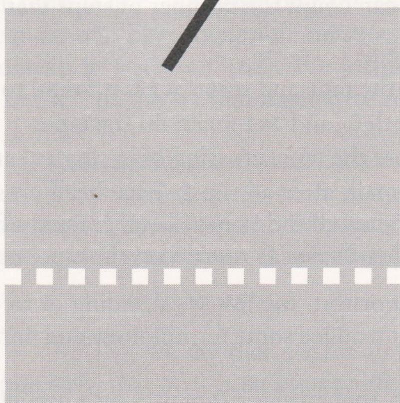
7



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 will *analyze* the algorithm for TM M_1 deciding A to determine how much time it uses. First, we introduce some terminology and notation for this purpose.

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 $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $f(n) = O(g(n))$ 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 $n^c = 2^{c \log_2 n}$, 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 $f, g\colon \mathcal{N} \longrightarrow \mathcal{R}^+$. Say that $f(n) = o(g(n))$ 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 $t\colon \mathcal{N} \longrightarrow \mathcal{R}^+$ 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 $A \in$ 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 7.49 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 tape 1 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 7.49 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.

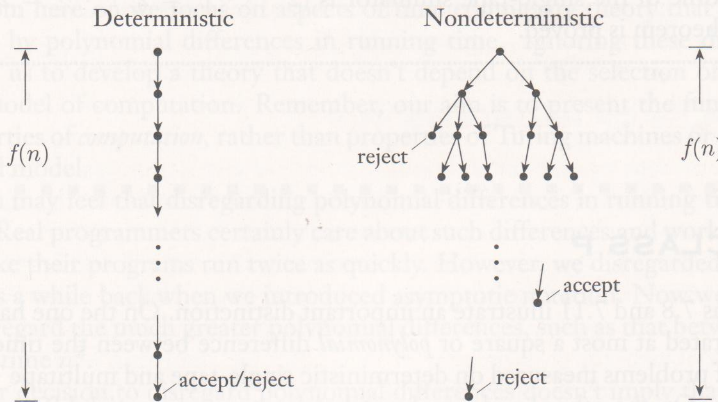


FIGURE 7.10
Measuring deterministic and nondeterministic time

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 it takes to start from the root and travel 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. Ignoring these differences allows us to develop a theory 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,

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

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 follow certain conventions when describing an algorithm so that we can analyze it for polynomiality.

We continue to describe algorithms with numbered stages. Now we must be sensitive to the number of Turing machine steps required to implement each stage, as well as to the total number of stages that the algorithm uses.

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

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

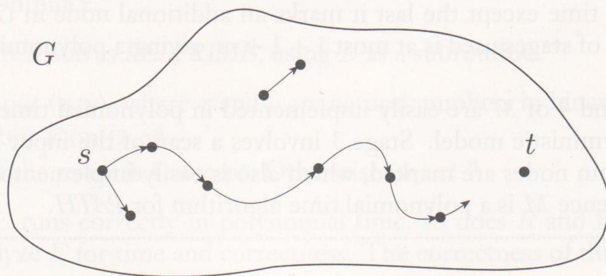


FIGURE 7.13
The *PATH* problem: Is there a path from s to t?

THEOREM 7.14

$PATH \in \mathrm{P}$.

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 $\in$ 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 $A \to BC$ 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 $A \to \mathtt{b}$.

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. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. ⟦ $w = \varepsilon$ case ⟧
2. For $i = 1$ to n: ⟦ examine each substring of length 1 ⟧
3. For each variable A:
4. Test whether $A \to \mathtt{b}$ 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 $A \to BC$:
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*; else, *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

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

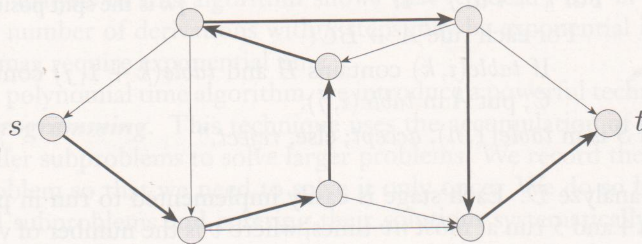


FIGURE 7.17
A Hamiltonian path goes through every node exactly once

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 has a feature called ***polynomial verifiability*** 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 a 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 NTIME$(t(n))$ as analogous to the deterministic time complexity class TIME$(t(n))$.

DEFINITION 7.21

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

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 with a 5-clique.

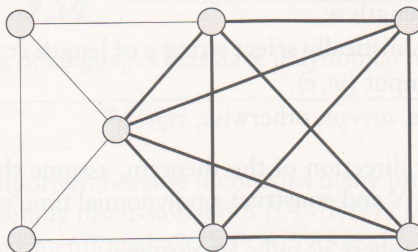


FIGURE 7.23
A graph with a 5-clique

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 subgraph with 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. We are given 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,

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

For example, $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *SUBSET-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.

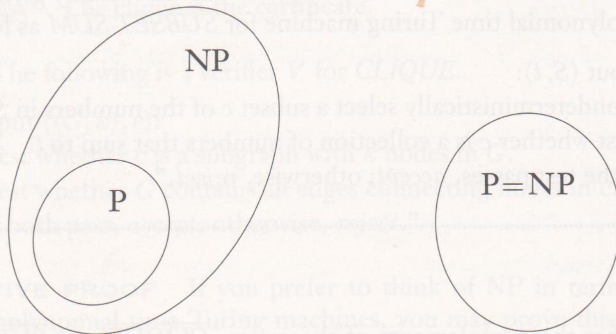


FIGURE 7.26
One of these two possibilities is correct

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

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

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 a theorem that links the complexity of the *SAT* problem to the complexities of all problems in NP.

THEOREM 7.27

$SAT \in P$ iff $P = NP$.

Next, we develop the method that is central to the proof of this 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 $f\colon \Sigma^* \longrightarrow \Sigma^*$ 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 $f\colon \Sigma^* \longrightarrow \Sigma^*$ 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. Figure 7.30 illustrates polynomial time reducibility.

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

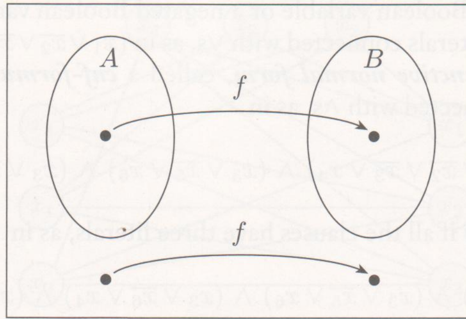


FIGURE 7.30
Polynomial time function f reducing A to B

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 $3SAT = \{\langle\phi\rangle |\ \phi \text{ is a satisfiable 3cnf-formula}\}$. If an assignment satisfies a cnf-formula, each clause must contain at least one literal that evaluates to 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

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

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 $\overline{x_2}$. Figure 7.33 illustrates this construction when $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$.

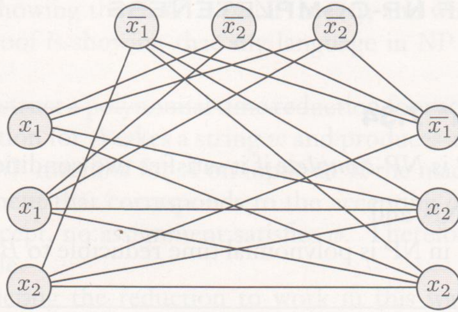


FIGURE 7.33
The graph that the reduction produces from
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

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.

DEFINITION OF NP-COMPLETENESS

DEFINITION 7.34

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

THEOREM 7.35

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

PROOF This theorem follows directly from the definition of polynomial time reducibility.

THEOREM 7.36

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.

PROOF We already know that C is in NP, so we must show that every A in NP is polynomial time reducible to C. Because B is NP-complete, every language in NP is polynomial time reducible to B, and B in turn is polynomial time reducible to C. Polynomial time reductions compose; that is, if A is polynomial time reducible to B and B is polynomial time reducible to C, then A is polynomial time reducible to C. Hence every language in NP is polynomial time reducible to C.

THE COOK–LEVIN THEOREM

Once we have one NP-complete problem, we may obtain others by polynomial time reduction from it. However, establishing the first NP-complete problem is more difficult. Now we do so by proving that *SAT* is NP-complete.

THEOREM 7.37

SAT is NP-complete.[2]

This theorem implies Theorem 7.27.

[2]An alternative proof of this theorem appears in Section 9.3.

PROOF IDEA Showing that *SAT* is in NP is easy, and we do so shortly. The hard part of the proof is showing that any language in NP is polynomial time reducible to *SAT*.

To do so, we construct a polynomial time reduction for each language A in NP to *SAT*. The reduction for A takes a string w and produces a Boolean formula ϕ that simulates the NP machine for A on input w. If the machine accepts, ϕ has a satisfying assignment that corresponds to the accepting computation. If the machine doesn't accept, no assignment satisfies ϕ. Therefore, w is in A if and only if ϕ is satisfiable.

Actually constructing the reduction to work in this way is a conceptually simple task, though we must cope with many details. A Boolean formula may contain the Boolean operations AND, OR, and NOT, and these operations form the basis for the circuitry used in electronic computers. Hence the fact that we can design a Boolean formula to simulate a Turing machine isn't surprising. The details are in the implementation of this idea.

PROOF First, we show that *SAT* is in NP. A nondeterministic polynomial time machine can guess an assignment to a given formula ϕ and accept if the assignment satisfies ϕ.

Next, we take any language A in NP and show that A is polynomial time reducible to *SAT*. Let N be a nondeterministic Turing machine that decides A in n^k time for some constant k. (For convenience, we actually assume that N runs in time $n^k - 3$; but only those readers interested in details should worry about this minor point.) The following notion helps to describe the reduction.

A ***tableau*** for N on w is an $n^k \times n^k$ table whose rows are the configurations of a branch of the computation of N on input w, as shown in the following figure.

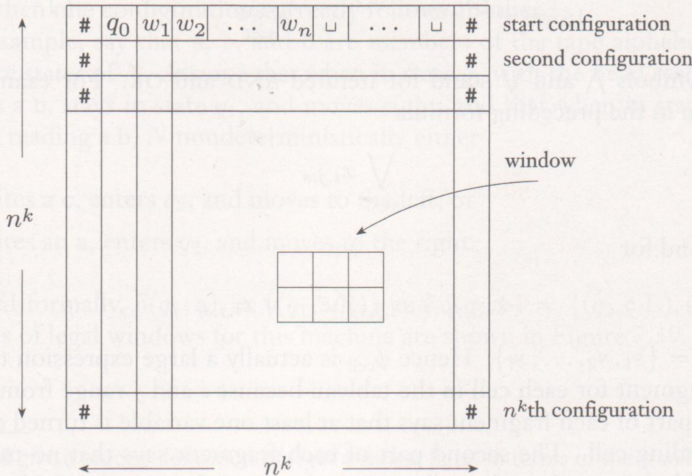


FIGURE 7.38
A tableau is an $n^k \times n^k$ table of configurations

For convenience later, we assume that each configuration starts and ends with a # symbol. Therefore, the first and last columns of a tableau are all #s. The first row of the tableau is the starting configuration of N on w, and each row follows the previous one according to N's transition function. A tableau is ***accepting*** if any row of the tableau is an accepting configuration.

Every accepting tableau for N on w corresponds to an accepting computation branch of N on w. Thus, the problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists.

Now we get to the description of the polynomial time reduction f from A to SAT. On input w, the reduction produces a formula ϕ. We begin by describing the variables of ϕ. Say that Q and Γ are the state set and tape alphabet of N, respectively. Let $C = Q \cup \Gamma \cup \{\#\}$. For each i and j between 1 and n^k and for each s in C, we have a variable, $x_{i,j,s}$.

Each of the $(n^k)^2$ entries of a tableau is called a ***cell***. The cell in row i and column j is called $cell[i,j]$ and contains a symbol from C. We represent the contents of the cells with the variables of ϕ. If $x_{i,j,s}$ takes on the value 1, it means that $cell[i,j]$ contains an s.

Now we design ϕ so that a satisfying assignment to the variables does correspond to an accepting tableau for N on w. The formula ϕ is the AND of four parts: $\phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}}$. We describe each part in turn.

As we mentioned previously, turning variable $x_{i,j,s}$ on corresponds to placing symbol s in $cell[i,j]$. The first thing we must guarantee in order to obtain a correspondence between an assignment and a tableau is that the assignment turns on exactly one variable for each cell. Formula ϕ_{cell} ensures this requirement by expressing it in terms of Boolean operations:

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \ne t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right].$$

The symbols $\bigwedge$ and $\bigvee$ stand for iterated AND and OR. For example, the expression in the preceding formula

$$\bigvee_{s \in C} x_{i,j,s}$$

is shorthand for

$$x_{i,j,s_1} \vee x_{i,j,s_2} \vee \cdots \vee x_{i,j,s_l}$$

where $C = \{s_1, s_2, \ldots, s_l\}$. Hence ϕ_{cell} is actually a large expression that contains a fragment for each cell in the tableau because i and j range from 1 to n^k. The first part of each fragment says that at least one variable is turned on in the corresponding cell. The second part of each fragment says that no more than one variable is turned on (literally, it says that in each pair of variables, at least one is turned off) in the corresponding cell. These fragments are connected by $\wedge$ operations.

The first part of ϕ_{cell} inside the brackets stipulates that at least one variable that is associated with each cell is on, whereas the second part stipulates that no more than one variable is on for each cell. Any assignment to the variables that satisfies ϕ (and therefore ϕ_{cell}) must have exactly one variable on for every cell. Thus, any satisfying assignment specifies one symbol in each cell of the table. Parts ϕ_{start}, ϕ_{move}, and ϕ_{accept} ensure that these symbols actually correspond to an accepting tableau as follows.

Formula ϕ_{start} ensures that the first row of the table is the starting configuration of N on w by explicitly stipulating that the corresponding variables are on:

$$\begin{aligned}\phi_{\text{start}} = {} & x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}\,.\end{aligned}$$

Formula ϕ_{accept} guarantees that an accepting configuration occurs in the tableau. It ensures that q_{accept}, the symbol for the accept state, appears in one of the cells of the tableau by stipulating that one of the corresponding variables is on:

$$\phi_{\text{accept}} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{\text{accept}}}\,.$$

Finally, formula ϕ_{move} guarantees that each row of the tableau corresponds to a configuration that legally follows the preceding row's configuration according to N's rules. It does so by ensuring that each 2×3 window of cells is legal. We say that a 2×3 window is ***legal*** if that window does not violate the actions specified by N's transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.[3]

For example, say that a, b, and c are members of the tape alphabet, and q_1 and q_2 are states of N. Assume that when in state q_1 with the head reading an a, N writes a b, stays in state q_1, and moves right; and that when in state q_1 with the head reading a b, N nondeterministically either

1. writes a c, enters q_2, and moves to the left, or
2. writes an a, enters q_2, and moves to the right.

Expressed formally, $\delta(q_1, \text{a}) = \{(q_1,\text{b},\text{R})\}$ and $\delta(q_1, \text{b}) = \{(q_2,\text{c},\text{L}), (q_2,\text{a},\text{R})\}$. Examples of legal windows for this machine are shown in Figure 7.39.

---

[3]We could give a precise definition of ***legal window*** here, in terms of the transition function. But doing so is quite tedious and would distract us from the main thrust of the proof argument. Anyone desiring more precision should refer to the related analysis in the proof of Theorem 5.15, the undecidability of the Post Correspondence Problem.

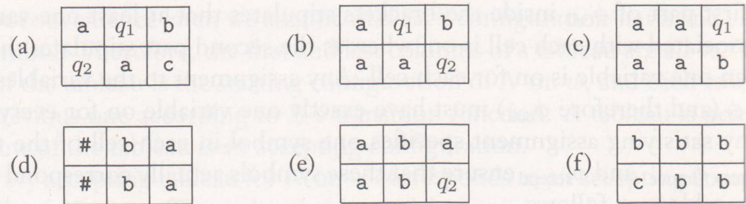


FIGURE 7.39
Examples of legal windows

In Figure 7.39, windows (a) and (b) are legal because the transition function allows N to move in the indicated way. Window (c) is legal because, with q_1 appearing on the right side of the top row, we don't know what symbol the head is over. That symbol could be an a, and q_1 might change it to a b and move to the right. That possibility would give rise to this window, so it doesn't violate N's rules. Window (d) is obviously legal because the top and bottom are identical, which would occur if the head weren't adjacent to the location of the window. Note that # may appear on the left or right of both the top and bottom rows in a legal window. Window (e) is legal because state q_1 reading a b might have been immediately to the right of the top row, and it would then have moved to the left in state q_2 to appear on the right-hand end of the bottom row. Finally, window (f) is legal because state q_1 might have been immediately to the left of the top row, and it might have changed the b to a c and moved to the left.

The windows shown in the following figure aren't legal for machine N.

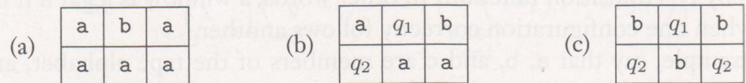


FIGURE 7.40
Examples of illegal windows

In window (a), the central symbol in the top row can't change because a state wasn't adjacent to it. Window (b) isn't legal because the transition function specifies that the b gets changed to a c but not to an a. Window (c) isn't legal because two states appear in the bottom row.

CLAIM 7.41 ....

If the top row of the tableau is the start configuration and every window in the tableau is legal, each row of the tableau is a configuration that legally follows the preceding one.

We prove this claim by considering any two adjacent configurations in the tableau, called the upper configuration and the lower configuration. In the upper configuration, every cell that contains a tape symbol and isn't adjacent to a state symbol is the center top cell in a window whose top row contains no states. Therefore, that symbol must appear unchanged in the center bottom of the window. Hence it appears in the same position in the bottom configuration.

The window containing the state symbol in the center top cell guarantees that the corresponding three positions are updated consistently with the transition function. Therefore, if the upper configuration is a legal configuration, so is the lower configuration, and the lower one follows the upper one according to N's rules. Note that this proof, though straightforward, depends crucially on our choice of a 2×3 window size, as Problem 7.41 shows.

Now we return to the construction of ϕ_{move}. It stipulates that all the windows in the tableau are legal. Each window contains six cells, which may be set in a fixed number of ways to yield a legal window. Formula ϕ_{move} says that the settings of those six cells must be one of these ways, or

$$\phi_{\text{move}} = \bigwedge_{1 \le i < n^k,\ 1 < j < n^k} \big(\text{the } (i,j)\text{-window is legal}\big).$$

The (i, j)-window has $cell[i, j]$ as the upper central position. We replace the text "the (i, j)-window is legal" in this formula with the following formula. We write the contents of six cells of a window as $a_1, \ldots, a_6$.

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} \left(x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6}\right)$$

Next, we analyze the complexity of the reduction to show that it operates in polynomial time. To do so, we examine the size of ϕ. First, we estimate the number of variables it has. Recall that the tableau is an $n^k \times n^k$ table, so it contains n^{2k} cells. Each cell has l variables associated with it, where l is the number of symbols in C. Because l depends only on the TM N and not on the length of the input n, the total number of variables is $O(n^{2k})$.

We estimate the size of each of the parts of ϕ. Formula ϕ_{cell} contains a fixed-size fragment of the formula for each cell of the tableau, so its size is $O(n^{2k})$. Formula ϕ_{start} has a fragment for each cell in the top row, so its size is $O(n^k)$. Formulas ϕ_{move} and ϕ_{accept} each contain a fixed-size fragment of the formula for each cell of the tableau, so their size is $O(n^{2k})$. Thus, ϕ's total size is $O(n^{2k})$. That bound is sufficient for our purposes because it shows that the size of ϕ is polynomial in n. If it were more than polynomial, the reduction wouldn't have any chance of generating it in polynomial time. (Actually, our estimates are low by a factor of $O(\log n)$ because each variable has indices that can range up to n^k and so may require $O(\log n)$ symbols to write into the formula, but this additional factor doesn't change the polynomiality of the result.)

To see that we can generate the formula in polynomial time, observe its highly repetitive nature. Each component of the formula is composed of many nearly

identical fragments, which differ only at the indices in a simple way. Therefore we may easily construct a reduction that produces ϕ in polynomial time from the input w.

Thus, we have concluded the proof of the Cook–Levin theorem, showing that *SAT* is NP-complete. Showing the NP-completeness of other languages generally doesn't require such a lengthy proof. Instead, NP-completeness can be proved with a polynomial time reduction from a language that is already known to be NP-complete. We can use *SAT* for this purpose; but using *3SAT*, the special case of *SAT* that we defined on page 302, is usually easier. Recall that the formulas in *3SAT* are in conjunctive normal form (cnf) with three literals per clause. First, we must show that *3SAT* itself is NP-complete. We prove this assertion as a corollary to Theorem 7.37.

COROLLARY 7.42

3SAT is NP-complete.

PROOF Obviously *3SAT* is in NP, so we only need to prove that all languages in NP reduce to *3SAT* in polynomial time. One way to do so is by showing that *SAT* polynomial time reduces to *3SAT*. Instead, we modify the proof of Theorem 7.37 so that it directly produces a formula in conjunctive normal form with three literals per clause.

Theorem 7.37 produces a formula that is already almost in conjunctive normal form. Formula ϕ_{cell} is a big AND of subformulas, each of which contains a big OR and a big AND of ORs. Thus, ϕ_{cell} is an AND of clauses and so is already in cnf. Formula ϕ_{start} is a big AND of variables. Taking each of these variables to be a clause of size 1, we see that ϕ_{start} is in cnf. Formula ϕ_{accept} is a big OR of variables and is thus a single clause. Formula ϕ_{move} is the only one that isn't already in cnf, but we may easily convert it into a formula that is in cnf as follows.

Recall that ϕ_{move} is a big AND of subformulas, each of which is an OR of ANDs that describes all possible legal windows. The distributive laws, as described in Chapter 0, state that we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each subformula, but it can only increase the total size of ϕ_{move} by a constant factor because the size of each subformula depends only on N. The result is a formula that is in conjunctive normal form.

Now that we have written the formula in cnf, we convert it to one with three literals per clause. In each clause that currently has one or two literals, we replicate one of the literals until the total number is three. In each clause that has more than three literals, we split it into several clauses and add additional variables to preserve the satisfiability or nonsatisfiability of the original.

For example, we replace clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$, wherein each a_i is a literal, with the two-clause expression $(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$, wherein z is a new

variable. If some setting of the a_i's satisfies the original clause, we can find some setting of z so that the two new clauses are satisfied and vice versa. In general, if the clause contains l literals,

$$(a_1 \vee a_2 \vee \cdots \vee a_l),$$

we can replace it with the $l - 2$ clauses

$$(a_1 \vee a_2 \vee z_1) \wedge (\overline{z_1} \vee a_3 \vee z_2) \wedge (\overline{z_2} \vee a_4 \vee z_3) \wedge \cdots \wedge (\overline{z_{l-3}} \vee a_{l-1} \vee a_l).$$

We may easily verify that the new formula is satisfiable iff the original formula was, so the proof is complete.

7.5

ADDITIONAL NP-COMPLETE PROBLEMS

The phenomenon of NP-completeness is widespread. NP-complete problems appear in many fields. For reasons that are not well understood, most naturally occurring NP-problems are known either to be in P or to be NP-complete. If you seek a polynomial time algorithm for a new NP-problem, spending part of your effort attempting to prove it NP-complete is sensible because doing so may prevent you from working to find a polynomial time algorithm that doesn't exist.

In this section, we present additional theorems showing that various languages are NP-complete. These theorems provide examples of the techniques that are used in proofs of this kind. Our general strategy is to exhibit a polynomial time reduction from *3SAT* to the language in question, though we sometimes reduce from other NP-complete languages when that is more convenient.

When constructing a polynomial time reduction from *3SAT* to a language, we look for structures in that language that can simulate the variables and clauses in Boolean formulas. Such structures are sometimes called ***gadgets***. For example, in the reduction from *3SAT* to *CLIQUE* presented in Theorem 7.32, individual nodes simulate variables and triples of nodes simulate clauses. An individual node may or may not be a member of the clique, corresponding to a variable that may or may not be true in a satisfying assignment. Each clause must contain a literal that is assigned TRUE. Correspondingly, each triple must contain a node in the clique (in order to reach the target size). The following corollary to Theorem 7.32 states that *CLIQUE* is NP-complete.

COROLLARY 7.43

CLIQUE is NP-complete.

THE VERTEX COVER PROBLEM

If G is an undirected graph, a ***vertex cover*** of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

$$\textit{VERTEX-COVER} = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

THEOREM 7.44

VERTEX-COVER is NP-complete.

PROOF IDEA To show that *VERTEX-COVER* is NP-complete, we must show that it is in NP and that all NP-problems are polynomial time reducible to it. The first part is easy; a certificate is simply a vertex cover of size k. To prove the second part, we show that *3SAT* is polynomial time reducible to *VERTEX-COVER*. The reduction converts a 3cnf-formula ϕ into a graph G and a number k, so that ϕ is satisfiable whenever G has a vertex cover with k nodes. The conversion is done without knowing whether ϕ is satisfiable. In effect, G simulates ϕ. The graph contains gadgets that mimic the variables and clauses of the formula. Designing these gadgets requires a bit of ingenuity.

For the variable gadget, we look for a structure in G that can participate in the vertex cover in either of two possible ways, corresponding to the two possible truth assignments to the variable. The variable gadget contains two nodes connected by an edge. That structure works because one of these nodes must appear in the vertex cover. We arbitrarily associate TRUE and FALSE with these two nodes.

For the clause gadget, we look for a structure that induces the vertex cover to include nodes in the variable gadgets corresponding to at least one true literal in the clause. The gadget contains three nodes and additional edges so that any vertex cover must include at least two of the nodes, or possibly all three. Only two nodes would be required if one of the variable gadget nodes helps by covering an edge, as would happen if the associated literal satisfies that clause. Otherwise, three nodes would be required. Finally, we chose k so that the sought-after vertex cover has one node per variable gadget and two nodes per clause gadget.

PROOF Here are the details of a reduction from *3SAT* to *VERTEX-COVER* that operates in polynomial time. The reduction maps a Boolean formula ϕ to a graph G and a value k. For each variable x in ϕ, we produce an edge connecting two nodes. We label the two nodes in this gadget x and $\overline{x}$. Setting x to be TRUE corresponds to selecting the node labeled x for the vertex cover, whereas FALSE corresponds to the node labeled $\overline{x}$.

The gadgets for the clauses are a bit more complex. Each clause gadget is a triple of nodes that are labeled with the three literals of the clause. These three nodes are connected to each other and to the nodes in the variable gadgets that have the identical labels. Thus, the total number of nodes that appear in G is $2m + 3l$, where ϕ has m variables and l clauses. Let k be $m + 2l$.

For example, if $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, the reduction produces $\langle G, k\rangle$ from ϕ, where $k = 8$ and G takes the form shown in the following figure.

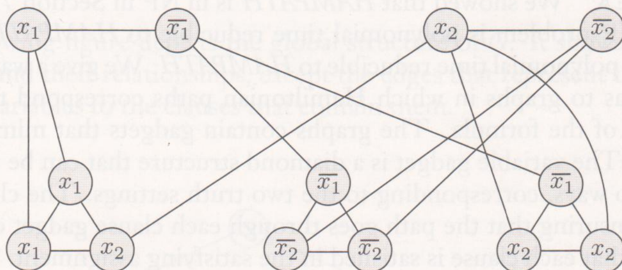


FIGURE 7.45
The graph that the reduction produces from $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

To prove that this reduction works, we need to show that ϕ is satisfiable if and only if G has a vertex cover with k nodes. We start with a satisfying assignment. We first put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover. Then, we select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover. Now we have a total of k nodes. They cover all edges because every variable gadget edge is clearly covered, all three edges within every clause gadget are covered, and all edges between variable and clause gadgets are covered. Hence G has a vertex cover with k nodes.

Second, if G has a vertex cover with k nodes, we show that ϕ is satisfiable by constructing the satisfying assignment. The vertex cover must contain one node in each variable gadget and two in every clause gadget in order to cover the edges of the variable gadgets and the three edges within the clause gadgets. That accounts for all the nodes, so none are left over. We take the nodes of the variable gadgets that are in the vertex cover and assign TRUE to the corresponding literals. That assignment satisfies ϕ because each of the three edges connecting the variable gadgets with each clause gadget is covered, and only two nodes of the clause gadget are in the vertex cover. Therefore, one of the edges must be covered by a node from a variable gadget and so that assignment satisfies the corresponding clause.

THE HAMILTONIAN PATH PROBLEM

Recall that the Hamiltonian path problem asks whether the input graph contains a path from s to t that goes through every node exactly once.

THEOREM 7.46

HAMPATH is NP-complete.

PROOF IDEA We showed that *HAMPATH* is in NP in Section 7.3. To show that every NP-problem is polynomial time reducible to *HAMPATH*, we show that *3SAT* is polynomial time reducible to *HAMPATH*. We give a way to convert 3cnf-formulas to graphs in which Hamiltonian paths correspond to satisfying assignments of the formula. The graphs contain gadgets that mimic variables and clauses. The variable gadget is a diamond structure that can be traversed in either of two ways, corresponding to the two truth settings. The clause gadget is a node. Ensuring that the path goes through each clause gadget corresponds to ensuring that each clause is satisfied in the satisfying assignment.

PROOF We previously demonstrated that *HAMPATH* is in NP, so all that remains to be done is to show $3SAT \leq_P HAMPATH$. For each 3cnf-formula ϕ, we show how to construct a directed graph G with two nodes, s and t, where a Hamiltonian path exists between s and t iff ϕ is satisfiable.

We start the construction with a 3cnf-formula ϕ containing k clauses,

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

where each a, b, and c is a literal x_i or $\overline{x_i}$. Let $x_1, \ldots, x_l$ be the l variables of ϕ.

Now we show how to convert ϕ to a graph G. The graph G that we construct has various parts to represent the variables and clauses that appear in ϕ.

We represent each variable x_i with a diamond-shaped structure that contains a horizontal row of nodes, as shown in the following figure. Later we specify the number of nodes that appear in the horizontal row.

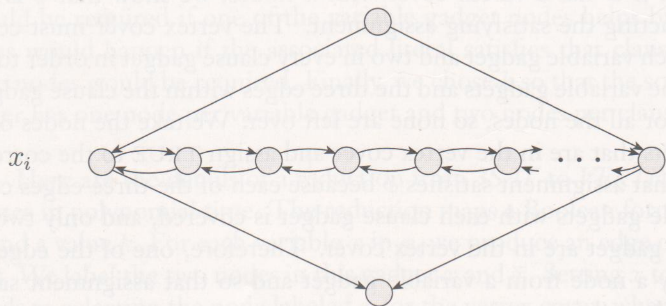
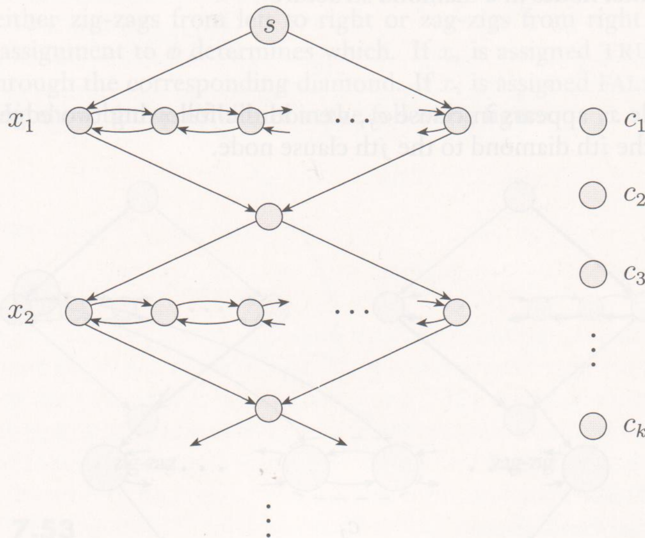


FIGURE 7.47
Representing the variable x_i as a diamond structure

We represent each clause of ϕ as a single node, as follows.

FIGURE 7.48
Representing the clause c_j as a node

The following figure depicts the global structure of G. It shows all the elements of G and their relationships, except the edges that represent the relationship of the variables to the clauses that contain them.



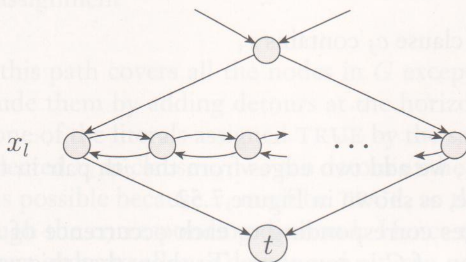


FIGURE 7.49
The high-level structure of G

Next, we show how to connect the diamonds representing the variables to the nodes representing the clauses. Each diamond structure contains a horizontal row of nodes connected by edges running in both directions. The horizontal row contains $3k + 1$ nodes in addition to the two nodes on the ends belonging to the diamond. These nodes are grouped into adjacent pairs, one for each clause, with extra separator nodes next to the pairs, as shown in the following figure.

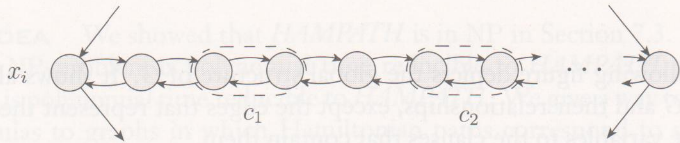


FIGURE 7.50
The horizontal nodes in a diamond structure

If variable x_i appears in clause c_j, we add the following two edges from the jth pair in the ith diamond to the jth clause node.

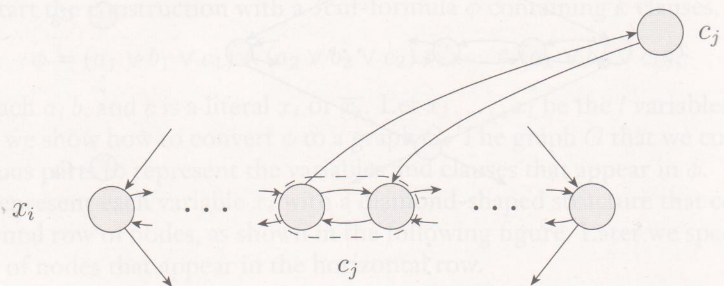


FIGURE 7.51
The additional edges when clause c_j contains x_i

If $\overline{x_i}$ appears in clause c_j, we add two edges from the jth pair in the ith diamond to the jth clause node, as shown in Figure 7.52.

After we add all the edges corresponding to each occurrence of x_i or $\overline{x_i}$ in each clause, the construction of G is complete. To show that this construction works, we argue that if ϕ is satisfiable, a Hamiltonian path exists from s to t; and conversely, if such a path exists, ϕ is satisfiable.

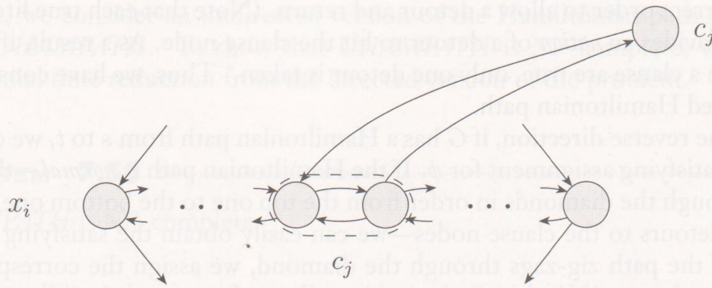


FIGURE 7.52
The additional edges when clause c_j contains $\overline{x_i}$

Suppose that ϕ is satisfiable. To demonstrate a Hamiltonian path from s to t, we first ignore the clause nodes. The path begins at s, goes through each diamond in turn, and ends up at t. To hit the horizontal nodes in a diamond, the path either zig-zags from left to right or zag-zigs from right to left; the satisfying assignment to ϕ determines which. If x_i is assigned TRUE, the path zig-zags through the corresponding diamond. If x_i is assigned FALSE, the path zag-zigs. We show both possibilities in the following figure.

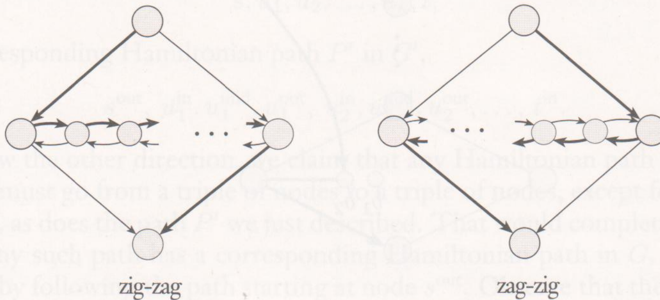


FIGURE 7.53
Zig-zagging and zag-zigging through a diamond, as determined by the satisfying assignment

So far, this path covers all the nodes in G except the clause nodes. We can easily include them by adding detours at the horizontal nodes. In each clause, we select one of the literals assigned TRUE by the satisfying assignment.

If we selected x_i in clause c_j, we can detour at the jth pair in the ith diamond. Doing so is possible because x_i must be TRUE, so the path zig-zags from left to right through the corresponding diamond. Hence the edges to the c_j node are in the correct order to allow a detour and return.

Similarly, if we selected $\overline{x_i}$ in clause c_j, we can detour at the jth pair in the ith diamond because x_i must be FALSE, so the path zag-zigs from right to left through the corresponding diamond. Hence the edges to the c_j node again are

in the correct order to allow a detour and return. (Note that each true literal in a clause provides an *option* of a detour to hit the clause node. As a result, if several literals in a clause are true, only one detour is taken.) Thus, we have constructed the desired Hamiltonian path.

For the reverse direction, if G has a Hamiltonian path from s to t, we demonstrate a satisfying assignment for ϕ. If the Hamiltonian path is *normal*—that is, it goes through the diamonds in order from the top one to the bottom one, except for the detours to the clause nodes—we can easily obtain the satisfying assignment. If the path zig-zags through the diamond, we assign the corresponding variable TRUE; and if it zag-zigs, we assign FALSE. Because each clause node appears on the path, by observing how the detour to it is taken, we may determine which of the literals in the corresponding clause is TRUE.

All that remains to be shown is that a Hamiltonian path must be normal. Normality may fail only if the path enters a clause from one diamond but returns to another, as in the following figure.

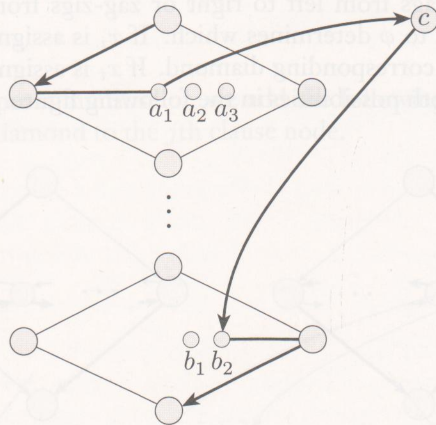


FIGURE 7.54
This situation cannot occur

The path goes from node a_1 to c; but instead of returning to a_2 in the same diamond, it returns to b_2 in a different diamond. If that occurs, either a_2 or a_3 must be a separator node. If a_2 were a separator node, the only edges entering a_2 would be from a_1 and a_3. If a_3 were a separator node, a_1 and a_2 would be in the same clause pair, and hence the only edges entering a_2 would be from a_1, a_3, and c. In either case, the path could not contain node a_2. The path cannot enter a_2 from c or a_1 because the path goes elsewhere from these nodes. The path cannot enter a_2 from a_3 because a_3 is the only available node that a_2 points at, so the path must exit a_2 via a_3. Hence a Hamiltonian path must be normal. This reduction obviously operates in polynomial time and the proof is complete.

Next, we consider an undirected version of the Hamiltonian path problem, called *UHAMPATH*. To show that *UHAMPATH* is NP-complete, we give a polynomial time reduction from the directed version of the problem.

THEOREM 7.55

UHAMPATH is NP-complete.

PROOF The reduction takes a directed graph G with nodes s and t, and constructs an undirected graph G' with nodes s' and t'. Graph G has a Hamiltonian path from s to t iff G' has a Hamiltonian path from s' to t'. We describe G' as follows.

Each node u of G, except for s and t, is replaced by a triple of nodes u^{in}, u^{mid}, and u^{out} in G'. Nodes s and t in G are replaced by nodes $s^{\text{out}} = s'$ and $t^{\text{in}} = t'$ in G'. Edges of two types appear in G'. First, edges connect u^{mid} with u^{in} and u^{out}. Second, an edge connects u^{out} with v^{in} if an edge goes from u to v in G. That completes the construction of G'.

We can demonstrate that this construction works by showing that G has a Hamiltonian path from s to t iff G' has a Hamiltonian path from s^{out} to t^{in}. To show one direction, we observe that a Hamiltonian path P in G,

$$s, u_1, u_2, \ldots, u_k, t,$$

has a corresponding Hamiltonian path P' in G',

$$s^{\text{out}}, u_1^{\text{in}}, u_1^{\text{mid}}, u_1^{\text{out}}, u_2^{\text{in}}, u_2^{\text{mid}}, u_2^{\text{out}}, \ldots, t^{\text{in}}.$$

To show the other direction, we claim that any Hamiltonian path in G' from s^{out} to t^{in} must go from a triple of nodes to a triple of nodes, except for the start and finish, as does the path P' we just described. That would complete the proof because any such path has a corresponding Hamiltonian path in G. We prove the claim by following the path starting at node s^{out}. Observe that the next node in the path must be u_i^{in} for some i because only those nodes are connected to s^{out}. The next node must be u_i^{mid} because no other way is available to include u_i^{mid} in the Hamiltonian path. After u_i^{mid} comes u_i^{out} because that is the only other node to which u_i^{mid} is connected. The next node must be u_j^{in} for some j because no other available node is connected to u_i^{out}. The argument then repeats until t^{in} is reached.

THE SUBSET SUM PROBLEM

Recall the *SUBSET-SUM* problem defined on page 297. In that problem, we were given a collection of numbers $x_1, \ldots, x_k$ together with a target number t, and were to determine whether the collection contains a subcollection that adds up to t. We now show that this problem is NP-complete.

THEOREM 7.56

SUBSET-SUM is NP-complete.

PROOF IDEA We have already shown that *SUBSET-SUM* is in NP in The-
orem 7.25. We prove that all languages in NP are polynomial time reducible
to *SUBSET-SUM* by reducing the NP-complete language *3SAT* to it. Given
a 3cnf-formula ϕ, we construct an instance of the *SUBSET-SUM* problem that
contains a subcollection summing to the target t if and only if ϕ is satisfiable.
Call this subcollection T.

To achieve this reduction, we find structures of the *SUBSET-SUM* problem
that represent variables and clauses. The *SUBSET-SUM* problem instance that
we construct contains numbers of large magnitude presented in decimal nota-
tion. We represent variables by pairs of numbers and clauses by certain positions
in the decimal representations of the numbers.

We represent variable x_i by two numbers, y_i and z_i. We prove that either y_i
or z_i must be in T for each i, which establishes the encoding for the truth value
of x_i in the satisfying assignment.

Each clause position contains a certain value in the target t, which imposes a
requirement on the subset T. We prove that this requirement is the same as the
one in the corresponding clause—namely, that one of the literals in that clause
is assigned TRUE.

PROOF We already know that *SUBSET-SUM* $\in$ NP, so we now show that
3SAT $\leq_{\mathrm{P}}$ *SUBSET-SUM*.

Let ϕ be a Boolean formula with variables $x_1, \ldots, x_l$ and clauses $c_1, \ldots$
The reduction converts ϕ to an instance of the *SUBSET-SUM* problem $\langle S, t \rangle$,
wherein the elements of S and the number t are the rows in the table in Fig-
ure 7.57, expressed in ordinary decimal notation. The rows above the double
line are labeled

$$y_1, z_1, y_2, z_2, \ldots, y_l, z_l \quad \text{and} \quad g_1, h_1, g_2, h_2, \ldots, g_k, h_k$$

and constitute the elements of S. The row below the double line is t.

Thus, S contains one pair of numbers, y_i, z_i, for each variable x_i in ϕ. The
decimal representation of these numbers is in two parts, as indicated in the table.
The left-hand part comprises a 1 followed by $l - i$ 0s. The right-hand part
contains one digit for each clause, where the digit of y_i in column c_j is 1 if clause
c_j contains literal x_i, and the digit of z_i in column c_j is 1 if clause c_j contains
literal $\overline{x_i}$. Digits not specified to be 1 are 0.

The table is partially filled in to illustrate sample clauses, c_1, c_2, and c_k:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots).$$

Additionally, S contains one pair of numbers, g_j, h_j, for each clause c_j. These
two numbers are equal and consist of a 1 followed by $k - j$ 0s.

Finally, the target number t, the bottom row of the table, consists of l 1s followed by k 3s.

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

FIGURE 7.57
Reducing *3SAT* to *SUBSET-SUM*

Next, we show why this construction works. We demonstrate that ϕ is satisfiable iff some subset of S sums to t.

Suppose that ϕ is satisfiable. We construct a subset of S as follows. We select y_i if x_i is assigned TRUE in the satisfying assignment, and z_i if x_i is assigned FALSE. If we add up what we have selected so far, we obtain a 1 in each of the first l digits because we have selected either y_i or z_i for each i. Furthermore, each of the last k digits is a number between 1 and 3 because each clause is satisfied and so contains between 1 and 3 true literals. We additionally select enough of the g and h numbers to bring each of the last k digits up to 3, thus hitting the target.

Suppose that a subset of S sums to t. We construct a satisfying assignment to ϕ after making several observations. First, all the digits in members of S are either 0 or 1. Furthermore, each column in the table describing S contains at most five 1s. Hence a "carry" into the next column never occurs when a subset of S is added. To get a 1 in each of the first l columns, the subset must have either y_i or z_i for each i, but not both.

Now we make the satisfying assignment. If the subset contains y_i, we assign x_i TRUE; otherwise, we assign it FALSE. This assignment must satisfy ϕ because in each of the final k columns, the sum is always 3. In column c_j, at most 2 can come from g_j and h_j, so at least 1 in this column must come from some y_i or z_i in the subset. If it is y_i, then x_i appears in c_j and is assigned TRUE, so c_j is satisfied. If it is z_i, then $\overline{x_i}$ appears in c_j and x_i is assigned FALSE, so c_j is satisfied. Therefore, ϕ is satisfied.

Finally, we must be sure that the reduction can be carried out in polynomial time. The table has a size of roughly $(k + l)^2$ and each entry can be easily calculated for any ϕ. So the total time is $O(n^2)$ easy stages.

EXERCISES

7.1 Answer each part TRUE or FALSE.

- **a.** $2n = O(n)$.
- **b.** $n^2 = O(n)$.
- $^{\mathrm{A}}$**c.** $n^2 = O(n \log^2 n)$.
- $^{\mathrm{A}}$**d.** $n \log n = O(n^2)$.
- **e.** $3^n = 2^{O(n)}$.
- **f.** $2^{2^n} = O(2^{2^n})$.

7.2 Answer each part TRUE or FALSE.

- **a.** $n = o(2n)$.
- **b.** $2n = o(n^2)$.
- $^{\mathrm{A}}$**c.** $2^n = o(3^n)$.
- $^{\mathrm{A}}$**d.** $1 = o(n)$.
- **e.** $n = o(\log n)$.
- **f.** $1 = o(1/n)$.

7.3 Which of the following pairs of numbers are relatively prime? Show the calculations that led to your conclusions.

- **a.** 1274 and 10505
- **b.** 7289 and 8029

7.4 Fill out the table described in the polynomial time algorithm for context-free language recognition from Theorem 7.16 for string $w =$ baba and CFG G:

$$S \to RT$$
$$R \to TR \mid \mathtt{a}$$
$$T \to TR \mid \mathtt{b}$$

7.5 Is the following formula satisfiable?

$$(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee \overline{y})$$

7.6 Show that P is closed under union, concatenation, and complement.

7.7 Show that NP is closed under union and concatenation.

7.8 Let $CONNECTED = \{\langle G \rangle |\ G$ is a connected undirected graph$\}$. Analyze the algorithm given on page 185 to show that this language is in P.

7.9 A ***triangle*** in an undirected graph is a 3-clique. Show that $TRIANGLE \in \text{P}$, where $TRIANGLE = \{\langle G \rangle |\ G$ contains a triangle$\}$.

7.10 Show that ALL_{DFA} is in P.

7.11 In both parts, provide an analysis of the time complexity of your algorithm.

a. Show that $EQ_{\text{DFA}} \in \text{P}$.

b. Say that a language A is ***star-closed*** if $A = A^*$. Give a polynomial time algorithm to test whether a DFA recognizes a star-closed language. (Note that EQ_{NFA} is not known to be in P.)

7.12 Call graphs G and H ***isomorphic*** if the nodes of G may be reordered so that it is identical to H. Let $ISO = \{\langle G, H \rangle |\ G$ and H are isomorphic graphs$\}$. Show that $ISO \in \text{NP}$.

PROBLEMS

7.13 Let

$$MODEXP = \{\langle a, b, c, p \rangle |\ a, b, c, \text{ and } p \text{ are positive binary integers such that } a^b \equiv c \pmod{p}\}.$$

Show that $MODEXP \in \text{P}$. (Note that the most obvious algorithm doesn't run in polynomial time. Hint: Try it first where b is a power of 2.)

7.14 A ***permutation*** on the set $\{1, \ldots, k\}$ is a one-to-one, onto function on this set. When p is a permutation, p^t means the composition of p with itself t times. Let

$$PERM\text{-}POWER = \{\langle p, q, t \rangle |\ p = q^t \text{ where } p \text{ and } q \text{ are permutations on } \{1, \ldots, k\} \text{ and } t \text{ is a binary integer}\}.$$

Show that $PERM\text{-}POWER \in \text{P}$. (Note that the most obvious algorithm doesn't run within polynomial time. Hint: First try it where t is a power of 2.)

7.15 Show that P is closed under the star operation. (Hint: Use dynamic programming. On input $y = y_1 \cdots y_n$ for $y_i \in \Sigma$, build a table indicating for each $i \leq j$ whether the substring $y_i \cdots y_j \in A^*$ for any $A \in \text{P}$.)

[A]**7.16** Show that NP is closed under the star operation.

7.17 Let $UNARY\text{-}SSUM$ be the subset sum problem in which all numbers are represented in unary. Why does the NP-completeness proof for $SUBSET\text{-}SUM$ fail to show $UNARY\text{-}SSUM$ is NP-complete? Show that $UNARY\text{-}SSUM \in \text{P}$.

7.18 Show that if P = NP, then every language $A \in \text{P}$, except $A = \emptyset$ and $A = \Sigma^*$, is NP-complete.