large scale integration (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.[7] Of course, in some cases, it took scientists centuries to come up with the simplest theories explaining the known data. This approach can be used to also approach nonscientific problems: One could find the simplest theory that explains, say, the list of books from the New York *Times*' bestseller list. (Of course even finding the right definition of "simplest" might require some breakthroughs in artificial intelligence and understanding natural language that themselves would use **NP**-algorithms.) All these applications will be a consequence of our study of the polynomial hierarchy in Chapter 5. (The problem of finding the smallest "theory" is closely related to problems like MIN-EQ-DNF studied in Chapter 5.)

Somewhat intriguingly, this Utopia would have no need for randomness. As we will later see, if $\mathbf{P} = \mathbf{NP}$, then randomized algorithms would buy essentially no efficiency gains over deterministic algorithms; see Chapter 7. (Philosophers should ponder this one.)

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

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

2.7.4 What if NP = coNP?

If $\mathbf{NP} = \mathbf{coNP}$, the consequences still seem dramatic. Mostly, they have to do with existence of short certificates for statements that do not seem to have any. To give an example, consider the **NP**-complete problem of finding whether or not a set of multivariate polynomials has a common root (see Exercise 2.20). In other words, it is **NP** complete to decide whether a system of equations of the following type has a solution:

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

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

If a solution exists, then that solution serves as a *certificate* to this effect (of course, we have to also show that the solution can be described using a polynomial number of

[7] Occam's Razor is a well-known principle in philosophy, but it has found new life in *machine learning*, a subfield of computer science. Valiant's Theory of the Learnable [Val84] gives mathematical basis for Occam's Razor. This theory is deeply influenced by computational complexity; an excellent treatment appears in the book of Kearns and Vazirani [KV94]. If $\mathbf{P} = \mathbf{NP}$, then many interesting problems in machine learning turn out to have polynomial-time algorithms.

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_is—and hence the number of bits used to represent them—could be exponential in n, m. (And it is simple to construct f_is 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 could be even greater than the effect of Hilbert's Nullstellensatz. (The Nullstellensatz appears again in Chapters 15 and 16.)

2.7.5 Is there anything between NP and NP-complete?

NP-completeness has been an extremely useful and influential theory, since thousands of useful problems are known to be **NP**-complete (and hence they are presumably not in **P**). However, there remain a few interesting **NP** problems that are neither known to be in **P** nor known to be **NP**-complete. For such problems, it would be nice to have some other way to show that they are nevertheless difficult to solve, but we know of very few ways of quantifying this. Sometimes researchers turn the more famous problems of this type into bona fide classes of their own. Some examples are the problem of factoring integers (used in Chapter 9) or the so-called *unique games labeling problem* (Chapter 22). The complexity of these problems is related to those of many other problems. Similarly, Papadimitriou [Pap90] has defined numerous interesting classes between **P** and **NP** that capture the complexity of various interesting problems. The most important is **PPAD**, which captures the problem of finding *Nash Equilibria* in two-person games.

Sometimes we can show that some of these problems are unlikely to be **NP**-complete. We do this by showing that *if* the problem is **NP**-complete, then this violates some other conjecture (that we believe almost as much as **P** $\neq$ **NP**); we'll see such results for the *graph isomorphism* problem in Section 8.1.3.

Another interesting result in Section 3.3 called *Ladner's Theorem* shows that if **P** $\neq$ **NP**, then there exist problems that are neither in **P** nor **NP**-complete.

2.7.6 Coping with NP hardness

NP-complete problems turn up in a great many applications, from flight scheduling to genome sequencing. What do you do if the problem you need to solve turns out to be **NP**-complete? At the outset, the situation looks bleak: If **P** $\neq$ **NP**, then there simply does not *exist* an efficient algorithm to solve such a problem.[8] However, there may still be some hope: **NP**-completeness only means that (assuming **P** $\neq$ **NP**) the problem does

[8] Not however that sometimes simple changes in the problem statement can dramatically change its complexity. Therefore, modeling a practical situation with an abstract problem requires great care; one must take care not to unnecessarily model a simple setting with an **NP**-complete problem.

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, find the shortest route that travels through all of them. Assume that you are indeed in charge of coming up with travel plans for traveling salespeople that need to visit various cities around your country. Does the fact that TSP is **NP**-complete means that you are bound to do a hopelessly suboptimal job? This does not have to be the case.

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

It turns out that neither of these observations on its own is sufficient to make the problem tractable. The TSP problem is still **NP**-complete even for Euclidean distances. Also if $\mathbf{P} \neq \mathbf{NP}$, then TSP is hard to approximate within any constant factor. However, *combining* the two observations together actually helps: For every ϵ there is a $\text{poly}(n(\log n)^{O(1/\epsilon)})$-time algorithm that given Euclidean distances between n cities comes up with a tour that is at most a factor of $(1+\epsilon)$ worse than the optimal tour [Aro96].

Thus, discovering that your problem is **NP**-complete should not be cause for immediate despair. Rather you should view this as indication that a more careful modeling of the problem is needed, letting the literature on complexity and algorithms guide you as to what features might make the problem more tractable. Alternatives to worst-case exact computation are explored in Chapters 18 and 11, which investigate *average-case complexity* and *approximation algorithms* respectively.

2.7.7 Finer explorations of time complexity

We have tended to focus the discussion in this chapter on the difference between polynomial and nonpolynomial time. Researchers have also explored finer issues about time complexity. For instance, consider a problem like INDSET. We believe that it cannot be solved in polynomial time. But what exactly is its complexity? Is it $n^{O(\log n)}$, or $2^{n^{0.2}}$ or $2^{n/10}$? Most researchers believe it is actually $2^{\Omega(n)}$. The intuitive feeling is that the trivial algorithm of enumerating all possible subsets is close to optimal.

It is useful to test this intuition when the size of the optimum independent set is at most k. The trivial algorithm of enumerating all k-size subsets of vertices takes $\binom{n}{k} \approx n^k$ time when $k \ll n$. (In fact, think of k as an arbitrarily large constant.) Can we do any better, say $2^k \text{poly}(n)$ time, or more generally $f(k)\text{poly}(n)$ time for some function f? The theory of *fixed parameter intractability* studies such questions. There is a large set of **NP** problems including INDSET that are *complete* in this respect, which means that one of them has a $f(k)\text{poly}(n)$ time algorithm iff all of them do. Needless to say, this

notion of "completeness" is with respect to some special notion of reducibility; the book [FG06] is a good resource on this topic.

Similarly, one can wonder if there is an extension of **NP**-completeness that butresses the intuition that the true complexity of INDSET and many other **NP**-complete problems is $2^{\Omega(n)}$ rather than simply nonpolynomial. Impagliazzo, Paturi, and Zane [IPZ98] have such a theory, including a notion of *reducibility* tailored to studying this issue.

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**. We can also define **NP** using nondeterministic 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 3*CNF* 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.
- The class **coNP** is the set of complements of **NP**-languages. We believe that **coNP** $\neq$ **NP**. This is a stronger assumption than **P** $\neq$ **NP**.

CHAPTER NOTES AND HISTORY

Since the 1950s, Soviet scientists were aware of the undesirability of using exhaustive or brute force search, (which they called *perebor*,) for combinatorial problems, and asked the question of whether certain problems *inherently* require such search (see [Tra84] for a history). In the West, the first published description of this issue is by Edmonds [Edm65], in the paper quoted in the previous chapter. However, on both sides of the iron curtain, it took some time to realize the right way to formulate the problem and to arrive at the modern definition of the classes **NP** and **P**. Amazingly, in his 1956 letter to von Neumann we quoted earlier, Gödel essentially asks the question of **P** vs. **NP**, although there is no hint that he realized that one of the particular problems he mentions is **NP**-complete. Unfortunately, von Neumann was very sick at the time, and as far as we know, no further research was done by either on them on this problem, and the letter was only discovered in the 1980s.

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

between eastern and western scientists. Trakktenbrot's survey [Tra84] describes Levin's discovery and also gives an accurate translation of Levin's paper. See Sipser's survey [Sip92] for more on the history of **P** and **NP** and a full translation of Gödel's remarkable letter.

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

Aaronson [Aar05] surveys various attempts to solve **NP** complete problems via "nontraditional" computing devices.

Even if **NP** $\neq$ **P**, this does not necessarily mean that all of the utopian applications mentioned in Section 2.7.3 are automatically ruled out. 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 18 for a more detailed study of *average-case* complexity. Also, Impagliazzo [Imp95b] has an excellent survey on this topic.

An intriguing possibility is that it is simply impossible to resolve the **P** vs. **NP** question using the accepted axioms of mathematics: This has turned out to be the case with some other questions such as Cantor's "Continuum Hypothesis." Aaronson's survey [Aar03] explores this possibility.

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

EXERCISES

2.1. Prove that allowing the certificate to be of size at *most* $p(|x|)$ (rather than equal to $p(|x|)$) in Definition 2.1 makes no difference. That is, show that for every polynomial-time Turing machine M and polynomial $p : \mathbb{N} \to \mathbb{N}$, the language

$$\{x : \exists u \text{ s.t. } |u| \leq p(|x|) \text{ and } M(x,u) = 1\}$$

is in **NP**.

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

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

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

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

Which ones of them are in **P**?

H531

2.3. Let LINEQ denote the set of satisfiable rational linear equations. That is, LINEQ consists of the set of all pairs $\langle A, \mathbf{b} \rangle$ where A is an $m \times n$ rational matrix and $\mathbf{b}$ is an m dimensional rational vector, such that $A\mathbf{x} = \mathbf{b}$ for some n-dimensional vector $\mathbf{x}$. Prove that LINEQ is in **NP** (the key is to prove that if there exists such a vector $\mathbf{x}$, then there exists an $\mathbf{x}$ whose coefficients can be represented using a number of bits that is polynomial in the representation of A,$\mathbf{b}$). (Note that LINEQ is actually in **P**: Can you show this?)

H531

2.4. Show that the Linear Programming problem from Example 2.3 is in **NP**. (Again, this problem is actually in **P**, though by a highly nontrivial algorithm [Kha79].)

H531

2.5. [Pra75] Let $\texttt{PRIMES} = \{ \llcorner n \lrcorner : n \text{ is prime} \}$. Show that $\texttt{PRIMES} \in \mathbf{NP}$. You can use the following fact: A number n is prime iff for every prime factor q of $n-1$, there exists a number $a \in \{2, \ldots, n-1\}$ satisfying $a^{n-1} = 1 \pmod{n}$ but $a^{(n-1)/q} \neq 1 \pmod{n}$.

H531

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

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

(b) Prove that there is such a universal NDTM that runs on these inputs for at most Ct steps.

H532

2.7. Prove Parts 2 and 3 of Theorem 2.8.

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

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

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

2.11. 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. (You don't need to know anything about ZF.)

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

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

H532

2.12. Show that for every time-constructible $T : \mathbb{N} \to \mathbb{N}$, if $L \in \mathbf{NTIME}(T(n))$, we can give a polynomial-time Karp reduction from L to 3SAT that transforms instances of

size n into 3CNF formulae of size $O(T(n) \log T(n))$. Can you make this reduction also run in $O(T(n) \text{poly}(\log T(n)))$?

2.13. Recall that a reduction f from an **NP**-language L to an **NP**-languages L' is *parsimonious* if the number of certificates of f is equal to the number of certificates of $f(x)$.

(a) Prove that the reduction from every **NP**-language L to SAT presented in the proof of Lemma 2.11 can be made parsimonious.

H532

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

2.14. Cook [Coo71] used a somewhat different notion of reduction: A language L is *polynomial-time Cook reducible* to a language L' if there is a polynomial time TM M that, given an *oracle* for deciding L', can decide L. An oracle for L' is a magical extra tape given to M, such that whenever M writes a string on this tape and goes into a special "invocation" state, then the string—in a single step!—gets overwritten by 1 or 0 depending upon whether the string is or is not in L'; see Section 3.4 for a more precise definition.

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

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

H532

2.16. 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.

2.17. 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.

H532

2.18. Prove that the language HAMPATH of *undirected* graphs with Hamiltonian paths is **NP**-complete. Prove that the language TSP described in Example 2.3 is **NP**-complete. Prove that the language HAMCYCLE of undirected graphs that contain Hamiltonian cycle (a simple cycle involving all the vertices) is **NP**-complete.

2.19. 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 $\sum_{i,j \in [n]} a_{i,j} u_i u_j = b$) where addition is modulo 2. Show that QUADEQ is **NP**-complete.

H532

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

H532

2.21. Prove that 3COL (see Exercise 2.2) is **NP**-complete.

H532

2.22. In a typical auction of n items, the auctioneer will sell the ith item to the person that gave it the highest bid. However, sometimes the items sold are related to one another (e.g., think of lots of land that may be adjacent to one another) and so people may be willing to pay a high price to get, say, the three items $\{2, 5, 17\}$, but only if they get all of them together. In this case, deciding what to sell to whom might not be an easy task. The COMBINATORIAL AUCTION problem is to decide, given numbers n, k, and a list of pairs $\{\langle S_i, x_i\rangle\}_{i=1}^{m}$ where S_i is a subset of $[n]$ and x_i is an integer, whether there exist disjoint sets $S_{i_1}, \ldots, S_{i_\ell}$ such that $\sum_{j=1}^{\ell} x_{i_j} \geq k$. That is, if x_i is the amount a bidder is willing to pay for the set S_i, then the problem is to decide if the auctioneer can sell items and get a revenue of at least k, under the obvious condition that he can't sell the same item twice. Prove that COMBINATORIAL AUCTION is **NP**-complete.

H532

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

2.24. Prove that Definitions 2.19 and 2.20 do indeed define the same class **coNP**.

2.25. Prove that if $\mathbf{P} = \mathbf{NP}$, then $\mathbf{NP} = \mathbf{coNP}$.

2.26. Show that $\mathbf{NP} = \mathbf{coNP}$ iff 3SAT and TAUTOLOGY are polynomial-time reducible to one another.

2.27. Give a definition of **NEXP** without using NDTMs, analogous to Definition 2.1 of the class **NP**, and prove that the two definitions are equivalent.

2.28. We say that a language is **NEXP**-complete if it is in **NEXP** and every language in **NEXP** is polynomial-time reducible to it. Describe a **NEXP**-complete language L. Prove that if $L \in \mathbf{EXP}$ then $\mathbf{NEXP} = \mathbf{EXP}$.

2.29. Suppose $L_1, L_2 \in \mathbf{NP} \cap \mathbf{coNP}$. Then show that $L_1 \oplus L_2$ is in $\mathbf{NP} \cap \mathbf{coNP}$, where $L_1 \oplus L_2 = \{x : x \text{ is in exactly one of } L_1, L_2\}$.

2.30. (Berman's Theorem 1978) A language is called *unary* if every string in it is of the form 1^i (the string of i ones) for some $i > 0$. Show that if there exists an **NP**-complete unary language then $\mathbf{P} = \mathbf{NP}$. (See Exercise 6.9 for a strengthening of this result.)

H532

2.31. Define the language UNARY SUBSET SUM to be the variant of the SUBSET SUM problem of Exercise 2.17 where all numbers are represented by the *unary* representation (i.e., the number k is represented as 1^k). Show that UNARY SUBSET SUM is in **P**.

H532

2.32. Prove that if every *unary* **NP**-language is in **P** then $\mathbf{EXP} = \mathbf{NEXP}$. (A language L is unary iff it is a subset of $\{1\}^*$, see Exercise 2.30.)

2.33. Let $\Sigma_2\mathsf{SAT}$ denote the following decision problem: Given a quantified formula ψ of the form

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

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

2.34. Suppose that you are given a graph G and a number K and are told that either (i) the smallest vertex cover (see Exercise 2.15) of G is of size at most K or (ii) it is of size at least $3K$. Show a polynomial-time algorithm that can distinguish between these two cases. Can you do it with a smaller constant than 3? Since VERTEX COVER is **NP**-hard, why does this algorithm not show that $\mathbf{P} = \mathbf{NP}$?