7

TIME COMPLEXITY

Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory. In this final part of the book we introduce computational complexity theory—an investigation of the time, memory, or other resources required for solving computational problems. We begin with time.

Our objective in this chapter is to present the basics of time complexity theory. First we introduce a way of measuring the time used to solve a problem. Then we show how to classify problems according to the amount of time required. After that we discuss the possibility that certain decidable problems require enormous amounts of time and how to determine when you are faced with such a problem.

7.1 MEASURING COMPLEXITY

Let's begin with an example. Take the language $A = \{0^k 1^k \mid k \geq 0\}$. Obviously A is a decidable language. How much time does a single-tape Turing machine need to decide A? We examine the following single-tape TM M_1 for A. We give

the Turing machine description at a low level, including the actual head motion on the tape so that we can count the number of steps that M_1 uses when it runs.

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

We *analyze* the algorithm for TM M_1 deciding A to determine how much time it uses.

The number of steps that an algorithm uses on a particular input may depend on several parameters. For instance, if the input is a graph, the number of steps may depend on the number of nodes, the number of edges, and the maximum degree of the graph, or some combination of these and/or other factors. For simplicity we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters. In ***worst-case analysis***, the form we consider here, we consider the longest running time of all inputs of a particular length. In ***average-case analysis***, we consider the average of all the running times of inputs of a particular length.

DEFINITION 7.1

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

BIG-O AND SMALL-O NOTATION

Because the exact running time of an algorithm often is a complex expression, we usually just estimate it. In one convenient form of estimation, called ***asymptotic analysis***, we seek to understand the running time of the algorithm when it is run on large inputs. We do so by considering only the highest order term of the expression for the running time of the algorithm, disregarding both the coefficient of that term and any lower order terms, because the highest order term dominates the other terms on large inputs.

For example, the function $f(n) = 6n^3 + 2n^2 + 20n + 45$ has four terms, and the highest order term is $6n^3$. Disregarding the coefficient 6, we say that f is asymptotically at most n^3. The ***asymptotic notation*** or ***big-O notation*** for describing this relationship is $f(n) = O(n^3)$. We formalize this notion in the following definition. Let $\mathcal{R}^+$ be the set of nonnegative real numbers.

DEFINITION 7.2

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = O(g(n))}$ if positive integers c and n_0 exist such that for every integer $n \geq n_0$

$$f(n) \leq c\, g(n).$$

When $f(n) = O(g(n))$ we say that $g(n)$ is an ***upper bound*** for $f(n)$, or more precisely, that $g(n)$ is an ***asymptotic upper bound*** for $f(n)$, to emphasize that we are suppressing constant factors.

Intuitively, $f(n) = O(g(n))$ means that f is less than or equal to g if we disregard differences up to a constant factor. You may think of O as representing a suppressed constant. In practice, most functions f that you are likely to encounter have an obvious highest order term h. In that case write $f(n) = O(g(n))$, where g is h without its coefficient.

EXAMPLE 7.3

Let $f_1(n)$ be the function $5n^3 + 2n^2 + 22n + 6$. Then, selecting the highest order term $5n^3$ and disregarding its coefficient 5 gives $f_1(n) = O(n^3)$.

Let's verify that this result satisfies the formal definition. We do so by letting c be 6 and n_0 be 10. Then, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ for every $n \geq 10$.

In addition, $f_1(n) = O(n^4)$ because n^4 is larger than n^3 and so is still an asymptotic upper bound on f_1.

However, $f_1(n)$ is not $O(n^2)$. Regardless of the values we assign to c and n_0, the definition remains unsatisfied in this case.

EXAMPLE 7.4

The big-O interacts with logarithms in a particular way. Usually when we use logarithms we must specify the base, as in $x = \log_2 n$. The base 2 here indicates that this equality is equivalent to the equality $2^x = n$. Changing the value of the base b changes the value of $\log_b n$ by a constant factor, owing to the identity $\log_b n = \log_2 n / \log_2 b$. Thus, when we write $f(n) = O(\log n)$, specifying the base is no longer necessary because we are suppressing constant factors anyway.

Let $f_2(n)$ be the function $3n \log_2 n + 5n \log_2 \log_2 n + 2$. In this case we have $f_2(n) = O(n \log n)$ because $\log n$ dominates $\log \log n$.

Big-O notation also appears in arithmetic expressions such as the expression $f(n) = O(n^2) + O(n)$. In that case each occurrence of the O symbol represents a different suppressed constant. Because the $O(n^2)$ term dominates the $O(n)$ term, that expression is equivalent to $f(n) = O(n^2)$. When the O symbol occurs in an exponent, as in the expression $f(n) = 2^{O(n)}$, the same idea applies. This expression represents an upper bound of 2^{cn} for some constant c.

The expression $f(n) = 2^{O(\log n)}$ occurs in some analyses. Using the identity $n = 2^{\log_2 n}$ and thus that $n^c = 2^{c \log_2 n}$, we see that $2^{O(\log n)}$ represents an upper bound of n^c for some c. The expression $n^{O(1)}$ represents the same bound in a different way, because the expression $O(1)$ represents a value that is never more than a fixed constant.

Frequently we derive bounds of the form n^c for c greater than 0. Such bounds are called ***polynomial bounds***. Bounds of the form $2^{(n^\delta)}$ are called ***exponential bounds*** when δ is a real number greater than 0.

Big-O notation has a companion called ***small-o notation***. Big-O notation says that one function is asymptotically *no more than* another. To say that one function is asymptotically *less than* another we use small-o notation. The difference between the big-O and small-o notations is analogous to the difference between $\leq$ and $<$.

DEFINITION 7.5

Let f and g be functions $f, g: \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $\boldsymbol{f(n) = o(g(n))}$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In other words, $f(n) = o(g(n))$ means that, for any real number $c > 0$, a number n_0 exists, where $f(n) < c\,g(n)$ for all $n \geq n_0$.

EXAMPLE 7.6

The following are easy to check.

1. $\sqrt{n} = o(n)$.
2. $n = o(n \log \log n)$.
3. $n \log \log n = o(n \log n)$.
4. $n \log n = o(n^2)$.
5. $n^2 = o(n^3)$.

However, $f(n)$ is never $o(f(n))$.

ANALYZING ALGORITHMS

Let's analyze the TM algorithm we gave for the language $A = \{0^k1^k \mid k \geq 0\}$. We repeat the algorithm here for convenience.

M_1 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat if both 0s and 1s remain on the tape:
3. Scan across the tape, crossing off a single 0 and a single 1.
4. If 0s still remain after all the 1s have been crossed off, or if 1s still remain after all the 0s have been crossed off, *reject*. Otherwise, if neither 0s nor 1s remain on the tape, *accept*."

To analyze M_1 we consider each of its four stages separately. In stage 1, the machine scans across the tape to verify that the input is of the form 0^*1^*. Performing this scan uses n steps. As we mentioned earlier, we typically use n to represent the length of the input. Repositioning the head at the left-hand end of the tape uses another n steps. So the total used in this stage is $2n$ steps. In big-O notation we say that this stage uses $O(n)$ steps. Note that we didn't mention the repositioning of the tape head in the machine description. Using asymptotic notation allows us to omit details of the machine description that affect the running time by at most a constant factor.

In stages 2 and 3, the machine repeatedly scans the tape and crosses off a 0 and 1 on each scan. Each scan uses $O(n)$ steps. Because each scan crosses off two symbols, at most $n/2$ scans can occur. So the total time taken by stages 2 and 3 is $(n/2)O(n) = O(n^2)$ steps.

In stage 4 the machine makes a single scan to decide whether to accept or reject. The time taken in this stage is at most $O(n)$.

Thus the total time of M_1 on an input of length n is $O(n) + O(n^2) + O(n)$, or $O(n^2)$. In other words, its running time is $O(n^2)$, which completes the time analysis of this machine.

Let's set up some notation for classifying languages according to their time requirements.

DEFINITION 7.7

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Recall the language $A = \{0^k1^k \mid k \geq 0\}$. The preceding analysis shows that $A \in \text{TIME}(n^2)$ because M_1 decides A in time $O(n^2)$ and $\text{TIME}(n^2)$ contains all languages that can be decided in $O(n^2)$ time.

Is there a machine that decides A asymptotically more quickly? In other words, is A in TIME($t(n)$) for $t(n) = o(n^2)$? We can improve the running time by crossing off two 0s and two 1s on every scan instead of just one because doing so cuts the number of scans by half. But that improves the running time only by a factor of 2 and doesn't affect the asymptotic running time. The following machine, M_2, uses a different method to decide A asymptotically faster. It shows that $A \in \text{TIME}(n \log n)$.

M_2 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Repeat as long as some 0s and some 1s remain on the tape:
3. Scan across the tape, checking whether the total number of 0s and 1s remaining is even or odd. If it is odd, *reject*.
4. Scan again across the tape, crossing off every other 0 starting with the first 0, and then crossing off every other 1 starting with the first 1.
5. If no 0s and no 1s remain on the tape, *accept*. Otherwise, *reject*."

Before analyzing M_2, let's verify that it actually decides A. On every scan performed in stage 4, the total number of 0s remaining is cut in half and any remainder is discarded. Thus, if we started with 13 0s, after stage 4 is executed a single time only 6 0s remain. After subsequent executions of this stage, 3, then 1, and then 0 remain. This stage has the same effect on the number of 1s.

Now we examine the even/odd parity of the number of 0s and the number of 1s at each execution of stage 3. Consider again starting with 13 0s and 13 1s. The first execution of stage 3 finds an odd number of 0s (because 13 is an odd number) and an odd number of 1s. On subsequent executions an even number (6) occurs, then an odd number (3), and an odd number (1). We do not execute this stage on 0 0s or 0 1s because of the condition on the repeat loop specified in stage 2. For the sequence of parities found (odd, even, odd, odd) if we replace the evens with 0s and the odds with 1s and then reverse the sequence, we obtain 1101, the binary representation of 13, or the number of 0s and 1s at the beginning. The sequence of parities always gives the reverse of the binary representation.

When stage 3 checks to determine that the total number of 0s and 1s remaining is even, it actually is checking on the agreement of the parity of the 0s with the parity of the 1s. If all parities agree, the binary representations of the numbers of 0s and of 1s agree, and so the two numbers are equal.

To analyze the running time of M_2, we first observe that every stage takes $O(n)$ time. We then determine the number of times that each is executed. Stages 1 and 5 are executed once, taking a total of $O(n)$ time. Stage 4 crosses off at least half the 0s and 1s each time it is executed, so at most $1 + \log_2 n$ iterations of the repeat loop occur before all get crossed off. Thus the total time of stages 2, 3, and 4 is $(1 + \log_2 n)O(n)$, or $O(n \log n)$. The running time of M_2 is $O(n) + O(n \log n) = O(n \log n)$.

Earlier we showed that $A \in \text{TIME}(n^2)$, but now we have a better bound—namely, $A \in \text{TIME}(n \log n)$. This result cannot be further improved on single-tape Turing machines. In fact, any language that can be decided in $o(n \log n)$ time on a single-tape Turing machine is regular, as Problem 7.47 asks you to show.

We can decide the language A in $O(n)$ time (also called ***linear time***) if the Turing machine has a second tape. The following two-tape TM M_3 decides A in linear time. Machine M_3 operates differently from the previous machines for A. It simply copies the 0s to its second tape and then matches them against the 1s.

M_3 = "On input string w:

1. Scan across the tape and *reject* if a 0 is found to the right of a 1.
2. Scan across the 0s on tape 1 until the first 1. At the same time, copy the 0s onto tape 2.
3. Scan across the 1s on tape 1 until the end of the input. For each 1 read on tape 1, cross off a 0 on tape 2. If all 0s are crossed off before all the 1s are read, *reject*.
4. If all the 0s have now been crossed off, *accept*. If any 0s remain, *reject*."

This machine is simple to analyze. Each of the four stages uses $O(n)$ steps, so the total running time is $O(n)$ and thus is linear. Note that this running time is the best possible because n steps are necessary just to read the input.

Let's summarize what we have shown about the time complexity of A, the amount of time required for deciding A. We produced a single-tape TM M_1 that decides A in $O(n^2)$ time and a faster single tape TM M_2 that decides A in $O(n \log n)$ time. The solution to Problem 7.47 implies that no single-tape TM can do it more quickly. Then we exhibited a two-tape TM M_3 that decides A in $O(n)$ time. Hence the time complexity of A on a single-tape TM is $O(n \log n)$ and on a two-tape TM it is $O(n)$. Note that the complexity of A depends on the model of computation selected.

This discussion highlights an important difference between complexity theory and computability theory. In computability theory, the Church–Turing thesis implies that all reasonable models of computation are equivalent—that is, they all decide the same class of languages. In complexity theory, the choice of model affects the time complexity of languages. Languages that are decidable in, say, linear time on one model aren't necessarily decidable in linear time on another.

In complexity theory, we classify computational problems according to their time complexity. But with which model do we measure time? The same language may have different time requirements on different models.

Fortunately, time requirements don't differ greatly for typical deterministic models. So, if our classification system isn't very sensitive to relatively small differences in complexity, the choice of deterministic model isn't crucial. We discuss this idea further in the next several sections.

COMPLEXITY RELATIONSHIPS AMONG MODELS

Here we examine how the choice of computational model can affect the time complexity of languages. We consider three models: the single-tape Turing machine; the multitape Turing machine; and the nondeterministic Turing machine.

THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

PROOF IDEA The idea behind the proof of this theorem is quite simple. Recall that in Theorem 3.13 we showed how to convert any multitape TM into a single-tape TM that simulates it. Now we analyze that simulation to determine how much additional time it requires. We show that simulating each step of the multitape machine uses at most $O(t(n))$ steps on the single-tape machine. Hence the total time used is $O(t^2(n))$ steps.

PROOF Let M be a k-tape TM that runs in $t(n)$ time. We construct a single-tape TM S that runs in $O(t^2(n))$ time.

Machine S operates by simulating M, as described in Theorem 3.13. To review that simulation, we recall that S uses its single tape to represent the contents on all k of M's tapes. The tapes are stored consecutively, with the positions of M's heads marked on the appropriate squares.

Initially, S puts its tape into the format that represents all the tapes of M and then simulates M's steps. To simulate one step, S scans all the information stored on its tape to determine the symbols under M's tape heads. Then S makes another pass over its tape to update the tape contents and head positions. If one of M's heads moves rightward onto the previously unread portion of its tape, S must increase the amount of space allocated to this tape. It does so by shifting a portion of its own tape one cell to the right.

Now we analyze this simulation. For each step of M, machine S makes two passes over the active portion of its tape. The first obtains the information necessary to determine the next move and the second carries it out. The length of the active portion of S's tape determines how long S takes to scan it, so we must determine an upper bound on this length. To do so we take the sum of the lengths of the active portions of M's k tapes. Each of these active portions has length at most $t(n)$ because M uses $t(n)$ tape cells in $t(n)$ steps if the head moves rightward at every step and even fewer if a head ever moves leftward. Thus a scan of the active portion of S's tape uses $O(t(n))$ steps.

To simulate each of M's steps, S performs two scans and possibly up to k rightward shifts. Each uses $O(t(n))$ time, so the total time for S to simulate one of M's steps is $O(t(n))$.

Now we bound the total time used by the simulation. The initial stage, where S puts its tape into the proper format, uses $O(n)$ steps. Afterward, S simulates each of the $t(n)$ steps of M, using $O(t(n))$ steps, so this part of the simulation uses $t(n) \times O(t(n)) = O(t^2(n))$ steps. Therefore the entire simulation of M uses

$O(n) + O(t^2(n))$ steps.

We have assumed that $t(n) \geq n$ (a reasonable assumption because M could not even read the entire input in less time). Therefore the running time of S is $O(t^2(n))$ and the proof is complete.

Next, we consider the analogous theorem for nondeterministic single-tape Turing machines. We show that any language that is decidable on such a machine is decidable on a deterministic single-tape Turing machine that requires significantly more time. Before doing so, we must define the running time of a nondeterministic Turing machine. Recall that a nondeterministic Turing machine is a decider if all its computation branches halt on all inputs.

DEFINITION 7.9

Let N be a nondeterministic Turing machine that is a decider. The ***running time*** of N is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation on any input of length n, as shown in the following figure.

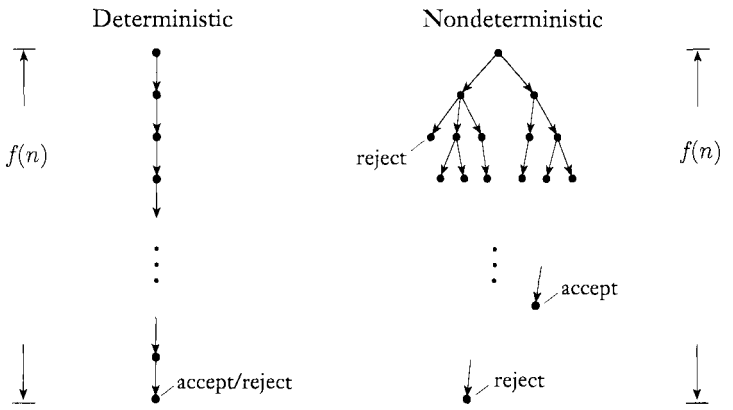

FIGURE 7.10
Measuring deterministic and nondeterministic time

The definition of the running time of a nondeterministic Turing machine is not intended to correspond to any real-world computing device. Rather, it is a useful mathematical definition that assists in characterizing the complexity of an important class of computational problems, as we demonstrate shortly.

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

PROOF Let N be a nondeterministic TM running in $t(n)$ time. We construct a deterministic TM D that simulates N as in the proof of Theorem 3.16 by searching N's nondeterministic computation tree. Now we analyze that simulation.

On an input of length n, every branch of N's nondeterministic computation tree has a length of at most $t(n)$. Every node in the tree can have at most b children, where b is the maximum number of legal choices given by N's transition function. Thus the total number of leaves in the tree is at most $b^{t(n)}$.

The simulation proceeds by exploring this tree breadth first. In other words, it visits all nodes at depth d before going on to any of the nodes at depth $d + 1$. The algorithm given in the proof of Theorem 3.16 inefficiently starts at the root and travels down to a node whenever it visits that node, but eliminating this inefficiency doesn't alter the statement of the current theorem, so we leave it as is. The total number of nodes in the tree is less than twice the maximum number of leaves, so we bound it by $O(b^{t(n)})$. The time for starting from the root and traveling down to a node is $O(t(n))$. Therefore the running time of D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$.

As described in Theorem 3.16, the TM D has three tapes. Converting to a single-tape TM at most squares the running time, by Theorem 7.8. Thus the running time of the single-tape simulator is $(2^{O(t(n))})^2 = 2^{O(2t(n))} = 2^{O(t(n))}$, and the theorem is proved.

7.2

THE CLASS P

Theorems 7.8 and 7.11 illustrate an important distinction. On the one hand, we demonstrated at most a square or *polynomial* difference between the time complexity of problems measured on deterministic single-tape and multitape Turing machines. On the other hand, we showed at most an *exponential* difference between the time complexity of problems on deterministic and nondeterministic Turing machines.

POLYNOMIAL TIME

For our purposes, polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large. Let's look at

why we chose to make this separation between polynomials and exponentials rather than between some other classes of functions.

First, note the dramatic difference between the growth rate of typically occurring polynomials such as n^3 and typically occurring exponentials such as 2^n. For example, let n be 1000, the size of a reasonable input to an algorithm. In that case, n^3 is 1 billion, a large, but manageable number, whereas 2^n is a number much larger than the number of atoms in the universe. Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful.

Exponential time algorithms typically arise when we solve problems by exhaustively searching through a space of solutions, called ***brute-force search***. For example, one way to factor a number into its constituent primes is to search through all potential divisors. The size of the search space is exponential, so this search uses exponential time. Sometimes, brute-force search may be avoided through a deeper understanding of a problem, which may reveal a polynomial time algorithm of greater utility.

All reasonable deterministic computational models are ***polynomially equivalent***. That is, any one of them can simulate another with only a polynomial increase in running time. When we say that all reasonable deterministic models are polynomially equivalent, we do not attempt to define *reasonable*. However, we have in mind a notion broad enough to include models that closely approximate running times on actual computers. For example, Theorem 7.8 shows that the deterministic single-tape and multitape Turing machine models are polynomially equivalent.

From here on we focus on aspects of time complexity theory that are unaffected by polynomial differences in running time. We consider such differences to be insignificant and ignore them. Doing so allows us to develop the theory in a way that doesn't depend on the selection of a particular model of computation. Remember, our aim is to present the fundamental properties of *computation*, rather than properties of Turing machines or any other special model.

You may feel that disregarding polynomial differences in running time is absurd. Real programmers certainly care about such differences and work hard just to make their programs run twice as quickly. However, we disregarded constant factors a while back when we introduced asymptotic notation. Now we propose to disregard the much greater polynomial differences, such as that between time n and time n^3.

Our decision to disregard polynomial differences doesn't imply that we consider such differences unimportant. On the contrary, we certainly do consider the difference between time n and time n^3 to be an important one. But some questions, such as the polynomiality or nonpolynomiality of the factoring problem, do not depend on polynomial differences and are important, too. We merely choose to focus on this type of question here. Ignoring the trees to see the forest doesn't mean that one is more important than the other—it just gives a different perspective.

Now we come to an important definition in complexity theory.

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

The class P plays a central role in our theory and is important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
2. P roughly corresponds to the class of problems that are realistically solvable on a computer.

Item 1 indicates that P is a mathematically robust class. It isn't affected by the particulars of the model of computation that we are using.

Item 2 indicates that P is relevant from a practical standpoint. When a problem is in P, we have a method of solving it that runs in time n^k for some constant k. Whether this running time is practical depends on k and on the application. Of course, a running time of n^{100} is unlikely to be of any practical use. Nevertheless, calling polynomial time the threshold of practical solvability has proven to be useful. Once a polynomial time algorithm has been found for a problem that formerly appeared to require exponential time, some key insight into it has been gained, and further reductions in its complexity usually follow, often to the point of actual practical utility.

EXAMPLES OF PROBLEMS IN P

When we present a polynomial time algorithm, we give a high-level description of it without reference to features of a particular computational model. Doing so avoids tedious details of tapes and head motions. We need to follow certain conventions when describing an algorithm so that we can analyze it for polynomiality.

We describe algorithms with numbered stages. The notion of a stage of an algorithm is analogous to a step of a Turing machine, though of course, implementing one stage of an algorithm on a Turing machine, in general, will require many Turing machine steps.

When we analyze an algorithm to show that it runs in polynomial time, we need to do two things. First, we have to give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on an input of length n. Then, we have to examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model. We choose the stages when we describe the algorithm to make this second part of the analysis easy to

do. When both tasks have been completed, we can conclude that the algorithm runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials is a polynomial.

One point that requires attention is the encoding method used for problems. We continue to use the angle-bracket notation $\langle \cdot \rangle$ to indicate a reasonable encoding of one or more objects into a string, without specifying any particular encoding method. Now, a reasonable method is one that allows for polynomial time encoding and decoding of objects into natural internal representations or into other reasonable encodings. Familiar encoding methods for graphs, automata, and the like all are reasonable. But note that unary notation for encoding numbers (as in the number 17 encoded by the unary string 11111111111111111) isn't reasonable because it is exponentially larger than truly reasonable encodings, such as base k notation for any $k \geq 2$.

Many computational problems you encounter in this chapter contain encodings of graphs. One reasonable encoding of a graph is a list of its nodes and edges. Another is the ***adjacency matrix***, where the (i, j)th entry is 1 if there is an edge from node i to node j and 0 if not. When we analyze algorithms on graphs, the running time may be computed in terms of the number of nodes instead of the size of the graph representation. In reasonable graph representations, the size of the representation is a polynomial in the number of nodes. Thus, if we analyze an algorithm and show that its running time is polynomial (or exponential) in the number of nodes, we know that it is polynomial (or exponential) in the size of the input.

The first problem concerns directed graphs. A directed graph G contains nodes s and t, as shown in the following figure. The *PATH* problem is to determine whether a directed path exists from s to t. Let

$$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

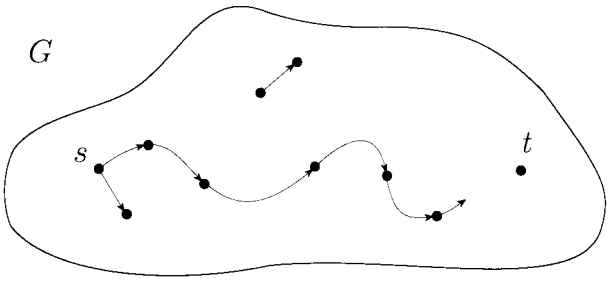

FIGURE 7.13
The *PATH* problem: Is there a path from s to t?

THEOREM 7.14

PATH $\in$ P.

PROOF IDEA We prove this theorem by presenting a polynomial time algorithm that decides *PATH*. Before describing that algorithm, let's observe that a brute-force algorithm for this problem isn't fast enough.

A brute-force algorithm for *PATH* proceeds by examining all potential paths in G and determining whether any is a directed path from s to t. A potential path is a sequence of nodes in G having a length of at most m, where m is the number of nodes in G. (If any directed path exists from s to t, one having a length of at most m exists because repeating a node never is necessary.) But the number of such potential paths is roughly m^m, which is exponential in the number of nodes in G. Therefore this brute-force algorithm uses exponential time.

To get a polynomial time algorithm for *PATH* we must do something that avoids brute force. One way is to use a graph-searching method such as breadth-first search. Here, we successively mark all nodes in G that are reachable from s by directed paths of length 1, then 2, then 3, through m. Bounding the running time of this strategy by a polynomial is easy.

PROOF A polynomial time algorithm M for *PATH* operates as follows.

M = "On input $\langle G, s, t\rangle$ where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

Now we analyze this algorithm to show that it runs in polynomial time. Obviously, stages 1 and 4 are executed only once. Stage 3 runs at most m times because each time except the last it marks an additional node in G. Thus the total number of stages used is at most $1 + 1 + m$, giving a polynomial in the size of G.

Stages 1 and 4 of M are easily implemented in polynomial time on any reasonable deterministic model. Stage 3 involves a scan of the input and a test of whether certain nodes are marked, which also is easily implemented in polynomial time. Hence M is a polynomial time algorithm for *PATH*.

Let's turn to another example of a polynomial time algorithm. Say that two numbers are ***relatively prime*** if 1 is the largest integer that evenly divides them both. For example, 10 and 21 are relatively prime, even though neither of them is a prime number by itself, whereas 10 and 22 are not relatively prime because both are divisible by 2. Let *RELPRIME* be the problem of testing whether two

numbers are relatively prime. Thus

$$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}.$$

THEOREM 7.15

$RELPRIME \in \mathrm{P}$.

PROOF IDEA One algorithm that solves this problem searches through all possible divisors of both numbers and accepts if none are greater than 1. However, the magnitude of a number represented in binary, or in any other base k notation for $k \geq 2$, is exponential in the length of its representation. Therefore this brute-force algorithm searches through an exponential number of potential divisors and has an exponential running time.

Instead, we solve this problem with an ancient numerical procedure, called the ***Euclidean algorithm***, for computing the greatest common divisor. The ***greatest common divisor*** of natural numbers x and y, written $\gcd(x, y)$, is the largest integer that evenly divides both x and y. For example, $\gcd(18, 24) = 6$. Obviously, x and y are relatively prime iff $\gcd(x, y) = 1$. We describe the Euclidean algorithm as algorithm E in the proof. It uses the mod function, where $x \bmod y$ is the remainder after the integer division of x by y.

PROOF The Euclidean algorithm E is as follows.

E = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

Algorithm R solves *RELPRIME*, using E as a subroutine.

R = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Run E on $\langle x, y\rangle$.
2. If the result is 1, *accept*. Otherwise, *reject*."

Clearly, if E runs correctly in polynomial time, so does R and hence we only need to analyze E for time and correctness. The correctness of this algorithm is well known so we won't discuss it further here.

To analyze the time complexity of E, we first show that every execution of stage 2 (except possibly the first), cuts the value of x by at least half. After stage 2 is executed, $x < y$ because of the nature of the mod function. After stage 3, $x > y$ because the two have been exchanged. Thus, when stage 2 is subsequently executed, $x > y$. If $x/2 \geq y$, then $x \bmod y < y \leq x/2$ and x drops by at least half. If $x/2 < y$, then $x \bmod y = x - y < x/2$ and x drops by at least half.

The values of x and y are exchanged every time stage 3 is executed, so each of the original values of x and y are reduced by at least half every other time through the loop. Thus the maximum number of times that stages 2 and 3 are executed is the lesser of $2 \log_2 x$ and $2 \log_2 y$. These logarithms are proportional to the lengths of the representations, giving the number of stages executed as $O(n)$. Each stage of E uses only polynomial time, so the total running time is polynomial.

The final example of a polynomial time algorithm shows that every context-free language is decidable in polynomial time.

THEOREM 7.16

Every context-free language is a member of P.

PROOF IDEA In Theorem 4.9 we proved that every CFL is decidable. To do so we gave an algorithm for each CFL that decides it. If that algorithm runs in polynomial time, the current theorem follows as a corollary. Let's recall that algorithm and find out whether it runs quickly enough.

Let L be a CFL generated by CFG G that is in Chomsky normal form. From Problem 2.26, any derivation of a string w has $2n - 1$ steps, where n is the length of w because G is in Chomsky normal form. The decider for L works by trying all possible derivations with $2n - 1$ steps when its input is a string of length n. If any of these is a derivation of w, the decider accepts; if not, it rejects.

A quick analysis of this algorithm shows that it doesn't run in polynomial time. The number of derivations with k steps may be exponential in k, so this algorithm may require exponential time.

To get a polynomial time algorithm we introduce a powerful technique called ***dynamic programming***. This technique uses the accumulation of information about smaller subproblems to solve larger problems. We record the solution to any subproblem so that we need to solve it only once. We do so by making a table of all subproblems and entering their solutions systematically as we find them.

In this case, we consider the subproblems of determining whether each variable in G generates each substring of w. The algorithm enters the solution to this subproblem in an $n \times n$ table. For $i \leq j$ the (i, j)th entry of the table contains the collection of variables that generate the substring $w_i w_{i+1} \cdots w_j$. For $i > j$ the table entries are unused.

The algorithm fills in the table entries for each substring of w. First it fills in the entries for the substrings of length 1, then those of length 2, and so on. It uses the entries for the shorter lengths to assist in determining the entries for the longer lengths.

For example, suppose that the algorithm has already determined which variables generate all substrings up to length k. To determine whether a variable A generates a particular substring of length $k+1$ the algorithm splits that substring into two nonempty pieces in the k possible ways. For each split, the algorithm examines each rule $A \rightarrow BC$ to determine whether B generates the first piece and C generates the second piece, using table entries previously computed. If both B and C generate the respective pieces, A generates the substring and so is added to the associated table entry. The algorithm starts the process with the strings of length 1 by examining the table for the rules $A \rightarrow \mathtt{b}$.

PROOF The following algorithm D implements the proof idea. Let G be a CFG in Chomsky normal form generating the CFL L. Assume that S is the start variable. (Recall that the empty string is handled specially in a Chomsky normal form grammar. The algorithm handles the special case in which $w = \varepsilon$ in stage 1.) Comments appear inside double brackets.

D = "On input $w = w_1 \cdots w_n$:

1. If $w = \varepsilon$ and $S \rightarrow \varepsilon$ is a rule, *accept*. [[handle $w = \varepsilon$ case]]
2. For $i = 1$ to n: [[examine each substring of length 1]]
3. For each variable A:
4. Test whether $A \rightarrow \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: [[i is the start position of the substring]]
8. Let $j = i + l - 1$, [[j is the end position of the substring]]
9. For $k = i$ to $j - 1$: [[k is the split position]]
10. For each rule $A \rightarrow BC$:
11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*. Otherwise, *reject*."

Now we analyze D. Each stage is easily implemented to run in polynomial time. Stages 4 and 5 run at most nv times, where v is the number of variables in G and is a fixed constant independent of n; hence these stages run $O(n)$ times. Stage 6 runs at most n times. Each time stage 6 runs, stage 7 runs at most n times. Each time stage 7 runs, stages 8 and 9 run at most n times. Each time stage 9 runs, stage 10 runs r times, where r is the number of rules of G and is another fixed constant. Thus stage 11, the inner loop of the algorithm, runs $O(n^3)$ times. Summing the total shows that D executes $O(n^3)$ stages.

7.3

THE CLASS NP

As we observed in Section 7.2, we can avoid brute-force search in many problems and obtain polynomial time solutions. However, attempts to avoid brute force in certain other problems, including many interesting and useful ones, haven't been successful, and polynomial time algorithms that solve them aren't known to exist.

Why have we been unsuccessful in finding polynomial time algorithms for these problems? We don't know the answer to this important question. Perhaps these problems have, as yet undiscovered, polynomial time algorithms that rest on unknown principles. Or possibly some of these problems simply *cannot* be solved in polynomial time. They may be intrinsically difficult.

One remarkable discovery concerning this question shows that the complexities of many problems are linked. A polynomial time algorithm for one such problem can be used to solve an entire class of problems. To understand this phenomenon, let's begin with an example.

A ***Hamiltonian path*** in a directed graph G is a directed path that goes through each node exactly once. We consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes, as shown in the following figure. Let

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}.$$

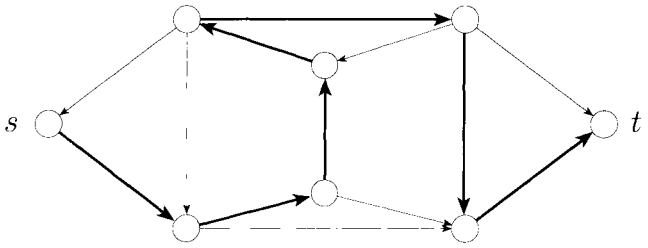

FIGURE **7.17**
A Hamiltonian path goes through every node exactly once

We can easily obtain an exponential time algorithm for the *HAMPATH* problem by modifying the brute-force algorithm for *PATH* given in Theorem 7.14. We need only add a check to verify that the potential path is Hamiltonian. No one knows whether *HAMPATH* is solvable in polynomial time.

The *HAMPATH* problem does have a feature called ***polynomial verifiabil-***

ity that is important for understanding its complexity. Even though we don't know of a fast (i.e., polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using the exponential time algorithm), we could easily convince someone else of its existence, simply by presenting it. In other words, *verifying* the existence of a Hamiltonian path may be much easier than *determining* its existence.

Another polynomially verifiable problem is compositeness. Recall that a natural number is ***composite*** if it is the product of two integers greater than 1 (i.e., a composite number is one that is not a prime number). Let

$$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}.$$

We can easily verify that a number is composite—all that is needed is a divisor of that number. Recently, a polynomial time algorithm for testing whether a number is prime or composite was discovered, but it is considerably more complicated than the preceding method for verifying compositeness.

Some problems may not be polynomially verifiable. For example, take $\overline{HAMPATH}$, the complement of the *HAMPATH* problem. Even if we could determine (somehow) that a graph did *not* have a Hamiltonian path, we don't know of a way for someone else to verify its nonexistence without using the same exponential time algorithm for making the determination in the first place. A formal definition follows.

DEFINITION 7.18

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

A verifier uses additional information, represented by the symbol c in Definition 7.18, to verify that a string w is a member of A. This information is called a ***certificate***, or ***proof***, of membership in A. Observe that, for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound. Let's apply this definition to the languages *HAMPATH* and *COMPOSITES*.

For the *HAMPATH* problem, a certificate for a string $\langle G, s, t \rangle \in HAMPATH$ simply is the Hamiltonian path from s to t. For the *COMPOSITES* problem, a certificate for the composite number x simply is one of its divisors. In both cases the verifier can check in polynomial time that the input is in the language when it is given the certificate.

DEFINITION 7.19

NP is the class of languages that have polynomial time verifiers.

The class NP is important because it contains many problems of practical interest. From the preceding discussion, both *HAMPATH* and *COMPOSITES* are members of NP. As we mentioned, *COMPOSITES* is also a member of P which is a subset of NP, but proving this stronger result is much more difficult. The term NP comes from ***nondeterministic polynomial time*** and is derived from an alternative characterization by using nondeterministic polynomial time Turing machines. Problems in NP are sometimes called NP-problems.

The following is a nondeterministic Turing machine (NTM) that decides the *HAMPATH* problem in nondeterministic polynomial time. Recall that in Definition 7.9 we defined the time of a nondeterministic machine to be the time used by the longest computation branch.

N_1 = "On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Write a list of m numbers, $p_1, \ldots, p_m$, where m is the number of nodes in G. Each number in the list is nondeterministically selected to be between 1 and m.
2. Check for repetitions in the list. If any are found, *reject*.
3. Check whether $s = p_1$ and $t = p_m$. If either fail, *reject*.
4. For each i between 1 and $m - 1$, check whether (p_i, p_{i+1}) is an edge of G. If any are not, *reject*. Otherwise, all tests have been passed, so *accept*."

To analyze this algorithm and verify that it runs in nondeterministic polynomial time, we examine each of its stages. In stage 1, the nondeterministic selection clearly runs in polynomial time. In stages 2 and 3, each part is a simple check, so together they run in polynomial time. Finally, stage 4 also clearly runs in polynomial time. Thus this algorithm runs in nondeterministic polynomial time.

THEOREM 7.20

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

PROOF IDEA We show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa. The NTM simulates the verifier by guessing the certificate. The verifier simulates the NTM by using the accepting branch as the certificate.

PROOF For the forward direction of this theorem, let $A \in$ NP and show that A is decided by a polynomial time NTM N. Let V be the polynomial time verifier for A that exists by the definition of NP. Assume that V is a TM that runs in time n^k and construct N as follows.

N = "On input w of length n:

1. Nondeterministically select string c of length at most n^k.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts, *accept*; otherwise, *reject*."

To prove the other direction of the theorem, assume that A is decided by a polynomial time NTM N and construct a polynomial time verifier V as follows.

V = "On input $\langle w, c \rangle$, where w and c are strings:

1. Simulate N on input w, treating each symbol of c as a description of the nondeterministic choice to make at each step (as in the proof of Theorem 3.16).
2. If this branch of N's computation accepts, *accept*; otherwise, *reject*."

We define the nondeterministic time complexity class $\text{NTIME}(t(n))$ as analogous to the deterministic time complexity class $\text{TIME}(t(n))$.

DEFINITION 7.21

NTIME*(t(n))* $= \{L \mid L$ is a language decided by a $O(t(n))$ time nondeterministic Turing machine$\}$.

COROLLARY 7.22

$\text{NP} = \bigcup_k \text{NTIME}(n^k)$.

The class NP is insensitive to the choice of reasonable nondeterministic computational model because all such models are polynomially equivalent. When describing and analyzing nondeterministic polynomial time algorithms, we follow the preceding conventions for deterministic polynomial time algorithms. Each stage of a nondeterministic polynomial time algorithm must have an obvious implementation in nondeterministic polynomial time on a reasonable nondeterministic computational model. We analyze the algorithm to show that every branch uses at most polynomially many stages.

EXAMPLES OF PROBLEMS IN NP

A ***clique*** in an undirected graph is a subgraph, wherein every two nodes are connected by an edge. A ***k-clique*** is a clique that contains k nodes. Figure 7.23 illustrates a graph having a 5-clique

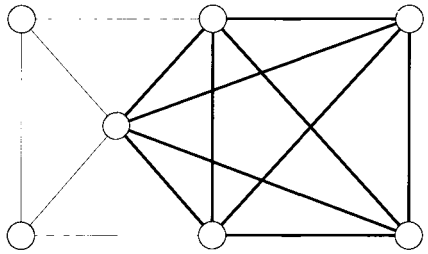

FIGURE 7.23
A graph with a 5-clique

The clique problem is to determine whether a graph contains a clique of a specified size. Let

$$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}.$$

THEOREM 7.24

CLIQUE is in NP.

PROOF IDEA The clique is the certificate.

PROOF The following is a verifier V for *CLIQUE*.

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a set of k nodes in G
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

ALTERNATIVE PROOF If you prefer to think of NP in terms of nondeterministic polynomial time Turing machines, you may prove this theorem by giving one that decides *CLIQUE*. Observe the similarity between the two proofs.

N = "On input $\langle G, k\rangle$, where G is a graph:

1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*."

Next we consider the *SUBSET-SUM* problem concerning integer arithmetic. In this problem we have a collection of numbers $x_1, \ldots, x_k$ and a target number t. We want to determine whether the collection contains a subcollection that

adds up to t. Thus

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}.$$

For example, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in SUBSET\text{-}SUM$ because $4 + 21 = 25$. Note that $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_l\}$ are considered to be ***multisets*** and so allow repetition of elements.

THEOREM 7.25

SUBSET-SUM is in NP.

PROOF IDEA The subset is the certificate.

PROOF The following is a verifier V for *SUBSET-SUM*.

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, *accept*; otherwise, *reject*."

ALTERNATIVE PROOF We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for *SUBSET-SUM* as follows.

N = "On input $\langle S, t\rangle$:

1. Nondeterministically select a subset c of the numbers in S.
2. Test whether c is a collection of numbers that sum to t.
3. If the test passes, *accept*; otherwise, *reject*."

Observe that the complements of these sets, $\overline{CLIQUE}$ and $\overline{SUBSET\text{-}SUM}$, are not obviously members of NP. Verifying that something is *not* present seems to be more difficult than verifying that it *is* present. We make a separate complexity class, called coNP, which contains the languages that are complements of languages in NP. We don't know whether coNP is different from NP.

THE P VERSUS NP QUESTION

As we have been saying, NP is the class of languages that are solvable in polynomial time on a nondeterministic Turing machine, or, equivalently, it is the class of languages whereby membership in the language can be verified in polynomial time. P is the class of languages where membership can be tested in polynomial time. We summarize this information as follows, where we loosely refer to

polynomial time solvable as solvable "quickly."

P = the class of languages for which membership can be *decided* quickly.
NP = the class of languages for which membership can be *verified* quickly.

We have presented examples of languages, such as *HAMPATH* and *CLIQUE*, that are members of NP but that are not known to be in P. The power of polynomial verifiability seems to be much greater than that of polynomial decidability. But, hard as it may be to imagine, P and NP could be equal. We are unable to *prove* the existence of a single language in NP that is not in P.

The question of whether P = NP is one of the greatest unsolved problems in theoretical computer science and contemporary mathematics. If these classes were equal, any polynomially verifiable problem would be polynomially decidable. Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in NP, without success. Researchers also have tried proving that the classes are unequal, but that would entail showing that no fast algorithm exists to replace brute-force search. Doing so is presently beyond scientific reach. The following figure shows the two possibilities.

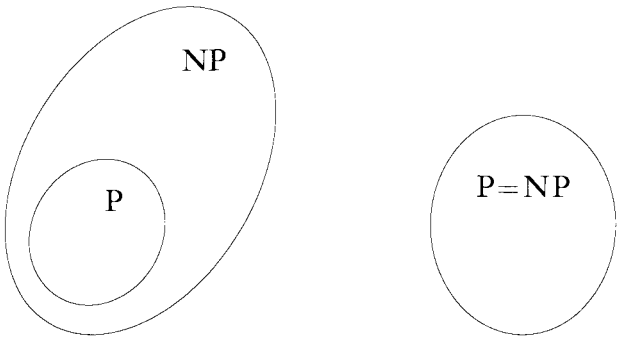

FIGURE 7.26
One of these two possibilities is correct

The best method known for solving languages in NP deterministically uses exponential time. In other words, we can prove that

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{TIME}(2^{n^k}),$$

but we don't know whether NP is contained in a smaller deterministic time complexity class.

7.4

NP-COMPLETENESS

One important advance on the P versus NP question came in the early 1970s with the work of Stephen Cook and Leonid Levin. They discovered certain problems in NP whose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called ***NP-complete***. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

On the theoretical side, a researcher trying to show that P is unequal to NP may focus on an NP-complete problem. If any problem in NP requires more than polynomial time, an NP-complete one does. Furthermore, a researcher attempting to prove that P equals NP only needs to find a polynomial time algorithm for an NP-complete problem to achieve this goal.

On the practical side, the phenomenon of NP-completeness may prevent wasting time searching for a nonexistent polynomial time algorithm to solve a particular problem. Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, we believe that P is unequal to NP, so proving that a problem is NP-complete is strong evidence of its nonpolynomiality.

The first NP-complete problem that we present is called the ***satisfiability problem***. Recall that variables that can take on the values TRUE and FALSE are called ***Boolean variables*** (see Section 0.2). Usually, we represent TRUE by 1 and FALSE by 0. The ***Boolean operations*** AND, OR, and NOT, represented by the symbols $\wedge$, $\vee$, and $\neg$, respectively, are described in the following list. We use the overbar as a shorthand for the $\neg$ symbol, so $\overline{x}$ means $\neg\, x$.

$$\begin{array}{lll} 0 \wedge 0 = 0 & 0 \vee 0 = 0 & \overline{0} = 1 \\ 0 \wedge 1 = 0 & 0 \vee 1 = 1 & \overline{1} = 0 \\ 1 \wedge 0 = 0 & 1 \vee 0 = 1 & \\ 1 \wedge 1 = 1 & 1 \vee 1 = 1 & \end{array}$$

A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

Now we state the Cook–Levin theorem, which links the complexity of the *SAT* problem to the complexities of all problems in NP.

THEOREM 7.27

Cook–Levin theorem $SAT \in \mathrm{P}$ iff $\mathrm{P} = \mathrm{NP}$.

Next, we develop the method that is central to the proof of the Cook–Levin theorem.

POLYNOMIAL TIME REDUCIBILITY

In Chapter 5 we defined the concept of reducing one problem to another. When problem A reduces to problem B, a solution to B can be used to solve A. Now we define a version of reducibility that takes the efficiency of computation into account. When problem A is *efficiently* reducible to problem B, an efficient solution to B can be used to solve A efficiently.

DEFINITION 7.28

A function $f : \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

DEFINITION 7.29

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_{\mathrm{P}} B$, if a polynomial time computable function $f : \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.

Polynomial time reducibility is the efficient analog to mapping reducibility as defined in Section 5.3. Other forms of efficient reducibility are available, but polynomial time reducibility is a simple form that is adequate for our purposes so we won't discuss the others here. The following figure illustrates polynomial time reducibility.

[1] It is called ***polynomial time many–one reducibility*** in some other textbooks.

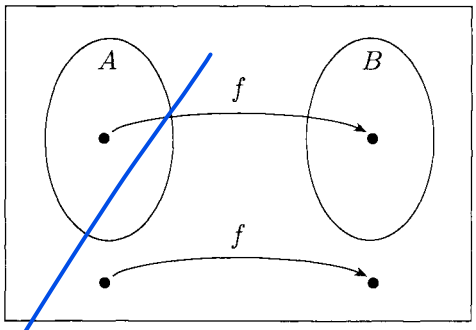

FIGURE 7.30
Polynomial time function f reducing A to B

As with an ordinary mapping reduction, a polynomial time reduction of A to B provides a way to convert membership testing in A to membership testing in B, but now the conversion is done efficiently. To test whether $w \in A$, we use the reduction f to map w to $f(w)$ and test whether $f(w) \in B$.

If one language is polynomial time reducible to a language already known to have a polynomial time solution, we obtain a polynomial time solution to the original language, as in the following theorem.

THEOREM 7.31

If $A \leq_P B$ and $B \in P$, then $A \in P$.

PROOF Let M be the polynomial time algorithm deciding B and f be the polynomial time reduction from A to B. We describe a polynomial time algorithm N deciding A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

We have $w \in A$ whenever $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Moreover, N runs in polynomial time because each of its two stages runs in polynomial time. Note that stage 2 runs in polynomial time because the composition of two polynomials is a polynomial.

Before demonstrating a polynomial time reduction we introduce *3SAT*, a special case of the satisfiability problem whereby all formulas are in a special form. A

literal is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$. A ***clause*** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in ***conjunctive normal form***, called a ***cnf-formula***, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

It is a ***3cnf-formula*** if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

Let $3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$. In a satisfiable cnf-formula, each clause must contain at least one literal that is assigned 1.

The following theorem presents a polynomial time reduction from the *3SAT* problem to the *CLIQUE* problem.

THEOREM 7.32

3SAT is polynomial time reducible to *CLIQUE*.

PROOF IDEA The polynomial time reduction f that we demonstrate from *3SAT* to *CLIQUE* converts formulas to graphs. In the constructed graphs, cliques of a specified size correspond to satisfying assignments of the formula. Structures within the graph are designed to mimic the behavior of the variables and clauses.

PROOF Let ϕ be a formula with k clauses such as

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

The reduction f generates the string $\langle G, k\rangle$, where G is an undirected graph defined as follows.

The nodes in G are organized into k groups of three nodes each called the ***triples***, $t_1, \ldots, t_k$. Each triple corresponds to one of the clauses in ϕ, and each node in a triple corresponds to a literal in the associated clause. Label each node of G with its corresponding literal in ϕ.

The edges of G connect all but two types of pairs of nodes in G. No edge is present between nodes in the same triple and no edge is present between two nodes with contradictory labels, as in x_2 and $\overline{x_2}$. The following figure illustrates this construction when $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

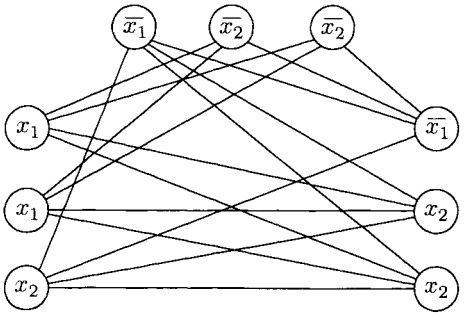

FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Now we demonstrate why this construction works. We show that ϕ is satisfiable iff G has a k-clique.

Suppose that ϕ has a satisfying assignment. In that satisfying assignment, at least one literal is true in every clause. In each triple of G, we select one node corresponding to a true literal in the satisfying assignment. If more than one literal is true in a particular clause, we choose one of the true literals arbitrarily. The nodes just selected form a k-clique. The number of nodes selected is k, because we chose one for each of the k triples. Each pair of selected nodes is joined by an edge because no pair fits one of the exceptions described previously. They could not be from the same triple because we selected only one node per triple. They could not have contradictory labels because the associated literals were both true in the satisfying assignment. Therefore G contains a k-clique.

Suppose that G has a k-clique. No two of the clique's nodes occur in the same triple because nodes in the same triple aren't connected by edges. Therefore each of the k triples contains exactly one of the k clique nodes. We assign truth values to the variables of ϕ so that each literal labeling a clique node is made true. Doing so is always possible because two nodes labeled in a contradictory way are not connected by an edge and hence both can't be in the clique. This assignment to the variables satisfies ϕ because each triple contains a clique node and hence each clause contains a literal that is assigned TRUE. Therefore ϕ is satisfiable.

Theorems 7.31 and 7.32 tell us that, if *CLIQUE* is solvable in polynomial time, so is *3SAT*. At first glance, this connection between these two problems appears quite remarkable because, superficially, they are rather different. But polynomial time reducibility allows us to link their complexities. Now we turn to a definition that will allow us similarly to link the complexities of an entire class of problems.