VII Selected Topics

Introduction

This part contains a selection of algorithmic topics that extend and complement earlier material in this book. Some chapters introduce new models of computation such as circuits or parallel computers. Others cover specialized domains such as computational geometry or number theory. The last two chapters discuss some of the known limitations to the design of efficient algorithms and introduce techniques for coping with those limitations.

Chapter 27 presents an algorithmic model for parallel computing based on dynamic multithreading. The chapter introduces the basics of the model, showing how to quantify parallelism in terms of the measures of work and span. It then investigates several interesting multithreaded algorithms, including algorithms for matrix multiplication and merge sorting.

Chapter 28 studies efficient algorithms for operating on matrices. It presents two general methods—LU decomposition and LUP decomposition—for solving linear equations by Gaussian elimination in $O(n^3)$ time. It also shows that matrix inversion and matrix multiplication can be performed equally fast. The chapter concludes by showing how to compute a least-squares approximate solution when a set of linear equations has no exact solution.

Chapter 29 studies linear programming, in which we wish to maximize or minimize an objective, given limited resources and competing constraints. Linear programming arises in a variety of practical application areas. This chapter covers how to formulate and solve linear programs. The solution method covered is the simplex algorithm, which is the oldest algorithm for linear programming. In contrast to many algorithms in this book, the simplex algorithm does not run in polynomial time in the worst case, but it is fairly efficient and widely used in practice.

Chapter 30 studies operations on polynomials and shows how to use a well-known signal-processing technique—the fast Fourier transform (FFT)—to multiply two degree-n polynomials in $O(n \lg n)$ time. It also investigates efficient implementations of the FFT, including a parallel circuit.

Chapter 31 presents number-theoretic algorithms. After reviewing elementary number theory, it presents Euclid's algorithm for computing greatest common divisors. Next, it studies algorithms for solving modular linear equations and for raising one number to a power modulo another number. Then, it explores an important application of number-theoretic algorithms: the RSA public-key cryptosystem. This cryptosystem can be used not only to encrypt messages so that an adversary cannot read them, but also to provide digital signatures. The chapter then presents the Miller-Rabin randomized primality test, with which we can find large primes efficiently—an essential requirement for the RSA system. Finally, the chapter covers Pollard's "rho" heuristic for factoring integers and discusses the state of the art of integer factorization.

Chapter 32 studies the problem of finding all occurrences of a given pattern string in a given text string, a problem that arises frequently in text-editing programs. After examining the naive approach, the chapter presents an elegant approach due to Rabin and Karp. Then, after showing an efficient solution based on finite automata, the chapter presents the Knuth-Morris-Pratt algorithm, which modifies the automaton-based algorithm to save space by cleverly preprocessing the pattern.

Chapter 33 considers a few problems in computational geometry. After discussing basic primitives of computational geometry, the chapter shows how to use a "sweeping" method to efficiently determine whether a set of line segments contains any intersections. Two clever algorithms for finding the convex hull of a set of points—Graham's scan and Jarvis's march—also illustrate the power of sweeping methods. The chapter closes with an efficient algorithm for finding the closest pair from among a given set of points in the plane.

Chapter 34 concerns NP-complete problems. Many interesting computational problems are NP-complete, but no polynomial-time algorithm is known for solving any of them. This chapter presents techniques for determining when a problem is NP-complete. Several classic problems are proved to be NP-complete: determining whether a graph has a hamiltonian cycle, determining whether a boolean formula is satisfiable, and determining whether a given set of numbers has a subset that adds up to a given target value. The chapter also proves that the famous traveling-salesman problem is NP-complete.

Chapter 35 shows how to find approximate solutions to NP-complete problems efficiently by using approximation algorithms. For some NP-complete problems, approximate solutions that are near optimal are quite easy to produce, but for others even the best approximation algorithms known work progressively more poorly as

the problem size increases. Then, there are some problems for which we can invest increasing amounts of computation time in return for increasingly better approximate solutions. This chapter illustrates these possibilities with the vertex-cover problem (unweighted and weighted versions), an optimization version of 3-CNF satisfiability, the traveling-salesman problem, the set-covering problem, and the subset-sum problem.

27 Multithreaded Algorithms

The vast majority of algorithms in this book are ***serial algorithms*** suitable for running on a uniprocessor computer in which only one instruction executes at a time. In this chapter, we shall extend our algorithmic model to encompass ***parallel algorithms***, which can run on a multiprocessor computer that permits multiple instructions to execute concurrently. In particular, we shall explore the elegant model of dynamic multithreaded algorithms, which are amenable to algorithmic design and analysis, as well as to efficient implementation in practice.

Parallel computers—computers with multiple processing units—have become increasingly common, and they span a wide range of prices and performance. Relatively inexpensive desktop and laptop ***chip multiprocessors*** contain a single ***multicore*** integrated-circuit chip that houses multiple processing "cores," each of which is a full-fledged processor that can access a common memory. At an intermediate price/performance point are clusters built from individual computers—often simple PC-class machines—with a dedicated network interconnecting them. The highest-priced machines are supercomputers, which often use a combination of custom architectures and custom networks to deliver the highest performance in terms of instructions executed per second.

Multiprocessor computers have been around, in one form or another, for decades. Although the computing community settled on the random-access machine model for serial computing early on in the history of computer science, no single model for parallel computing has gained as wide acceptance. A major reason is that vendors have not agreed on a single architectural model for parallel computers. For example, some parallel computers feature ***shared memory***, where each processor can directly access any location of memory. Other parallel computers employ ***distributed memory***, where each processor's memory is private, and an explicit message must be sent between processors in order for one processor to access the memory of another. With the advent of multicore technology, however, every new laptop and desktop machine is now a shared-memory parallel computer,

and the trend appears to be toward shared-memory multiprocessing. Although time will tell, that is the approach we shall take in this chapter.

One common means of programming chip multiprocessors and other shared-memory parallel computers is by using ***static threading***, which provides a software abstraction of "virtual processors," or ***threads***, sharing a common memory. Each thread maintains an associated program counter and can execute code independently of the other threads. The operating system loads a thread onto a processor for execution and switches it out when another thread needs to run. Although the operating system allows programmers to create and destroy threads, these operations are comparatively slow. Thus, for most applications, threads persist for the duration of a computation, which is why we call them "static."

Unfortunately, programming a shared-memory parallel computer directly using static threads is difficult and error-prone. One reason is that dynamically partitioning the work among the threads so that each thread receives approximately the same load turns out to be a complicated undertaking. For any but the simplest of applications, the programmer must use complex communication protocols to implement a scheduler to load-balance the work. This state of affairs has led toward the creation of ***concurrency platforms***, which provide a layer of software that coordinates, schedules, and manages the parallel-computing resources. Some concurrency platforms are built as runtime libraries, but others provide full-fledged parallel languages with compiler and runtime support.

Dynamic multithreaded programming

One important class of concurrency platform is ***dynamic multithreading***, which is the model we shall adopt in this chapter. Dynamic multithreading allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming. The concurrency platform contains a scheduler, which load-balances the computation automatically, thereby greatly simplifying the programmer's chore. Although the functionality of dynamic-multithreading environments is still evolving, almost all support two features: nested parallelism and parallel loops. Nested parallelism allows a subroutine to be "spawned," allowing the caller to proceed while the spawned subroutine is computing its result. A parallel loop is like an ordinary **for** loop, except that the iterations of the loop can execute concurrently.

These two features form the basis of the model for dynamic multithreading that we shall study in this chapter. A key aspect of this model is that the programmer needs to specify only the logical parallelism within a computation, and the threads within the underlying concurrency platform schedule and load-balance the computation among themselves. We shall investigate multithreaded algorithms written for

this model, as well how the underlying concurrency platform can schedule computations efficiently.

Our model for dynamic multithreading offers several important advantages:

- It is a simple extension of our serial programming model. We can describe a multithreaded algorithm by adding to our pseudocode just three "concurrency" keywords: **parallel**, **spawn**, and **sync**. Moreover, if we delete these concurrency keywords from the multithreaded pseudocode, the resulting text is serial pseudocode for the same problem, which we call the "serialization" of the multithreaded algorithm.
- It provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."
- Many multithreaded algorithms involving nested parallelism follow naturally from the divide-and-conquer paradigm. Moreover, just as serial divide-and-conquer algorithms lend themselves to analysis by solving recurrences, so do multithreaded algorithms.
- The model is faithful to how parallel-computing practice is evolving. A growing number of concurrency platforms support one variant or another of dynamic multithreading, including Cilk [51, 118], Cilk++ [71], OpenMP [59], Task Parallel Library [230], and Threading Building Blocks [292].

Section 27.1 introduces the dynamic multithreading model and presents the metrics of work, span, and parallelism, which we shall use to analyze multithreaded algorithms. Section 27.2 investigates how to multiply matrices with multithreading, and Section 27.3 tackles the tougher problem of multithreading merge sort.

27.1 The basics of dynamic multithreading

We shall begin our exploration of dynamic multithreading using the example of computing Fibonacci numbers recursively. Recall that the Fibonacci numbers are defined by recurrence (3.22):

$$\begin{aligned} F_0 &= 0 , \\ F_1 &= 1 , \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2 . \end{aligned}$$

Here is a simple, recursive, serial algorithm to compute the nth Fibonacci number:

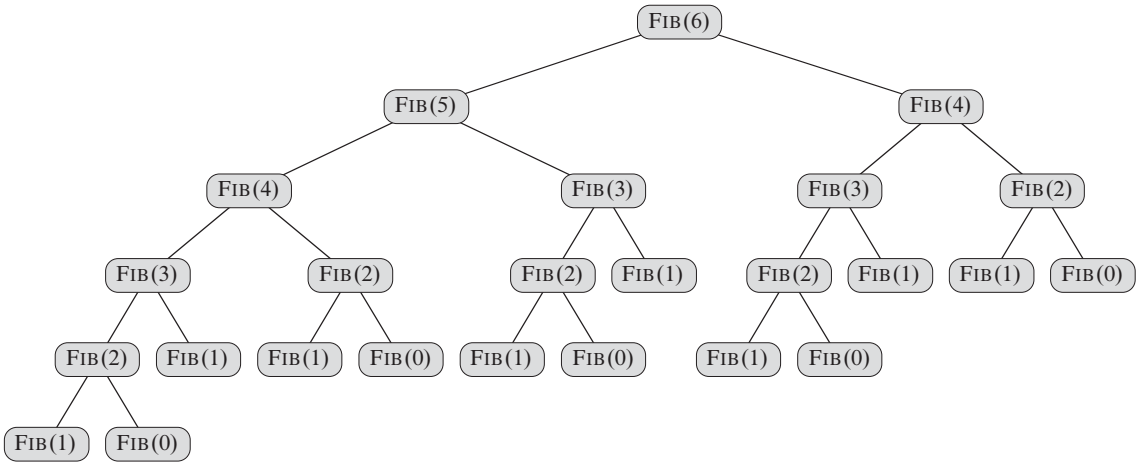


Figure 27.1 The tree of recursive procedure instances when computing FIB(6). Each instance of FIB with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

```
FIB(n)
1  if n ≤ 1
2      return n
3  else x = FIB(n − 1)
4       y = FIB(n − 2)
5       return x + y
```

You would not really want to compute large Fibonacci numbers this way, because this computation does much repeated work. Figure 27.1 shows the tree of recursive procedure instances that are created when computing F_6. For example, a call to FIB(6) recursively calls FIB(5) and then FIB(4). But, the call to FIB(5) also results in a call to FIB(4). Both instances of FIB(4) return the same result ($F_4 = 3$). Since the FIB procedure does not memoize, the second call to FIB(4) replicates the work that the first call performs.

Let $T(n)$ denote the running time of FIB(n). Since FIB(n) contains two recursive calls plus a constant amount of extra work, we obtain the recurrence

$$T(n) = T(n-1) + T(n-2) + \Theta(1) .$$

This recurrence has solution $T(n) = \Theta(F_n)$, which we can show using the substitution method. For an inductive hypothesis, assume that $T(n) \le a F_n - b$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

$$\begin{aligned} T(n) &\le (aF_{n-1} - b) + (aF_{n-2} - b) + \Theta(1) \\ &= a(F_{n-1} + F_{n-2}) - 2b + \Theta(1) \\ &= aF_n - b - (b - \Theta(1)) \\ &\le aF_n - b \end{aligned}$$

if we choose b large enough to dominate the constant in the $\Theta(1)$. We can then choose a large enough to satisfy the initial condition. The analytical bound

$$T(n) = \Theta(\phi^n) , \tag{27.1}$$

where $\phi = (1 + \sqrt{5})/2$ is the golden ratio, now follows from equation (3.25). Since F_n grows exponentially in n, this procedure is a particularly slow way to compute Fibonacci numbers. (See Problem 31-3 for much faster ways.)

Although the FIB procedure is a poor way to compute Fibonacci numbers, it makes a good example for illustrating key concepts in the analysis of multithreaded algorithms. Observe that within FIB(n), the two recursive calls in lines 3 and 4 to FIB($n-1$) and FIB($n-2$), respectively, are independent of each other: they could be called in either order, and the computation performed by one in no way affects the other. Therefore, the two recursive calls can run in parallel.

We augment our pseudocode to indicate parallelism by adding the ***concurrency keywords*** **spawn** and **sync**. Here is how we can rewrite the FIB procedure to use dynamic multithreading:

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)
4       y = P-FIB(n − 2)
5       sync
6       return x + y
```

Notice that if we delete the concurrency keywords **spawn** and **sync** from P-FIB, the resulting pseudocode text is identical to FIB (other than renaming the procedure in the header and in the two recursive calls). We define the ***serialization*** of a multithreaded algorithm to be the serial algorithm that results from deleting the multithreaded keywords: **spawn**, **sync**, and when we examine parallel loops, **parallel**. Indeed, our multithreaded pseudocode has the nice property that a serialization is always ordinary serial pseudocode to solve the same problem.

Nested parallelism occurs when the keyword **spawn** precedes a procedure call, as in line 3. The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn—the ***parent***—may continue to execute in parallel with the spawned subroutine—its ***child***—instead of waiting

for the child to complete, as would normally happen in a serial execution. In this case, while the spawned child is computing P-FIB$(n-1)$, the parent may go on to compute P-FIB$(n-2)$ in line 4 in parallel with the spawned child. Since the P-FIB procedure is recursive, these two subroutine calls themselves create nested parallelism, as do their children, thereby creating a potentially vast tree of subcomputations, all executing in parallel.

The keyword **spawn** does not say, however, that a procedure *must* execute concurrently with its spawned children, only that it *may*. The concurrency keywords express the ***logical parallelism*** of the computation, indicating which parts of the computation may proceed in parallel. At runtime, it is up to a ***scheduler*** to determine which subcomputations actually run concurrently by assigning them to available processors as the computation unfolds. We shall discuss the theory behind schedulers shortly.

A procedure cannot safely use the values returned by its spawned children until after it executes a **sync** statement, as in line 5. The keyword **sync** indicates that the procedure must wait as necessary for all its spawned children to complete before proceeding to the statement after the **sync**. In the P-FIB procedure, a **sync** is required before the **return** statement in line 6 to avoid the anomaly that would occur if x and y were summed before x was computed. In addition to explicit synchronization provided by the **sync** statement, every procedure executes a **sync** implicitly before it returns, thus ensuring that all its children terminate before it does.

A model for multithreaded execution

It helps to think of a ***multithreaded computation***—the set of runtime instructions executed by a processor on behalf of a multithreaded program—as a directed acyclic graph $G = (V, E)$, called a ***computation dag***. As an example, Figure 27.2 shows the computation dag that results from computing P-FIB(4). Conceptually, the vertices in V are instructions, and the edges in E represent dependencies between instructions, where $(u, v) \in E$ means that instruction u must execute before instruction v. For convenience, however, if a chain of instructions contains no parallel control (no **spawn**, **sync**, or **return** from a spawn—via either an explicit **return** statement or the return that happens implicitly upon reaching the end of a procedure), we may group them into a single ***strand***, each of which represents one or more instructions. Instructions involving parallel control are not included in strands, but are represented in the structure of the dag. For example, if a strand has two successors, one of them must have been spawned, and a strand with multiple predecessors indicates the predecessors joined because of a **sync** statement. Thus, in the general case, the set V forms the set of strands, and the set E of directed edges represents dependencies between strands induced by parallel control.

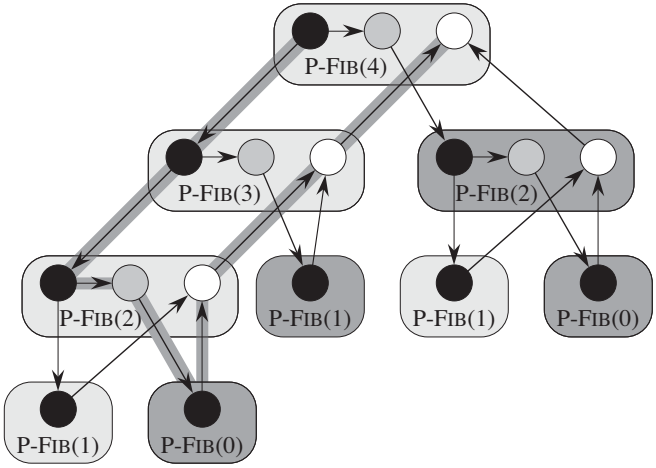


Figure 27.2 A directed acyclic graph representing the computation of P-FIB(4). Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3, shaded circles representing the part of the procedure that calls P-FIB($n-2$) in line 4 up to the **sync** in line 5, where it suspends until the spawn of P-FIB($n-1$) returns, and white circles representing the part of the procedure after the **sync** where it sums x and y up to the point where it returns the result. Each group of strands belonging to the same procedure is surrounded by a rounded rectangle, lightly shaded for spawned procedures and heavily shaded for called procedures. Spawn edges and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, the work equals 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with shaded edges—contains 8 strands.

If G has a directed path from strand u to strand v, we say that the two strands are ***(logically) in series***. Otherwise, strands u and v are ***(logically) in parallel***.

We can picture a multithreaded computation as a dag of strands embedded in a tree of procedure instances. For example, Figure 27.1 shows the tree of procedure instances for P-FIB(6) without the detailed structure showing strands. Figure 27.2 zooms in on a section of that tree, showing the strands that constitute each procedure. All directed edges connecting strands run either within a procedure or along undirected edges in the procedure tree.

We can classify the edges of a computation dag to indicate the kind of dependencies between the various strands. A ***continuation edge*** (u, u'), drawn horizontally in Figure 27.2, connects a strand u to its successor u' within the same procedure instance. When a strand u spawns a strand v, the dag contains a ***spawn edge*** (u, v), which points downward in the figure. ***Call edges***, representing normal procedure calls, also point downward. Strand u spawning strand v differs from u calling v in that a spawn induces a horizontal continuation edge from u to the strand u' fol-

lowing u in its procedure, indicating that u' is free to execute at the same time as v, whereas a call induces no such edge. When a strand u returns to its calling procedure and x is the strand immediately following the next **sync** in the calling procedure, the computation dag contains ***return edge*** (u, x), which points upward. A computation starts with a single ***initial strand***—the black vertex in the procedure labeled P-FIB(4) in Figure 27.2—and ends with a single ***final strand***—the white vertex in the procedure labeled P-FIB(4).

We shall study the execution of multithreaded algorithms on an ***ideal parallel computer***, which consists of a set of processors and a ***sequentially consistent*** shared memory. Sequential consistency means that the shared memory, which may in reality be performing many loads and stores from the processors at the same time, produces the same results as if at each step, exactly one instruction from one of the processors is executed. That is, the memory behaves as if the instructions were executed sequentially according to some global linear order that preserves the individual orders in which each processor issues its own instructions. For dynamic multithreaded computations, which are scheduled onto processors automatically by the concurrency platform, the shared memory behaves as if the multithreaded computation's instructions were interleaved to produce a linear order that preserves the partial order of the computation dag. Depending on scheduling, the ordering could differ from one run of the program to another, but the behavior of any execution can be understood by assuming that the instructions are executed in some linear order consistent with the computation dag.

In addition to making assumptions about semantics, the ideal-parallel-computer model makes some performance assumptions. Specifically, it assumes that each processor in the machine has equal computing power, and it ignores the cost of scheduling. Although this last assumption may sound optimistic, it turns out that for algorithms with sufficient "parallelism" (a term we shall define precisely in a moment), the overhead of scheduling is generally minimal in practice.

Performance measures

We can gauge the theoretical efficiency of a multithreaded algorithm by using two metrics: "work" and "span." The ***work*** of a multithreaded computation is the total time to execute the entire computation on one processor. In other words, the work is the sum of the times taken by each of the strands. For a computation dag in which each strand takes unit time, the work is just the number of vertices in the dag. The ***span*** is the longest time to execute the strands along any path in the dag. Again, for a dag in which each strand takes unit time, the span equals the number of vertices on a longest or ***critical path*** in the dag. (Recall from Section 24.2 that we can find a critical path in a dag $G = (V, E)$ in $\Theta(V + E)$ time.) For example, the computation dag of Figure 27.2 has 17 vertices in all and 8 vertices on its critical

path, so that if each strand takes unit time, its work is 17 time units and its span is 8 time units.

The actual running time of a multithreaded computation depends not only on its work and its span, but also on how many processors are available and how the scheduler allocates strands to processors. To denote the running time of a multithreaded computation on P processors, we shall subscript by P. For example, we might denote the running time of an algorithm on P processors by T_P. The work is the running time on a single processor, or T_1. The span is the running time if we could run each strand on its own processor—in other words, if we had an unlimited number of processors—and so we denote the span by T_∞.

The work and span provide lower bounds on the running time T_P of a multithreaded computation on P processors:

- In one step, an ideal parallel computer with P processors can do at most P units of work, and thus in T_P time, it can perform at most PT_P work. Since the total work to do is T_1, we have $PT_P \geq T_1$. Dividing by P yields the ***work law***:

$$T_P \geq T_1/P \ . \tag{27.2}$$

- A P-processor ideal parallel computer cannot run any faster than a machine with an unlimited number of processors. Looked at another way, a machine with an unlimited number of processors can emulate a P-processor machine by using just P of its processors. Thus, the ***span law*** follows:

$$T_P \geq T_\infty \ . \tag{27.3}$$

We define the ***speedup*** of a computation on P processors by the ratio T_1/T_P, which says how many times faster the computation is on P processors than on 1 processor. By the work law, we have $T_P \geq T_1/P$, which implies that $T_1/T_P \leq P$. Thus, the speedup on P processors can be at most P. When the speedup is linear in the number of processors, that is, when $T_1/T_P = \Theta(P)$, the computation exhibits ***linear speedup***, and when $T_1/T_P = P$, we have ***perfect linear speedup***.

The ratio T_1/T_∞ of the work to the span gives the ***parallelism*** of the multithreaded computation. We can view the parallelism from three perspectives. As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path. As an upper bound, the parallelism gives the maximum possible speedup that can be achieved on any number of processors. Finally, and perhaps most important, the parallelism provides a limit on the possibility of attaining perfect linear speedup. Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup. To see this last point, suppose that $P > T_1/T_\infty$, in which case

the span law implies that the speedup satisfies $T_1/T_P \le T_1/T_\infty < P$. Moreover, if the number P of processors in the ideal parallel computer greatly exceeds the parallelism—that is, if $P \gg T_1/T_\infty$—then $T_1/T_P \ll P$, so that the speedup is much less than the number of processors. In other words, the more processors we use beyond the parallelism, the less perfect the speedup.

As an example, consider the computation P-FIB(4) in Figure 27.2, and assume that each strand takes unit time. Since the work is $T_1 = 17$ and the span is $T_\infty = 8$, the parallelism is $T_1/T_\infty = 17/8 = 2.125$. Consequently, achieving much more than double the speedup is impossible, no matter how many processors we employ to execute the computation. For larger input sizes, however, we shall see that P-FIB(n) exhibits substantial parallelism.

We define the ***(parallel) slackness*** of a multithreaded computation executed on an ideal parallel computer with P processors to be the ratio $(T_1/T_\infty)/P = T_1/(PT_\infty)$, which is the factor by which the parallelism of the computation exceeds the number of processors in the machine. Thus, if the slackness is less than 1, we cannot hope to achieve perfect linear speedup, because $T_1/(PT_\infty) < 1$ and the span law imply that the speedup on P processors satisfies $T_1/T_P \le T_1/T_\infty < P$. Indeed, as the slackness decreases from 1 toward 0, the speedup of the computation diverges further and further from perfect linear speedup. If the slackness is greater than 1, however, the work per processor is the limiting constraint. As we shall see, as the slackness increases from 1, a good scheduler can achieve closer and closer to perfect linear speedup.

Scheduling

Good performance depends on more than just minimizing the work and span. The strands must also be scheduled efficiently onto the processors of the parallel machine. Our multithreaded programming model provides no way to specify which strands to execute on which processors. Instead, we rely on the concurrency platform's scheduler to map the dynamically unfolding computation to individual processors. In practice, the scheduler maps the strands to static threads, and the operating system schedules the threads on the processors themselves, but this extra level of indirection is unnecessary for our understanding of scheduling. We can just imagine that the concurrency platform's scheduler maps strands to processors directly.

A multithreaded scheduler must schedule the computation with no advance knowledge of when strands will be spawned or when they will complete—it must operate ***on-line***. Moreover, a good scheduler operates in a distributed fashion, where the threads implementing the scheduler cooperate to load-balance the computation. Provably good on-line, distributed schedulers exist, but analyzing them is complicated.

Instead, to keep our analysis simple, we shall investigate an on-line ***centralized*** scheduler, which knows the global state of the computation at any given time. In particular, we shall analyze ***greedy schedulers***, which assign as many strands to processors as possible in each time step. If at least P strands are ready to execute during a time step, we say that the step is a ***complete step***, and a greedy scheduler assigns any P of the ready strands to processors. Otherwise, fewer than P strands are ready to execute, in which case we say that the step is an ***incomplete step***, and the scheduler assigns each ready strand to its own processor.

From the work law, the best running time we can hope for on P processors is $T_P = T_1/P$, and from the span law the best we can hope for is $T_P = T_\infty$. The following theorem shows that greedy scheduling is provably good in that it achieves the sum of these two lower bounds as an upper bound.

Theorem 27.1
On an ideal parallel computer with P processors, a greedy scheduler executes a multithreaded computation with work T_1 and span T_∞ in time

$$T_P \le T_1/P + T_\infty \; . \tag{27.4}$$

Proof We start by considering the complete steps. In each complete step, the P processors together perform a total of P work. Suppose for the purpose of contradiction that the number of complete steps is strictly greater than $\lfloor T_1/P \rfloor$. Then, the total work of the complete steps is at least

$$\begin{aligned} P \cdot (\lfloor T_1/P \rfloor + 1) &= P \lfloor T_1/P \rfloor + P \\ &= T_1 - (T_1 \bmod P) + P && \text{(by equation (3.8))} \\ &> T_1 && \text{(by inequality (3.9))} \; . \end{aligned}$$

Thus, we obtain the contradiction that the P processors would perform more work than the computation requires, which allows us to conclude that the number of complete steps is at most $\lfloor T_1/P \rfloor$.

Now, consider an incomplete step. Let G be the dag representing the entire computation, and without loss of generality, assume that each strand takes unit time. (We can replace each longer strand by a chain of unit-time strands.) Let G' be the subgraph of G that has yet to be executed at the start of the incomplete step, and let G'' be the subgraph remaining to be executed after the incomplete step. A longest path in a dag must necessarily start at a vertex with in-degree 0. Since an incomplete step of a greedy scheduler executes all strands with in-degree 0 in G', the length of a longest path in G'' must be 1 less than the length of a longest path in G'. In other words, an incomplete step decreases the span of the unexecuted dag by 1. Hence, the number of incomplete steps is at most T_∞.

Since each step is either complete or incomplete, the theorem follows. ■

The following corollary to Theorem 27.1 shows that a greedy scheduler always performs well.

Corollary 27.2
The running time T_P of any multithreaded computation scheduled by a greedy scheduler on an ideal parallel computer with P processors is within a factor of 2 of optimal.

Proof Let T_P^* be the running time produced by an optimal scheduler on a machine with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Since the work and span laws—inequalities (27.2) and (27.3)—give us $T_P^* \geq \max(T_1/P, T_\infty)$, Theorem 27.1 implies that

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2 \cdot \max(T_1/P, T_\infty) \\ &\leq 2T_P^* . \end{aligned}$$

■

The next corollary shows that, in fact, a greedy scheduler achieves near-perfect linear speedup on any multithreaded computation as the slackness grows.

Corollary 27.3
Let T_P be the running time of a multithreaded computation produced by a greedy scheduler on an ideal parallel computer with P processors, and let T_1 and T_∞ be the work and span of the computation, respectively. Then, if $P \ll T_1/T_\infty$, we have $T_P \approx T_1/P$, or equivalently, a speedup of approximately P.

Proof If we suppose that $P \ll T_1/T_\infty$, then we also have $T_\infty \ll T_1/P$, and hence Theorem 27.1 gives us $T_P \leq T_1/P + T_\infty \approx T_1/P$. Since the work law (27.2) dictates that $T_P \geq T_1/P$, we conclude that $T_P \approx T_1/P$, or equivalently, that the speedup is $T_1/T_P \approx P$. ■

The $\ll$ symbol denotes "much less," but how much is "much less"? As a rule of thumb, a slackness of at least 10—that is, 10 times more parallelism than processors—generally suffices to achieve good speedup. Then, the span term in the greedy bound, inequality (27.4), is less than 10% of the work-per-processor term, which is good enough for most engineering situations. For example, if a computation runs on only 10 or 100 processors, it doesn't make sense to value parallelism of, say 1,000,000 over parallelism of 10,000, even with the factor of 100 difference. As Problem 27-2 shows, sometimes by reducing extreme parallelism, we can obtain algorithms that are better with respect to other concerns and which still scale up well on reasonable numbers of processors.

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

(a)

Work: $T_1(A \cup B) = T_1(A) + T_1(B)$
Span: $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

(b)

Figure 27.3 The work and span of composed subcomputations. **(a)** When two subcomputations are joined in series, the work of the composition is the sum of their work, and the span of the composition is the sum of their spans. **(b)** When two subcomputations are joined in parallel, the work of the composition remains the sum of their work, but the span of the composition is only the maximum of their spans.

Analyzing multithreaded algorithms

We now have all the tools we need to analyze multithreaded algorithms and provide good bounds on their running times on various numbers of processors. Analyzing the work is relatively straightforward, since it amounts to nothing more than analyzing the running time of an ordinary serial algorithm—namely, the serialization of the multithreaded algorithm—which you should already be familiar with, since that is what most of this textbook is about! Analyzing the span is more interesting, but generally no harder once you get the hang of it. We shall investigate the basic ideas using the P-FIB program.

Analyzing the work $T_1(n)$ of P-FIB(n) poses no hurdles, because we've already done it. The original FIB procedure is essentially the serialization of P-FIB, and hence $T_1(n) = T(n) = \Theta(\phi^n)$ from equation (27.1).

Figure 27.3 illustrates how to analyze the span. If two subcomputations are joined in series, their spans add to form the span of their composition, whereas if they are joined in parallel, the span of their composition is the maximum of the spans of the two subcomputations. For P-FIB(n), the spawned call to P-FIB$(n-1)$ in line 3 runs in parallel with the call to P-FIB$(n-2)$ in line 4. Hence, we can express the span of P-FIB(n) as the recurrence

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) \,, \end{aligned}$$

which has solution $T_\infty(n) = \Theta(n)$.

The parallelism of P-FIB(n) is $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$, which grows dramatically as n gets large. Thus, on even the largest parallel computers, a modest

value for n suffices to achieve near perfect linear speedup for P-FIB(n), because this procedure exhibits considerable parallel slackness.

Parallel loops

Many algorithms contain loops all of whose iterations can operate in parallel. As we shall see, we can parallelize such loops using the **spawn** and **sync** keywords, but it is much more convenient to specify directly that the iterations of such loops can run concurrently. Our pseudocode provides this functionality via the **parallel** concurrency keyword, which precedes the **for** keyword in a **for** loop statement.

As an example, consider the problem of multiplying an $n \times n$ matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. The resulting n-vector $y = (y_i)$ is given by the equation

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \ ,$$

for $i = 1, 2, \ldots, n$. We can perform matrix-vector multiplication by computing all the entries of y in parallel as follows:

```
MAT-VEC(A, x)
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4       y_i = 0
5  parallel for i = 1 to n
6       for j = 1 to n
7            y_i = y_i + a_ij x_j
8  return y
```

In this code, the **parallel for** keywords in lines 3 and 5 indicate that the iterations of the respective loops may be run concurrently. A compiler can implement each **parallel for** loop as a divide-and-conquer subroutine using nested parallelism. For example, the **parallel for** loop in lines 5–7 can be implemented with the call MAT-VEC-MAIN-LOOP$(A, x, y, n, 1, n)$, where the compiler produces the auxiliary subroutine MAT-VEC-MAIN-LOOP as follows:

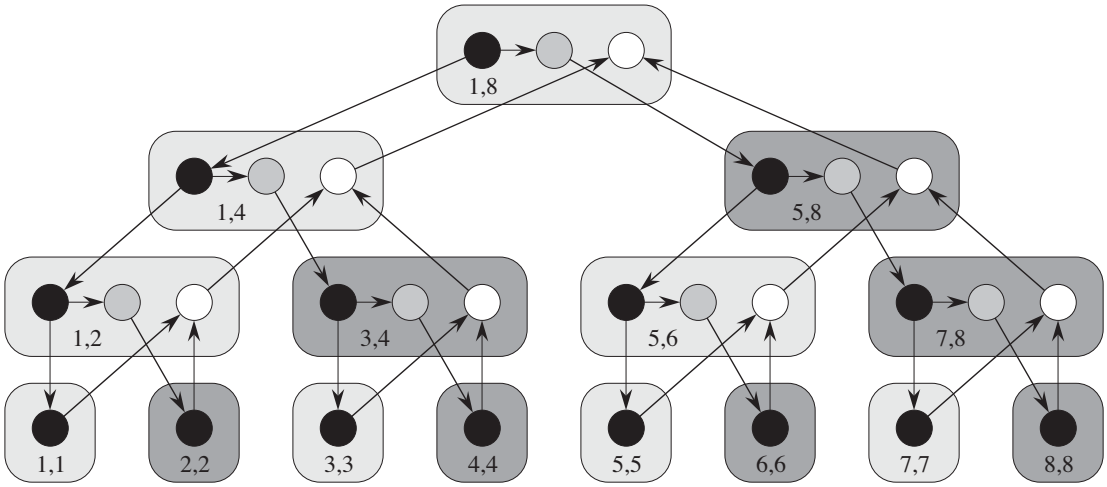


Figure 27.4 A dag representing the computation of MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn or call) of the procedure. The black circles represent strands corresponding to either the base case or the part of the procedure up to the spawn of MAT-VEC-MAIN-LOOP in line 5; the shaded circles represent strands corresponding to the part of the procedure that calls MAT-VEC-MAIN-LOOP in line 6 up to the **sync** in line 7, where it suspends until the spawned subroutine in line 5 returns; and the white circles represent strands corresponding to the (negligible) part of the procedure after the **sync** up to the point where it returns.

```
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1  if i == i'
2       for j = 1 to n
3            y_i = y_i + a_ij x_j
4  else mid = ⌊(i + i')/2⌋
5       spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6       MAT-VEC-MAIN-LOOP(A, x, y, n, mid + 1, i')
7       sync
```

This code recursively spawns the first half of the iterations of the loop to execute in parallel with the second half of the iterations and then executes a **sync**, thereby creating a binary tree of execution where the leaves are individual loop iterations, as shown in Figure 27.4.

To calculate the work $T_1(n)$ of MAT-VEC on an $n \times n$ matrix, we simply compute the running time of its serialization, which we obtain by replacing the **parallel for** loops with ordinary **for** loops. Thus, we have $T_1(n) = \Theta(n^2)$, because the quadratic running time of the doubly nested loops in lines 5–7 dominates. This analysis

seems to ignore the overhead for recursive spawning in implementing the parallel loops, however. In fact, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serialization, but not asymptotically. To see why, observe that since the tree of recursive procedure instances is a full binary tree, the number of internal nodes is 1 fewer than the number of leaves (see Exercise B.5-3). Each internal node performs constant work to divide the iteration range, and each leaf corresponds to an iteration of the loop, which takes at least constant time ($\Theta(n)$ time in this case). Thus, we can amortize the overhead of recursive spawning against the work of the iterations, contributing at most a constant factor to the overall work.

As a practical matter, dynamic-multithreading concurrency platforms sometimes ***coarsen*** the leaves of the recursion by executing several iterations in a single leaf, either automatically or under programmer control, thereby reducing the overhead of recursive spawning. This reduced overhead comes at the expense of also reducing the parallelism, however, but if the computation has sufficient parallel slackness, near-perfect linear speedup need not be sacrificed.

We must also account for the overhead of recursive spawning when analyzing the span of a parallel-loop construct. Since the depth of recursive calling is logarithmic in the number of iterations, for a parallel loop with n iterations in which the ith iteration has span $iter_\infty(i)$, the span is

$$T_\infty(n) = \Theta(\lg n) + \max_{1 \le i \le n} iter_\infty(i) \; .$$

For example, for MAT-VEC on an $n \times n$ matrix, the parallel initialization loop in lines 3–4 has span $\Theta(\lg n)$, because the recursive spawning dominates the constant-time work of each iteration. The span of the doubly nested loops in lines 5–7 is $\Theta(n)$, because each iteration of the outer **parallel for** loop contains n iterations of the inner (serial) **for** loop. The span of the remaining code in the procedure is constant, and thus the span is dominated by the doubly nested loops, yielding an overall span of $\Theta(n)$ for the whole procedure. Since the work is $\Theta(n^2)$, the parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$. (Exercise 27.1-6 asks you to provide an implementation with even more parallelism.)

Race conditions

A multithreaded algorithm is ***deterministic*** if it always does the same thing on the same input, no matter how the instructions are scheduled on the multicore computer. It is ***nondeterministic*** if its behavior might vary from run to run. Often, a multithreaded algorithm that is intended to be deterministic fails to be, because it contains a "determinacy race."

Race conditions are the bane of concurrency. Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured sev-

eral others, and the North American Blackout of 2003, which left over 50 million people without power. These pernicious bugs are notoriously hard to find. You can run tests in the lab for days without a failure only to discover that your software sporadically crashes in the field.

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write. The following procedure illustrates a race condition:

```
RACE-EXAMPLE( )
1  x = 0
2  parallel for i = 1 to 2
3       x = x + 1
4  print x
```

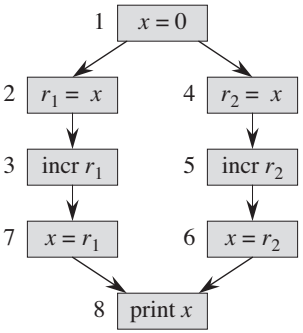
After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments x in line 3. Although it might seem that RACE-EXAMPLE should always print the value 2 (its serialization certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

When a processor increments x, the operation is not indivisible, but is composed of a sequence of instructions:

1. Read x from memory into one of the processor's registers.
2. Increment the value in the register.
3. Write the value in the register back into x in memory.

Figure 27.5(a) illustrates a computation dag representing the execution of RACE-EXAMPLE, with the strands broken down to individual instructions. Recall that since an ideal parallel computer supports sequential consistency, we can view the parallel execution of a multithreaded algorithm as an interleaving of instructions that respects the dependencies in the dag. Part (b) of the figure shows the values in an execution of the computation that elicits the anomaly. The value x is stored in memory, and r_1 and r_2 are processor registers. In step 1, one of the processors sets x to 0. In steps 2 and 3, processor 1 reads x from memory into its register r_1 and increments it, producing the value 1 in r_1. At that point, processor 2 comes into the picture, executing instructions 4–6. Processor 2 reads x from memory into register r_2; increments it, producing the value 1 in r_2; and then stores this value into x, setting x to 1. Now, processor 1 resumes with step 7, storing the value 1 in r_1 into x, which leaves the value of x unchanged. Therefore, step 8 prints the value 1, rather than 2, as the serialization would print.

We can see what has happened. If the effect of the parallel execution were that processor 1 executed all its instructions before processor 2, the value 2 would be



(a)

step	x	r_1	r_2
1	0	–	–
2	0	0	–
3	0	1	–
4	0	1	0
5	0	1	1
6	1	1	1
7	1	1	1

(b)

Figure 27.5 Illustration of the determinacy race in RACE-EXAMPLE. **(a)** A computation dag showing the dependencies among individual instructions. The processor registers are r_1 and r_2. Instructions unrelated to the race, such as the implementation of loop control, are omitted. **(b)** An execution sequence that elicits the bug, showing the values of x in memory and registers r_1 and r_2 for each step in the execution sequence.

printed. Conversely, if the effect were that processor 2 executed all its instructions before processor 1, the value 2 would still be printed. When the instructions of the two processors execute at the same time, however, it is possible, as in this example execution, that one of the updates to x is lost.

Of course, many executions do not elicit the bug. For example, if the execution order were $\langle 1, 2, 3, 7, 4, 5, 6, 8\rangle$ or $\langle 1, 4, 5, 6, 2, 3, 7, 8\rangle$, we would get the correct result. That's the problem with determinacy races. Generally, most orderings produce correct results—such as any in which the instructions on the left execute before the instructions on the right, or vice versa. But some orderings generate improper results when the instructions interleave. Consequently, races can be extremely hard to test for. You can run tests for days and never see the bug, only to experience a catastrophic system crash in the field when the outcome is critical.

Although we can cope with races in a variety of ways, including using mutual-exclusion locks and other methods of synchronization, for our purposes, we shall simply ensure that strands that operate in parallel are ***independent***: they have no determinacy races among them. Thus, in a **parallel for** construct, all the iterations should be independent. Between a **spawn** and the corresponding **sync**, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children. Note that arguments to a spawned child are evaluated in the parent before the actual spawn occurs, and thus the evaluation of arguments to a spawned subroutine is in series with any accesses to those arguments after the spawn.

As an example of how easy it is to generate code with races, here is a faulty implementation of multithreaded matrix-vector multiplication that achieves a span of $\Theta(\lg n)$ by parallelizing the inner **for** loop:

```
MAT-VEC-WRONG(A, x)
1  n = A.rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4       y_i = 0
5  parallel for i = 1 to n
6       parallel for j = 1 to n
7            y_i = y_i + a_ij x_j
8  return y
```

This procedure is, unfortunately, incorrect due to races on updating y_i in line 7, which executes concurrently for all n values of j. Exercise 27.1-6 asks you to give a correct implementation with $\Theta(\lg n)$ span.

A multithreaded algorithm with races can sometimes be correct. As an example, two parallel threads might store the same value into a shared variable, and it wouldn't matter which stored the value first. Generally, however, we shall consider code with races to be illegal.

A chess lesson

We close this section with a true story that occurred during the development of the world-class multithreaded chess-playing program ⋆Socrates [80], although the timings below have been simplified for exposition. The program was prototyped on a 32-processor computer but was ultimately to run on a supercomputer with 512 processors. At one point, the developers incorporated an optimization into the program that reduced its running time on an important benchmark on the 32-processor machine from $T_{32} = 65$ seconds to $T'_{32} = 40$ seconds. Yet, the developers used the work and span performance measures to conclude that the optimized version, which was faster on 32 processors, would actually be slower than the original version on 512 processsors. As a result, they abandoned the "optimization."

Here is their analysis. The original version of the program had work $T_1 = 2048$ seconds and span $T_\infty = 1$ second. If we treat inequality (27.4) as an equation, $T_P = T_1/P + T_\infty$, and use it as an approximation to the running time on P processors, we see that indeed $T_{32} = 2048/32 + 1 = 65$. With the optimization, the work became $T'_1 = 1024$ seconds and the span became $T'_\infty = 8$ seconds. Again using our approximation, we get $T'_{32} = 1024/32 + 8 = 40$.

The relative speeds of the two versions switch when we calculate the running times on 512 processors, however. In particular, we have $T_{512} = 2048/512 + 1 = 5$

seconds, and $T'_{512} = 1024/512 + 8 = 10$ seconds. The optimization that sped up the program on 32 processors would have made the program twice as slow on 512 processors! The optimized version's span of 8, which was not the dominant term in the running time on 32 processors, became the dominant term on 512 processors, nullifying the advantage from using more processors.

The moral of the story is that work and span can provide a better means of extrapolating performance than can measured running times.

Exercises

27.1-1

Suppose that we spawn P-FIB$(n-2)$ in line 4 of P-FIB, rather than calling it as is done in the code. What is the impact on the asymptotic work, span, and parallelism?

27.1-2

Draw the computation dag that results from executing P-FIB(5). Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation? Show how to schedule the dag on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

27.1-3

Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proven in Theorem 27.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \, . \tag{27.5}$$

27.1-4

Construct a computation dag for which one execution of a greedy scheduler can take nearly twice the time of another execution of a greedy scheduler on the same number of processors. Describe how the two executions would proceed.

27.1-5

Professor Karan measures her deterministic multithreaded algorithm on 4, 10, and 64 processors of an ideal parallel computer using a greedy scheduler. She claims that the three runs yielded $T_4 = 80$ seconds, $T_{10} = 42$ seconds, and $T_{64} = 10$ seconds. Argue that the professor is either lying or incompetent. (*Hint:* Use the work law (27.2), the span law (27.3), and inequality (27.5) from Exercise 27.1-3.)

27.1-6
Give a multithreaded algorithm to multiply an $n \times n$ matrix by an n-vector that achieves $\Theta(n^2/\lg n)$ parallelism while maintaining $\Theta(n^2)$ work.

27.1-7
Consider the following multithreaded pseudocode for transposing an $n \times n$ matrix A in place:

```
P-TRANSPOSE(A)
1  n = A.rows
2  parallel for j = 2 to n
3      parallel for i = 1 to j − 1
4          exchange a_ij with a_ji
```

Analyze the work, span, and parallelism of this algorithm.

27.1-8
Suppose that we replace the **parallel for** loop in line 3 of P-TRANSPOSE (see Exercise 27.1-7) with an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

27.1-9
For how many processors do the two versions of the chess programs run equally fast, assuming that $T_P = T_1/P + T_\infty$?

27.2 Multithreaded matrix multiplication

In this section, we examine how to multithread matrix multiplication, a problem whose serial running time we studied in Section 4.2. We'll look at multithreaded algorithms based on the standard triply nested loop, as well as divide-and-conquer algorithms.

Multithreaded matrix multiplication

The first algorithm we study is the straighforward algorithm based on parallelizing the loops in the procedure SQUARE-MATRIX-MULTIPLY on page 75:

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  parallel for i = 1 to n
4      parallel for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

To analyze this algorithm, observe that since the serialization of the algorithm is just SQUARE-MATRIX-MULTIPLY, the work is therefore simply $T_1(n) = \Theta(n^3)$, the same as the running time of SQUARE-MATRIX-MULTIPLY. The span is $T_\infty(n) = \Theta(n)$, because it follows a path down the tree of recursion for the **parallel for** loop starting in line 3, then down the tree of recursion for the **parallel for** loop starting in line 4, and then executes all n iterations of the ordinary **for** loop starting in line 6, resulting in a total span of $\Theta(\lg n) + \Theta(\lg n) + \Theta(n) = \Theta(n)$. Thus, the parallelism is $\Theta(n^3)/\Theta(n) = \Theta(n^2)$. Exercise 27.2-3 asks you to parallelize the inner loop to obtain a parallelism of $\Theta(n^3/\lg n)$, which you cannot do straightforwardly using **parallel for**, because you would create races.

A divide-and-conquer multithreaded algorithm for matrix multiplication

As we learned in Section 4.2, we can multiply $n \times n$ matrices serially in time $\Theta(n^{\lg 7}) = O(n^{2.81})$ using Strassen's divide-and-conquer strategy, which motivates us to look at multithreading such an algorithm. We begin, as we did in Section 4.2, with multithreading a simpler divide-and-conquer algorithm.

Recall from page 77 that the SQUARE-MATRIX-MULTIPLY-RECURSIVE procedure, which multiplies two $n \times n$ matrices A and B to produce the $n \times n$ matrix C, relies on partitioning each of the three matrices into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Then, we can write the matrix product as

$$\begin{aligned} \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}. \end{aligned} \tag{27.6}$$

Thus, to multiply two $n \times n$ matrices, we perform eight multiplications of $n/2 \times n/2$ matrices and one addition of $n \times n$ matrices. The following pseudocode implements

this divide-and-conquer strategy using nested parallelism. Unlike the SQUARE-MATRIX-MULTIPLY-RECURSIVE procedure on which it is based, P-MATRIX-MULTIPLY-RECURSIVE takes the output matrix as a parameter to avoid allocating matrices unnecessarily.

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
 1  n = A.rows
 2  if n == 1
 3      c_11 = a_11 b_11
 4  else let T be a new n × n matrix
 5      partition A, B, C, and T into n/2 × n/2 submatrices
            A_11, A_12, A_21, A_22; B_11, B_12, B_21, B_22; C_11, C_12, C_21, C_22;
            and T_11, T_12, T_21, T_22; respectively
 6      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_11, A_11, B_11)
 7      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_12, A_11, B_12)
 8      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_21, A_21, B_11)
 9      spawn P-MATRIX-MULTIPLY-RECURSIVE(C_22, A_21, B_12)
10      spawn P-MATRIX-MULTIPLY-RECURSIVE(T_11, A_12, B_21)
11      spawn P-MATRIX-MULTIPLY-RECURSIVE(T_12, A_12, B_22)
12      spawn P-MATRIX-MULTIPLY-RECURSIVE(T_21, A_22, B_21)
13      P-MATRIX-MULTIPLY-RECURSIVE(T_22, A_22, B_22)
14      sync
15      parallel for i = 1 to n
16          parallel for j = 1 to n
17              c_ij = c_ij + t_ij
```

Line 3 handles the base case, where we are multiplying 1×1 matrices. We handle the recursive case in lines 4–17. We allocate a temporary matrix T in line 4, and line 5 partitions each of the matrices A, B, C, and T into $n/2 \times n/2$ submatrices. (As with SQUARE-MATRIX-MULTIPLY-RECURSIVE on page 77, we gloss over the minor issue of how to use index calculations to represent submatrix sections of a matrix.) The recursive call in line 6 sets the submatrix C_{11} to the submatrix product $A_{11}B_{11}$, so that C_{11} equals the first of the two terms that form its sum in equation (27.6). Similarly, lines 7–9 set C_{12}, C_{21}, and C_{22} to the first of the two terms that equal their sums in equation (27.6). Line 10 sets the submatrix T_{11} to the submatrix product $A_{12}B_{21}$, so that T_{11} equals the second of the two terms that form C_{11}'s sum. Lines 11–13 set T_{12}, T_{21}, and T_{22} to the second of the two terms that form the sums of C_{12}, C_{21}, and C_{22}, respectively. The first seven recursive calls are spawned, and the last one runs in the main strand. The **sync** statement in line 14 ensures that all the submatrix products in lines 6–13 have been computed,

after which we add the products from T into C in using the doubly nested **parallel for** loops in lines 15–17.

We first analyze the work $M_1(n)$ of the P-MATRIX-MULTIPLY-RECURSIVE procedure, echoing the serial running-time analysis of its progenitor SQUARE-MATRIX-MULTIPLY-RECURSIVE. In the recursive case, we partition in $\Theta(1)$ time, perform eight recursive multiplications of $n/2 \times n/2$ matrices, and finish up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus, the recurrence for the work $M_1(n)$ is

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

by case 1 of the master theorem. In other words, the work of our multithreaded algorithm is asymptotically the same as the running time of the procedure SQUARE-MATRIX-MULTIPLY in Section 4.2, with its triply nested loops.

To determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE, we first observe that the span for partitioning is $\Theta(1)$, which is dominated by the $\Theta(\lg n)$ span of the doubly nested **parallel for** loops in lines 15–17. Because the eight parallel recursive calls all execute on matrices of the same size, the maximum span for any recursive call is just the span of any one. Hence, the recurrence for the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE is

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \; . \tag{27.7}$$

This recurrence does not fall under any of the cases of the master theorem, but it does meet the condition of Exercise 4.6-2. By Exercise 4.6-2, therefore, the solution to recurrence (27.7) is $M_\infty(n) = \Theta(\lg^2 n)$.

Now that we know the work and span of P-MATRIX-MULTIPLY-RECURSIVE, we can compute its parallelism as $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is very high.

Multithreading Strassen's method

To multithread Strassen's algorithm, we follow the same general outline as on page 79, only using nested parallelism:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (27.6). This step takes $\Theta(1)$ work and span by index calculation.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span by using doubly nested **parallel for** loops.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively spawn the computation of seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$.
4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices, once again using doubly nested **parallel for** loops. We can compute all four submatrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

To analyze this algorithm, we first observe that since the serialization is the same as the original serial algorithm, the work is just the running time of the serialization, namely, $\Theta(n^{\lg 7})$. As for P-MATRIX-MULTIPLY-RECURSIVE, we can devise a recurrence for the span. In this case, seven recursive calls execute in parallel, but since they all operate on matrices of the same size, we obtain the same recurrence (27.7) as we did for P-MATRIX-MULTIPLY-RECURSIVE, which has solution $\Theta(\lg^2 n)$. Thus, the parallelism of multithreaded Strassen's method is $\Theta(n^{\lg 7}/\lg^2 n)$, which is high, though slightly less than the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

Exercises

27.2-1
Draw the computation dag for computing P-SQUARE-MATRIX-MULTIPLY on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Use the convention that spawn and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, analyze the work, span, and parallelism of this computation.

27.2-2
Repeat Exercise 27.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

27.2-3
Give pseudocode for a multithreaded algorithm that multiplies two $n \times n$ matrices with work $\Theta(n^3)$ but span only $\Theta(\lg n)$. Analyze your algorithm.

27.2-4
Give pseudocode for an efficient multithreaded algorithm that multiplies a $p \times q$ matrix by a $q \times r$ matrix. Your algorithm should be highly parallel even if any of p, q, and r are 1. Analyze your algorithm.

27.2-5
Give pseudocode for an efficient multithreaded algorithm that transposes an $n \times n$ matrix in place by using divide-and-conquer to divide the matrix recursively into four $n/2 \times n/2$ submatrices. Analyze your algorithm.

27.2-6
Give pseudocode for an efficient multithreaded implementation of the Floyd-Warshall algorithm (see Section 25.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

27.3 Multithreaded merge sort

We first saw serial merge sort in Section 2.3.1, and in Section 2.3.2 we analyzed its running time and showed it to be $\Theta(n \lg n)$. Because merge sort already uses the divide-and-conquer paradigm, it seems like a terrific candidate for multithreading using nested parallelism. We can easily modify the pseudocode so that the first recursive call is spawned:

```
MERGE-SORT'(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      spawn MERGE-SORT'(A, p, q)
4      MERGE-SORT'(A, q + 1, r)
5      sync
6      MERGE(A, p, q, r)
```

Like its serial counterpart, MERGE-SORT′ sorts the subarray $A[p..r]$. After the two recursive subroutines in lines 3 and 4 have completed, which is ensured by the **sync** statement in line 5, MERGE-SORT′ calls the same MERGE procedure as on page 31.

Let us analyze MERGE-SORT′. To do so, we first need to analyze MERGE. Recall that its serial running time to merge n elements is $\Theta(n)$. Because MERGE is serial, both its work and its span are $\Theta(n)$. Thus, the following recurrence characterizes the work $MS'_1(n)$ of MERGE-SORT′ on n elements:

$$\begin{aligned} MS'_1(n) &= 2\,MS'_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n)\,, \end{aligned}$$

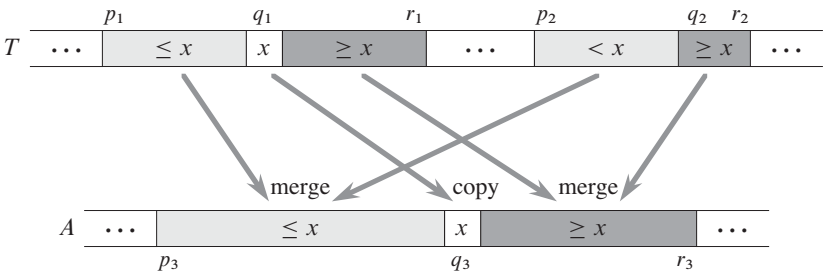


Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into the subarray $A[p_3 \,..\, r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \,..\, r_1]$ and q_2 be the place in $T[p_2 \,..\, r_2]$ such that x would fall between $T[q_2 - 1]$ and $T[q_2]$, every element in subarrays $T[p_1 \,..\, q_1 - 1]$ and $T[p_2 \,..\, q_2 - 1]$ (lightly shaded) is less than or equal to x, and every element in the subarrays $T[q_1 + 1 \,..\, r_1]$ and $T[q_2 + 1 \,..\, r_2]$ (heavily shaded) is at least x. To merge, we compute the index q_3 where x belongs in $A[p_3 \,..\, r_3]$, copy x into $A[q_3]$, and then recursively merge $T[p_1 \,..\, q_1 - 1]$ with $T[p_2 \,..\, q_2 - 1]$ into $A[p_3 \,..\, q_3 - 1]$ and $T[q_1 + 1 \,..\, r_1]$ with $T[q_2 \,..\, r_2]$ into $A[q_3 + 1 \,..\, r_3]$.

which is the same as the serial running time of merge sort. Since the two recursive calls of MERGE-SORT′ can run in parallel, the span MS'_∞ is given by the recurrence

$$\begin{aligned} MS'_\infty(n) &= MS'_\infty(n/2) + \Theta(n) \\ &= \Theta(n) \,. \end{aligned}$$

Thus, the parallelism of MERGE-SORT′ comes to $MS'_1(n)/MS'_\infty(n) = \Theta(\lg n)$, which is an unimpressive amount of parallelism. To sort 10 million elements, for example, it might achieve linear speedup on a few processors, but it would not scale up effectively to hundreds of processors.

You probably have already figured out where the parallelism bottleneck is in this multithreaded merge sort: the serial MERGE procedure. Although merging might initially seem to be inherently serial, we can, in fact, fashion a multithreaded version of it by using nested parallelism.

Our divide-and-conquer strategy for multithreaded merging, which is illustrated in Figure 27.6, operates on subarrays of an array T. Suppose that we are merging the two sorted subarrays $T[p_1 \,..\, r_1]$ of length $n_1 = r_1 - p_1 + 1$ and $T[p_2 \,..\, r_2]$ of length $n_2 = r_2 - p_2 + 1$ into another subarray $A[p_3 \,..\, r_3]$, of length $n_3 = r_3 - p_3 + 1 = n_1 + n_2$. Without loss of generality, we make the simplifying assumption that $n_1 \ge n_2$.

We first find the middle element $x = T[q_1]$ of the subarray $T[p_1 \,..\, r_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because the subarray is sorted, x is a median of $T[p_1 \,..\, r_1]$: every element in $T[p_1 \,..\, q_1 - 1]$ is no more than x, and every element in $T[q_1 + 1 \,..\, r_1]$ is no less than x. We then use binary search to find the

index q_2 in the subarray $T[p_2 \,..\, r_2]$ so that the subarray would still be sorted if we inserted x between $T[q_2 - 1]$ and $T[q_2]$.

We next merge the original subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into $A[p_3 \,..\, r_3]$ as follows:

1. Set $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$.
2. Copy x into $A[q_3]$.
3. Recursively merge $T[p_1 \,..\, q_1 - 1]$ with $T[p_2 \,..\, q_2 - 1]$, and place the result into the subarray $A[p_3 \,..\, q_3 - 1]$.
4. Recursively merge $T[q_1 + 1 \,..\, r_1]$ with $T[q_2 \,..\, r_2]$, and place the result into the subarray $A[q_3 + 1 \,..\, r_3]$.

When we compute q_3, the quantity $q_1 - p_1$ is the number of elements in the subarray $T[p_1 \,..\, q_1 - 1]$, and the quantity $q_2 - p_2$ is the number of elements in the subarray $T[p_2 \,..\, q_2 - 1]$. Thus, their sum is the number of elements that end up before x in the subarray $A[p_3 \,..\, r_3]$.

The base case occurs when $n_1 = n_2 = 0$, in which case we have no work to do to merge the two empty subarrays. Since we have assumed that the subarray $T[p_1 \,..\, r_1]$ is at least as long as $T[p_2 \,..\, r_2]$, that is, $n_1 \geq n_2$, we can check for the base case by just checking whether $n_1 = 0$. We must also ensure that the recursion properly handles the case when only one of the two subarrays is empty, which, by our assumption that $n_1 \geq n_2$, must be the subarray $T[p_2 \,..\, r_2]$.

Now, let's put these ideas into pseudocode. We start with the binary search, which we express serially. The procedure BINARY-SEARCH(x, T, p, r) takes a key x and a subarray $T[p \,..\, r]$, and it returns one of the following:

- If $T[p \,..\, r]$ is empty ($r < p$), then it returns the index p.
- If $x \leq T[p]$, and hence less than or equal to all the elements of $T[p \,..\, r]$, then it returns the index p.
- If $x > T[p]$, then it returns the largest index q in the range $p < q \leq r + 1$ such that $T[q - 1] < x$.

Here is the pseudocode:

```
BINARY-SEARCH(x, T, p, r)
1  low = p
2  high = max(p, r + 1)
3  while low < high
4      mid = ⌊(low + high)/2⌋
5      if x ≤ T[mid]
6          high = mid
7      else low = mid + 1
8  return high
```

The call BINARY-SEARCH(x, T, p, r) takes $\Theta(\lg n)$ serial time in the worst case, where $n = r - p + 1$ is the size of the subarray on which it runs. (See Exercise 2.3-5.) Since BINARY-SEARCH is a serial procedure, its worst-case work and span are both $\Theta(\lg n)$.

We are now prepared to write pseudocode for the multithreaded merging procedure itself. Like the MERGE procedure on page 31, the P-MERGE procedure assumes that the two subarrays to be merged lie within the same array. Unlike MERGE, however, P-MERGE does not assume that the two subarrays to be merged are adjacent within the array. (That is, P-MERGE does not require that $p_2 = r_1 + 1$.) Another difference between MERGE and P-MERGE is that P-MERGE takes as an argument an output subarray A into which the merged values should be stored. The call P-MERGE$(T, p_1, r_1, p_2, r_2, A, p_3)$ merges the sorted subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$ into the subarray $A[p_3 \,..\, r_3]$, where $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$ and is not provided as an input.

```
P-MERGE(T, p1, r1, p2, r2, A, p3)
 1  n1 = r1 − p1 + 1
 2  n2 = r2 − p2 + 1
 3  if n1 < n2                    // ensure that n1 ≥ n2
 4      exchange p1 with p2
 5      exchange r1 with r2
 6      exchange n1 with n2
 7  if n1 == 0                    // both empty?
 8      return
 9  else q1 = ⌊(p1 + r1)/2⌋
10      q2 = BINARY-SEARCH(T[q1], T, p2, r2)
11      q3 = p3 + (q1 − p1) + (q2 − p2)
12      A[q3] = T[q1]
13      spawn P-MERGE(T, p1, q1 − 1, p2, q2 − 1, A, p3)
14      P-MERGE(T, q1 + 1, r1, q2, r2, A, q3 + 1)
15      sync
```

The P-MERGE procedure works as follows. Lines 1–2 compute the lengths n_1 and n_2 of the subarrays $T[p_1 \,..\, r_1]$ and $T[p_2 \,..\, r_2]$, respectively. Lines 3–6 enforce the assumption that $n_1 \geq n_2$. Line 7 tests for the base case, where the subarray $T[p_1 \,..\, r_1]$ is empty (and hence so is $T[p_2 \,..\, r_2]$), in which case we simply return. Lines 9–15 implement the divide-and-conquer strategy. Line 9 computes the midpoint of $T[p_1 \,..\, r_1]$, and line 10 finds the point q_2 in $T[p_2 \,..\, r_2]$ such that all elements in $T[p_2 \,..\, q_2 - 1]$ are less than $T[q_1]$ (which corresponds to x) and all the elements in $T[q_2 \,..\, p_2]$ are at least as large as $T[q_1]$. Line 11 com-

putes the index q_3 of the element that divides the output subarray $A[p_3 \,..\, r_3]$ into $A[p_3 \,..\, q_3 - 1]$ and $A[q_3 + 1 \,..\, r_3]$, and then line 12 copies $T[q_1]$ directly into $A[q_3]$.

Then, we recurse using nested parallelism. Line 13 spawns the first subproblem, while line 14 calls the second subproblem in parallel. The **sync** statement in line 15 ensures that the subproblems have completed before the procedure returns. (Since every procedure implicitly executes a **sync** before returning, we could have omitted the **sync** statement in line 15, but including it is good coding practice.) There is some cleverness in the coding to ensure that when the subarray $T[p_2 \,..\, r_2]$ is empty, the code operates correctly. The way it works is that on each recursive call, a median element of $T[p_1 \,..\, r_1]$ is placed into the output subarray, until $T[p_1 \,..\, r_1]$ itself finally becomes empty, triggering the base case.

Analysis of multithreaded merging

We first derive a recurrence for the span $PM_\infty(n)$ of P-MERGE, where the two subarrays contain a total of $n = n_1 + n_2$ elements. Because the spawn in line 13 and the call in line 14 operate logically in parallel, we need examine only the costlier of the two calls. The key is to understand that in the worst case, the maximum number of elements in either of the recursive calls can be at most $3n/4$, which we see as follows. Because lines 3–6 ensure that $n_2 \le n_1$, it follows that $n_2 = 2n_2/2 \le (n_1 + n_2)/2 = n/2$. In the worst case, one of the two recursive calls merges $\lfloor n_1/2 \rfloor$ elements of $T[p_1 \,..\, r_1]$ with all n_2 elements of $T[p_2 \,..\, r_2]$, and hence the number of elements involved in the call is

$$\begin{aligned} \lfloor n_1/2 \rfloor + n_2 &\le n_1/2 + n_2/2 + n_2/2 \\ &= (n_1 + n_2)/2 + n_2/2 \\ &\le n/2 + n/4 \\ &= 3n/4 \,. \end{aligned}$$

Adding in the $\Theta(\lg n)$ cost of the call to BINARY-SEARCH in line 10, we obtain the following recurrence for the worst-case span:

$$PM_\infty(n) = PM_\infty(3n/4) + \Theta(\lg n) \,. \tag{27.8}$$

(For the base case, the span is $\Theta(1)$, since lines 1–8 execute in constant time.) This recurrence does not fall under any of the cases of the master theorem, but it meets the condition of Exercise 4.6-2. Therefore, the solution to recurrence (27.8) is $PM_\infty(n) = \Theta(\lg^2 n)$.

We now analyze the work $PM_1(n)$ of P-MERGE on n elements, which turns out to be $\Theta(n)$. Since each of the n elements must be copied from array T to array A, we have $PM_1(n) = \Omega(n)$. Thus, it remains only to show that $PM_1(n) = O(n)$.

We shall first derive a recurrence for the worst-case work. The binary search in line 10 costs $\Theta(\lg n)$ in the worst case, which dominates the other work outside

of the recursive calls. For the recursive calls, observe that although the recursive calls in lines 13 and 14 might merge different numbers of elements, together the two recursive calls merge at most n elements (actually $n-1$ elements, since $T[q_1]$ does not participate in either recursive call). Moreover, as we saw in analyzing the span, a recursive call operates on at most $3n/4$ elements. We therefore obtain the recurrence

$$PM_1(n) = PM_1(\alpha n) + PM_1((1-\alpha)n) + O(\lg n) \,, \tag{27.9}$$

where α lies in the range $1/4 \le \alpha \le 3/4$, and where we understand that the actual value of α may vary for each level of recursion.

We prove that recurrence (27.9) has solution $PM_1 = O(n)$ via the substitution method. Assume that $PM_1(n) \le c_1 n - c_2 \lg n$ for some positive constants c_1 and c_2. Substituting gives us

$$\begin{aligned}
PM_1(n) &\le (c_1\alpha n - c_2 \lg(\alpha n)) + (c_1(1-\alpha)n - c_2 \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2 \lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2 \lg n \,,
\end{aligned}$$

since we can choose c_2 large enough that $c_2(\lg n + \lg(\alpha(1-\alpha)))$ dominates the $\Theta(\lg n)$ term. Furthermore, we can choose c_1 large enough to satisfy the base conditions of the recurrence. Since the work $PM_1(n)$ of P-MERGE is both $\Omega(n)$ and $O(n)$, we have $PM_1(n) = \Theta(n)$.

The parallelism of P-MERGE is $PM_1(n)/PM_\infty(n) = \Theta(n/\lg^2 n)$.

Multithreaded merge sort

Now that we have a nicely parallelized multithreaded merging procedure, we can incorporate it into a multithreaded merge sort. This version of merge sort is similar to the MERGE-SORT′ procedure we saw earlier, but unlike MERGE-SORT′, it takes as an argument an output subarray B, which will hold the sorted result. In particular, the call P-MERGE-SORT(A, p, r, B, s) sorts the elements in $A[p..r]$ and stores them in $B[s..s+r-p]$.

```
P-MERGE-SORT(A, p, r, B, s)
 1  n = r − p + 1
 2  if n == 1
 3       B[s] = A[p]
 4  else let T[1..n] be a new array
 5       q = ⌊(p + r)/2⌋
 6       q′ = q − p + 1
 7       spawn P-MERGE-SORT(A, p, q, T, 1)
 8       P-MERGE-SORT(A, q + 1, r, T, q′ + 1)
 9       sync
10       P-MERGE(T, 1, q′, q′ + 1, n, B, s)
```

After line 1 computes the number n of elements in the input subarray $A[p..r]$, lines 2–3 handle the base case when the array has only 1 element. Lines 4–6 set up for the recursive spawn in line 7 and call in line 8, which operate in parallel. In particular, line 4 allocates a temporary array T with n elements to store the results of the recursive merge sorting. Line 5 calculates the index q of $A[p..r]$ to divide the elements into the two subarrays $A[p..q]$ and $A[q+1..r]$ that will be sorted recursively, and line 6 goes on to compute the number q' of elements in the first subarray $A[p..q]$, which line 8 uses to determine the starting index in T of where to store the sorted result of $A[q+1..r]$. At that point, the spawn and recursive call are made, followed by the **sync** in line 9, which forces the procedure to wait until the spawned procedure is done. Finally, line 10 calls P-MERGE to merge the sorted subarrays, now in $T[1..q']$ and $T[q'+1..n]$, into the output subarray $B[s..s+r-p]$.

Analysis of multithreaded merge sort

We start by analyzing the work $PMS_1(n)$ of P-MERGE-SORT, which is considerably easier than analyzing the work of P-MERGE. Indeed, the work is given by the recurrence

$$\begin{aligned} PMS_1(n) &= 2\,PMS_1(n/2) + PM_1(n) \\ &= 2\,PMS_1(n/2) + \Theta(n)\,. \end{aligned}$$

This recurrence is the same as the recurrence (4.4) for ordinary MERGE-SORT from Section 2.3.1 and has solution $PMS_1(n) = \Theta(n \lg n)$ by case 2 of the master theorem.

We now derive and analyze a recurrence for the worst-case span $PMS_\infty(n)$. Because the two recursive calls to P-MERGE-SORT on lines 7 and 8 operate logically in parallel, we can ignore one of them, obtaining the recurrence

$$\begin{aligned} PMS_\infty(n) &= PMS_\infty(n/2) + PM_\infty(n) \\ &= PMS_\infty(n/2) + \Theta(\lg^2 n) \, . \end{aligned} \tag{27.10}$$

As for recurrence (27.8), the master theorem does not apply to recurrence (27.10), but Exercise 4.6-2 does. The solution is $PMS_\infty(n) = \Theta(\lg^3 n)$, and so the span of P-MERGE-SORT is $\Theta(\lg^3 n)$.

Parallel merging gives P-MERGE-SORT a significant parallelism advantage over MERGE-SORT′. Recall that the parallelism of MERGE-SORT′, which calls the serial MERGE procedure, is only $\Theta(\lg n)$. For P-MERGE-SORT, the parallelism is

$$\begin{aligned} PMS_1(n)/PMS_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) \, , \end{aligned}$$

which is much better both in theory and in practice. A good implementation in practice would sacrifice some parallelism by coarsening the base case in order to reduce the constants hidden by the asymptotic notation. The straightforward way to coarsen the base case is to switch to an ordinary serial sort, perhaps quicksort, when the size of the array is sufficiently small.

Exercises

27.3-1
Explain how to coarsen the base case of P-MERGE.

27.3-2
Instead of finding a median element in the larger subarray, as P-MERGE does, consider a variant that finds a median element of all the elements in the two sorted subarrays using the result of Exercise 9.3-8. Give pseudocode for an efficient multithreaded merging procedure that uses this median-finding procedure. Analyze your algorithm.

27.3-3
Give an efficient multithreaded algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 171. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (*Hint:* You may need an auxiliary array and may need to make more than one pass over the input elements.)

27.3-4
Give a multithreaded version of RECURSIVE-FFT on page 911. Make your implementation as parallel as possible. Analyze your algorithm.

27.3-5 ⋆
Give a multithreaded version of RANDOMIZED-SELECT on page 216. Make your implementation as parallel as possible. Analyze your algorithm. (*Hint:* Use the partitioning algorithm from Exercise 27.3-3.)

27.3-6 ⋆
Show how to multithread SELECT from Section 9.3. Make your implementation as parallel as possible. Analyze your algorithm.

Problems

27-1 Implementing parallel loops using nested parallelism
Consider the following multithreaded algorithm for performing pairwise addition on n-element arrays $A[1..n]$ and $B[1..n]$, storing the sums in $C[1..n]$:

```
SUM-ARRAYS(A, B, C)
1  parallel for i = 1 to A.length
2      C[i] = A[i] + B[i]
```

a. Rewrite the parallel loop in SUM-ARRAYS using nested parallelism (**spawn** and **sync**) in the manner of MAT-VEC-MAIN-LOOP. Analyze the parallelism of your implementation.

Consider the following alternative implementation of the parallel loop, which contains a value *grain-size* to be specified:

```
SUM-ARRAYS'(A, B, C)
1  n = A.length
2  grain-size = ?                 // to be determined
3  r = ⌈n/grain-size⌉
4  for k = 0 to r − 1
5      spawn ADD-SUBARRAY(A, B, C, k · grain-size + 1,
                                   min((k + 1) · grain-size, n))
6  sync

ADD-SUBARRAY(A, B, C, i, j)
1  for k = i to j
2      C[k] = A[k] + B[k]
```

b. Suppose that we set *grain-size* $= 1$. What is the parallelism of this implementation?

c. Give a formula for the span of SUM-ARRAYS′ in terms of n and *grain-size*. Derive the best value for *grain-size* to maximize parallelism.

27-2 Saving temporary space in matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure has the disadvantage that it must allocate a temporary matrix T of size $n \times n$, which can adversely affect the constants hidden by the Θ-notation. The P-MATRIX-MULTIPLY-RECURSIVE procedure does have high parallelism, however. For example, ignoring the constants in the Θ-notation, the parallelism for multiplying 1000×1000 matrices comes to approximately $1000^3/10^2 = 10^7$, since $\lg 1000 \approx 10$. Most parallel computers have far fewer than 10 million processors.

a. Describe a recursive multithreaded algorithm that eliminates the need for the temporary matrix T at the cost of increasing the span to $\Theta(n)$. (*Hint:* Compute $C = C + AB$ following the general strategy of P-MATRIX-MULTIPLY-RECURSIVE, but initialize C in parallel and insert a **sync** in a judiciously chosen location.)

b. Give and solve recurrences for the work and span of your implementation.

c. Analyze the parallelism of your implementation. Ignoring the constants in the Θ-notation, estimate the parallelism on 1000×1000 matrices. Compare with the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

27-3 Multithreaded matrix algorithms

a. Parallelize the LU-DECOMPOSITION procedure on page 821 by giving pseudocode for a multithreaded version of this algorithm. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.

b. Do the same for LUP-DECOMPOSITION on page 824.

c. Do the same for LUP-SOLVE on page 817.

d. Do the same for a multithreaded algorithm based on equation (28.13) for inverting a symmetric positive-definite matrix.

27-4 Multithreading reductions and prefix computations

A ***⊗-reduction*** of an array $x[1..n]$, where $\otimes$ is an associative operator, is the value

$$y = x[1] \otimes x[2] \otimes \cdots \otimes x[n] .$$

The following procedure computes the $\otimes$-reduction of a subarray $x[i..j]$ serially.

```
REDUCE(x, i, j)
1  y = x[i]
2  for k = i + 1 to j
3       y = y ⊗ x[k]
4  return y
```

a. Use nested parallelism to implement a multithreaded algorithm P-REDUCE, which performs the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span. Analyze your algorithm.

A related problem is that of computing a ***⊗-prefix computation***, sometimes called a ***⊗-scan***, on an array $x[1..n]$, where $\otimes$ is once again an associative operator. The $\otimes$-scan produces the array $y[1..n]$ given by

$$\begin{aligned}
y[1] &= x[1] , \\
y[2] &= x[1] \otimes x[2] , \\
y[3] &= x[1] \otimes x[2] \otimes x[3] , \\
&\vdots \\
y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n] ,
\end{aligned}$$

that is, all prefixes of the array x "summed" using the $\otimes$ operator. The following serial procedure SCAN performs a $\otimes$-prefix computation:

```
SCAN(x)
1  n = x.length
2  let y[1..n] be a new array
3  y[1] = x[1]
4  for i = 2 to n
5       y[i] = y[i − 1] ⊗ x[i]
6  return y
```

Unfortunately, multithreading SCAN is not straightforward. For example, changing the **for** loop to a **parallel for** loop would create races, since each iteration of the loop body depends on the previous iteration. The following procedure P-SCAN-1 performs the $\otimes$-prefix computation in parallel, albeit inefficiently:

```
P-SCAN-1(x)
1  n = x.length
2  let y[1..n] be a new array
3  P-SCAN-1-AUX(x, y, 1, n)
4  return y

P-SCAN-1-AUX(x, y, i, j)
1  parallel for l = i to j
2       y[l] = P-REDUCE(x, 1, l)
```

b. Analyze the work, span, and parallelism of P-SCAN-1.

By using nested parallelism, we can obtain a more efficient $\otimes$-prefix computation:

```
P-SCAN-2(x)
1  n = x.length
2  let y[1..n] be a new array
3  P-SCAN-2-AUX(x, y, 1, n)
4  return y

P-SCAN-2-AUX(x, y, i, j)
1  if i == j
2       y[i] = x[i]
3  else k = ⌊(i + j)/2⌋
4       spawn P-SCAN-2-AUX(x, y, i, k)
5       P-SCAN-2-AUX(x, y, k + 1, j)
6       sync
7       parallel for l = k + 1 to j
8            y[l] = y[k] ⊗ y[l]
```

c. Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

We can improve on both P-SCAN-1 and P-SCAN-2 by performing the $\otimes$-prefix computation in two distinct passes over the data. On the first pass, we gather the terms for various contiguous subarrays of x into a temporary array t, and on the second pass we use the terms in t to compute the final result y. The following pseudocode implements this strategy, but certain expressions have been omitted:

```
P-SCAN-3(x)
1  n = x.length
2  let y[1..n] and t[1..n] be new arrays
3  y[1] = x[1]
4  if n > 1
5      P-SCAN-UP(x, t, 2, n)
6      P-SCAN-DOWN(x[1], x, t, y, 2, n)
7  return y

P-SCAN-UP(x, t, i, j)
1  if i == j
2      return x[i]
3  else
4      k = ⌊(i + j)/2⌋
5      t[k] = spawn P-SCAN-UP(x, t, i, k)
6      right = P-SCAN-UP(x, t, k + 1, j)
7      sync
8      return _____          // fill in the blank

P-SCAN-DOWN(v, x, t, y, i, j)
1  if i == j
2      y[i] = v ⊗ x[i]
3  else
4      k = ⌊(i + j)/2⌋
5      spawn P-SCAN-DOWN(_____, x, t, y, i, k)    // fill in the blank
6      P-SCAN-DOWN(_____, x, t, y, k + 1, j)      // fill in the blank
7      sync
```

d. Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6 of P-SCAN-DOWN. Argue that with expressions you supplied, P-SCAN-3 is correct. (*Hint:* Prove that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

e. Analyze the work, span, and parallelism of P-SCAN-3.

27-5 Multithreading a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries does not change during the computation and is called a ***stencil***. For example, Section 15.4 presents

a stencil algorithm to compute a longest common subsequence, where the value in entry $c[i, j]$ depends only on the values in $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, as well as the elements x_i and y_j within the two sequences given as inputs. The input sequences are fixed, but the algorithm fills in the two-dimensional array c so that it computes entry $c[i, j]$ after computing all three entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$.

In this problem, we examine how to use nested parallelism to multithread a simple stencil calculation on an $n \times n$ array A in which, of the values in A, the value placed into entry $A[i, j]$ depends only on values in $A[i', j']$, where $i' \le i$ and $j' \le j$ (and of course, $i' \ne i$ or $j' \ne j$). In other words, the value in an entry depends only on values in entries that are above it and/or to its left, along with static information outside of the array. Furthermore, we assume throughout this problem that once we have filled in the entries upon which $A[i, j]$ depends, we can fill in $A[i, j]$ in $\Theta(1)$ time (as in the LCS-LENGTH procedure of Section 15.4).

We can partition the $n \times n$ array A into four $n/2 \times n/2$ subarrays as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}. \tag{27.11}$$

Observe now that we can fill in subarray A_{11} recursively, since it does not depend on the entries of the other three subarrays. Once A_{11} is complete, we can continue to fill in A_{12} and A_{21} recursively in parallel, because although they both depend on A_{11}, they do not depend on each other. Finally, we can fill in A_{22} recursively.

a. Give multithreaded pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (27.11) and the discussion above. (Don't worry about the details of the base case, which depends on the specific stencil.) Give and solve recurrences for the work and span of this algorithm in terms of n. What is the parallelism?

b. Modify your solution to part (a) to divide an $n \times n$ array into nine $n/3 \times n/3$ subarrays, again recursing with as much parallelism as possible. Analyze this algorithm. How much more or less parallelism does this algorithm have compared with the algorithm from part (a)?

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer $b \ge 2$. Divide an $n \times n$ array into b^2 subarrays, each of size $n/b \times n/b$, recursing with as much parallelism as possible. In terms of n and b, what are the work, span, and parallelism of your algorithm? Argue that, using this approach, the parallelism must be $o(n)$ for any choice of $b \ge 2$. (*Hint:* For this last argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \ge 2$.)

d. Give pseudocode for a multithreaded algorithm for this simple stencil calculation that achieves $\Theta(n/\lg n)$ parallelism. Argue using notions of work and span that the problem, in fact, has $\Theta(n)$ inherent parallelism. As it turns out, the divide-and-conquer nature of our multithreaded pseudocode does not let us achieve this maximal parallelism.

27-6 Randomized multithreaded algorithms

Just as with ordinary serial algorithms, we sometimes want to implement randomized multithreaded algorithms. This problem explores how to adapt the various performance measures in order to handle the expected behavior of such algorithms. It also asks you to design and analyze a multithreaded algorithm for randomized quicksort.

a. Explain how to modify the work law (27.2), span law (27.3), and greedy scheduler bound (27.4) to work with expectations when T_P, T_1, and T_∞ are all random variables.

b. Consider a randomized multithreaded algorithm for which 1% of the time we have $T_1 = 10^4$ and $T_{10,000} = 1$, but for 99% of the time we have $T_1 = T_{10,000} = 10^9$. Argue that the ***speedup*** of a randomized multithreaded algorithm should be defined as $\mathrm{E}[T_1]/\mathrm{E}[T_P]$, rather than $\mathrm{E}[T_1/T_P]$.

c. Argue that the ***parallelism*** of a randomized multithreaded algorithm should be defined as the ratio $\mathrm{E}[T_1]/\mathrm{E}[T_\infty]$.

d. Multithread the RANDOMIZED-QUICKSORT algorithm on page 179 by using nested parallelism. (Do not parallelize RANDOMIZED-PARTITION.) Give the pseudocode for your P-RANDOMIZED-QUICKSORT algorithm.

e. Analyze your multithreaded algorithm for randomized quicksort. (*Hint:* Review the analysis of RANDOMIZED-SELECT on page 216.)

Chapter notes

Parallel computers, models for parallel computers, and algorithmic models for parallel programming have been around in various forms for years. Prior editions of this book included material on sorting networks and the PRAM (Parallel Random-Access Machine) model. The data-parallel model [48, 168] is another popular algorithmic programming model, which features operations on vectors and matrices as primitives.

Graham [149] and Brent [55] showed that there exist schedulers achieving the bound of Theorem 27.1. Eager, Zahorjan, and Lazowska [98] showed that any greedy scheduler achieves this bound and proposed the methodology of using work and span (although not by those names) to analyze parallel algorithms. Blelloch [47] developed an algorithmic programming model based on work and span (which he called the "depth" of the computation) for data-parallel programming. Blumofe and Leiserson [52] gave a distributed scheduling algorithm for dynamic multithreading based on randomized "work-stealing" and showed that it achieves the bound $\mathrm{E}[T_P] \leq T_1/P + O(T_\infty)$. Arora, Blumofe, and Plaxton [19] and Blelloch, Gibbons, and Matias [49] also provided provably good algorithms for scheduling dynamic multithreaded computations.

The multithreaded pseudocode and programming model were heavily influenced by the Cilk [51, 118] project at MIT and the Cilk++ [71] extensions to C++ distributed by Cilk Arts, Inc. Many of the multithreaded algorithms in this chapter appeared in unpublished lecture notes by C. E. Leiserson and H. Prokop and have been implemented in Cilk or Cilk++. The multithreaded merge-sorting algorithm was inspired by an algorithm of Akl [12].

The notion of sequential consistency is due to Lamport [223].

28 Matrix Operations

Because operations on matrices lie at the heart of scientific computing, efficient algorithms for working with matrices have many practical applications. This chapter focuses on how to multiply matrices and solve sets of simultaneous linear equations. Appendix D reviews the basics of matrices.

Section 28.1 shows how to solve a set of linear equations using LUP decompositions. Then, Section 28.2 explores the close relationship between multiplying and inverting matrices. Finally, Section 28.3 discusses the important class of symmetric positive-definite matrices and shows how we can use them to find a least-squares solution to an overdetermined set of linear equations.

One important issue that arises in practice is ***numerical stability***. Due to the limited precision of floating-point representations in actual computers, round-off errors in numerical computations may become amplified over the course of a computation, leading to incorrect results; we call such computations ***numerically unstable***. Although we shall briefly consider numerical stability on occasion, we do not focus on it in this chapter. We refer you to the excellent book by Golub and Van Loan [144] for a thorough discussion of stability issues.

28.1 Solving systems of linear equations

Numerous applications need to solve sets of simultaneous linear equations. We can formulate a linear system as a matrix equation in which each matrix or vector element belongs to a field, typically the real numbers $\mathbb{R}$. This section discusses how to solve a system of linear equations using a method called LUP decomposition.

We start with a set of linear equations in n unknowns $x_1, x_2, \ldots, x_n$:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 , \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 , \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n . \end{aligned} \tag{28.1}$$

A ***solution*** to the equations (28.1) is a set of values for $x_1, x_2, \ldots, x_n$ that satisfy all of the equations simultaneously. In this section, we treat only the case in which there are exactly n equations in n unknowns.

We can conveniently rewrite equations (28.1) as the matrix-vector equation

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

or, equivalently, letting $A = (a_{ij})$, $x = (x_i)$, and $b = (b_i)$, as

$$Ax = b . \tag{28.2}$$

If A is nonsingular, it possesses an inverse A^{-1}, and

$$x = A^{-1}b \tag{28.3}$$

is the solution vector. We can prove that x is the unique solution to equation (28.2) as follows. If there are two solutions, x and x', then $Ax = Ax' = b$ and, letting I denote an identity matrix,

$$\begin{aligned} x &= Ix \\ &= (A^{-1}A)x \\ &= A^{-1}(Ax) \\ &= A^{-1}(Ax') \\ &= (A^{-1}A)x' \\ &= x' . \end{aligned}$$

In this section, we shall be concerned predominantly with the case in which A is nonsingular or, equivalently (by Theorem D.1), the rank of A is equal to the number n of unknowns. There are other possibilities, however, which merit a brief discussion. If the number of equations is less than the number n of unknowns—or, more generally, if the rank of A is less than n—then the system is ***underdetermined***. An underdetermined system typically has infinitely many solutions, although it may have no solutions at all if the equations are inconsistent. If the number of equations exceeds the number n of unknowns, the system is ***overdetermined***, and there may not exist any solutions. Section 28.3 addresses the important

problem of finding good approximate solutions to overdetermined systems of linear equations.

Let us return to our problem of solving the system $Ax = b$ of n equations in n unknowns. We could compute A^{-1} and then, using equation (28.3), multiply b by A^{-1}, yielding $x = A^{-1}b$. This approach suffers in practice from numerical instability. Fortunately, another approach—LUP decomposition—is numerically stable and has the further advantage of being faster in practice.

Overview of LUP decomposition

The idea behind LUP decomposition is to find three $n \times n$ matrices L, U, and P such that

$$PA = LU \ , \tag{28.4}$$

where

- L is a unit lower-triangular matrix,
- U is an upper-triangular matrix, and
- P is a permutation matrix.

We call matrices L, U, and P satisfying equation (28.4) an ***LUP decomposition*** of the matrix A. We shall show that every nonsingular matrix A possesses such a decomposition.

Computing an LUP decomposition for the matrix A has the advantage that we can more easily solve linear systems when they are triangular, as is the case for both matrices L and U. Once we have found an LUP decomposition for A, we can solve equation (28.2), $Ax = b$, by solving only triangular linear systems, as follows. Multiplying both sides of $Ax = b$ by P yields the equivalent equation $PAx = Pb$, which, by Exercise D.1-4, amounts to permuting the equations (28.1). Using our decomposition (28.4), we obtain

$$LUx = Pb \ .$$

We can now solve this equation by solving two triangular linear systems. Let us define $y = Ux$, where x is the desired solution vector. First, we solve the lower-triangular system

$$Ly = Pb \tag{28.5}$$

for the unknown vector y by a method called "forward substitution." Having solved for y, we then solve the upper-triangular system

$$Ux = y \tag{28.6}$$

for the unknown x by a method called "back substitution." Because the permutation matrix P is invertible (Exercise D.2-3), multiplying both sides of equation (28.4) by P^{-1} gives $P^{-1}PA = P^{-1}LU$, so that

$$A = P^{-1}LU \; . \tag{28.7}$$

Hence, the vector x is our solution to $Ax = b$:

$$\begin{aligned} Ax &= P^{-1}LUx && \text{(by equation (28.7))} \\ &= P^{-1}Ly && \text{(by equation (28.6))} \\ &= P^{-1}Pb && \text{(by equation (28.5))} \\ &= b \; . \end{aligned}$$

Our next step is to show how forward and back substitution work and then attack the problem of computing the LUP decomposition itself.

Forward and back substitution

Forward substitution can solve the lower-triangular system (28.5) in $\Theta(n^2)$ time, given L, P, and b. For convenience, we represent the permutation P compactly by an array $\pi[1..n]$. For $i = 1, 2, \ldots, n$, the entry $\pi[i]$ indicates that $P_{i,\pi[i]} = 1$ and $P_{ij} = 0$ for $j \neq \pi[i]$. Thus, PA has $a_{\pi[i],j}$ in row i and column j, and Pb has $b_{\pi[i]}$ as its ith element. Since L is unit lower-triangular, we can rewrite equation (28.5) as

$$\begin{aligned} y_1 &= b_{\pi[1]} \; , \\ l_{21}y_1 + y_2 &= b_{\pi[2]} \; , \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi[3]} \; , \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \cdots + y_n &= b_{\pi[n]} \; . \end{aligned}$$

The first equation tells us that $y_1 = b_{\pi[1]}$. Knowing the value of y_1, we can substitute it into the second equation, yielding

$$y_2 = b_{\pi[2]} - l_{21}y_1 \; .$$

Now, we can substitute both y_1 and y_2 into the third equation, obtaining

$$y_3 = b_{\pi[3]} - (l_{31}y_1 + l_{32}y_2) \; .$$

In general, we substitute $y_1, y_2, \ldots, y_{i-1}$ "forward" into the ith equation to solve for y_i:

$$y_i = b_{\pi[i]} - \sum_{j=1}^{i-1} l_{ij} y_j \ .$$

Having solved for y, we solve for x in equation (28.6) using ***back substitution***, which is similar to forward substitution. Here, we solve the nth equation first and work backward to the first equation. Like forward substitution, this process runs in $\Theta(n^2)$ time. Since U is upper-triangular, we can rewrite the system (28.6) as

$$\begin{aligned}
u_{11}x_1 + u_{12}x_2 + \cdots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1 \ , \\
u_{22}x_2 + \cdots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2 \ , \\
&\vdots \\
u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2} \ , \\
u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1} \ , \\
u_{n,n}x_n &= y_n \ .
\end{aligned}$$

Thus, we can solve for $x_n, x_{n-1}, \ldots, x_1$ successively as follows:

$$\begin{aligned}
x_n &= y_n/u_{n,n} \ , \\
x_{n-1} &= (y_{n-1} - u_{n-1,n}x_n)/u_{n-1,n-1} \ , \\
x_{n-2} &= (y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n))/u_{n-2,n-2} \ , \\
&\vdots
\end{aligned}$$

or, in general,

$$x_i = \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii} \ .$$

Given P, L, U, and b, the procedure LUP-SOLVE solves for x by combining forward and back substitution. The pseudocode assumes that the dimension n appears in the attribute $L.rows$ and that the permutation matrix P is represented by the array π.

```
LUP-SOLVE(L, U, π, b)
1  n = L.rows
2  let x be a new vector of length n
3  for i = 1 to n
4       y_i = b_{π[i]} − Σ_{j=1}^{i−1} l_ij y_j
5  for i = n downto 1
6       x_i = (y_i − Σ_{j=i+1}^{n} u_ij x_j) / u_ii
7  return x
```

Procedure LUP-SOLVE solves for y using forward substitution in lines 3–4, and then it solves for x using backward substitution in lines 5–6. Since the summation within each of the **for** loops includes an implicit loop, the running time is $\Theta(n^2)$.

As an example of these methods, consider the system of linear equations defined by

$$\begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} x = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

where

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix},$$

$$b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix},$$

and we wish to solve for the unknown x. The LUP decomposition is

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix},$$

$$U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix},$$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(You might want to verify that $PA = LU$.) Using forward substitution, we solve $Ly = Pb$ for y:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 3 \\ 7 \end{pmatrix},$$

obtaining

$$y = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix}$$

by computing first y_1, then y_2, and finally y_3. Using back substitution, we solve $Ux = y$ for x:

$$\begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 1.4 \\ 1.5 \end{pmatrix},$$

thereby obtaining the desired answer

$$x = \begin{pmatrix} -1.4 \\ 2.2 \\ 0.6 \end{pmatrix}$$

by computing first x_3, then x_2, and finally x_1.

Computing an LU decomposition

We have now shown that if we can create an LUP decomposition for a nonsingular matrix A, then forward and back substitution can solve the system $Ax = b$ of linear equations. Now we show how to efficiently compute an LUP decomposition for A. We start with the case in which A is an $n \times n$ nonsingular matrix and P is absent (or, equivalently, $P = I_n$). In this case, we factor $A = LU$. We call the two matrices L and U an ***LU decomposition*** of A.

We use a process known as ***Gaussian elimination*** to create an LU decomposition. We start by subtracting multiples of the first equation from the other equations in order to remove the first variable from those equations. Then, we subtract multiples of the second equation from the third and subsequent equations so that now the first and second variables are removed from them. We continue this process until the system that remains has an upper-triangular form—in fact, it is the matrix U. The matrix L is made up of the row multipliers that cause variables to be eliminated.

Our algorithm to implement this strategy is recursive. We wish to construct an LU decomposition for an $n \times n$ nonsingular matrix A. If $n = 1$, then we are done, since we can choose $L = I_1$ and $U = A$. For $n > 1$, we break A into four parts:

$$\begin{aligned} A &= \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}, \end{aligned}$$

where v is a column $(n-1)$-vector, w^{T} is a row $(n-1)$-vector, and A' is an $(n-1) \times (n-1)$ matrix. Then, using matrix algebra (verify the equations by

simply multiplying through), we can factor A as

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} . \end{aligned} \tag{28.8}$$

The 0s in the first and second matrices of equation (28.8) are row and column $(n-1)$-vectors, respectively. The term vw^{T}/a_{11}, formed by taking the outer product of v and w and dividing each element of the result by a_{11}, is an $(n-1) \times (n-1)$ matrix, which conforms in size to the matrix A' from which it is subtracted. The resulting $(n-1) \times (n-1)$ matrix

$$A' - vw^{\mathrm{T}}/a_{11} \tag{28.9}$$

is called the ***Schur complement*** of A with respect to a_{11}.

We claim that if A is nonsingular, then the Schur complement is nonsingular, too. Why? Suppose that the Schur complement, which is $(n-1) \times (n-1)$, is singular. Then by Theorem D.1, it has row rank strictly less than $n-1$. Because the bottom $n-1$ entries in the first column of the matrix

$$\begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}$$

are all 0, the bottom $n-1$ rows of this matrix must have row rank strictly less than $n-1$. The row rank of the entire matrix, therefore, is strictly less than n. Applying Exercise D.2-8 to equation (28.8), A has rank strictly less than n, and from Theorem D.1 we derive the contradiction that A is singular.

Because the Schur complement is nonsingular, we can now recursively find an LU decomposition for it. Let us say that

$$A' - vw^{\mathrm{T}}/a_{11} = L'U' ,$$

where L' is unit lower-triangular and U' is upper-triangular. Then, using matrix algebra, we have

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU , \end{aligned}$$

thereby providing our LU decomposition. (Note that because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.)

Of course, if $a_{11} = 0$, this method doesn't work, because it divides by 0. It also doesn't work if the upper leftmost entry of the Schur complement $A' - vw^{\mathrm{T}}/a_{11}$ is 0, since we divide by it in the next step of the recursion. The elements by which we divide during LU decomposition are called ***pivots***, and they occupy the diagonal elements of the matrix U. The reason we include a permutation matrix P during LUP decomposition is that it allows us to avoid dividing by 0. When we use permutations to avoid division by 0 (or by small numbers, which would contribute to numerical instability), we are ***pivoting***.

An important class of matrices for which LU decomposition always works correctly is the class of symmetric positive-definite matrices. Such matrices require no pivoting, and thus we can employ the recursive strategy outlined above without fear of dividing by 0. We shall prove this result, as well as several others, in Section 28.3.

Our code for LU decomposition of a matrix A follows the recursive strategy, except that an iteration loop replaces the recursion. (This transformation is a standard optimization for a "tail-recursive" procedure—one whose last operation is a recursive call to itself. See Problem 7-4.) It assumes that the attribute $A.rows$ gives the dimension of A. We initialize the matrix U with 0s below the diagonal and matrix L with 1s on its diagonal and 0s above the diagonal.

LU-DECOMPOSITION(A)

```
 1  n = A.rows
 2  let L and U be new n × n matrices
 3  initialize U with 0s below the diagonal
 4  initialize L with 1s on the diagonal and 0s above the diagonal
 5  for k = 1 to n
 6      u_kk = a_kk
 7      for i = k + 1 to n
 8          l_ik = a_ik/u_kk          // l_ik holds v_i
 9          u_ki = a_ki               // u_ki holds w_i^T
10      for i = k + 1 to n
11          for j = k + 1 to n
12              a_ij = a_ij − l_ik u_kj
13  return L and U
```

The outer **for** loop beginning in line 5 iterates once for each recursive step. Within this loop, line 6 determines the pivot to be $u_{kk} = a_{kk}$. The **for** loop in lines 7–9 (which does not execute when $k = n$), uses the v and w^{T} vectors to update L and U. Line 8 determines the elements of the v vector, storing v_i in l_{ik}, and line 9 computes the elements of the w^{T} vector, storing w_i^{T} in u_{ki}. Finally, lines 10–12 compute the elements of the Schur complement and store them back into the ma-

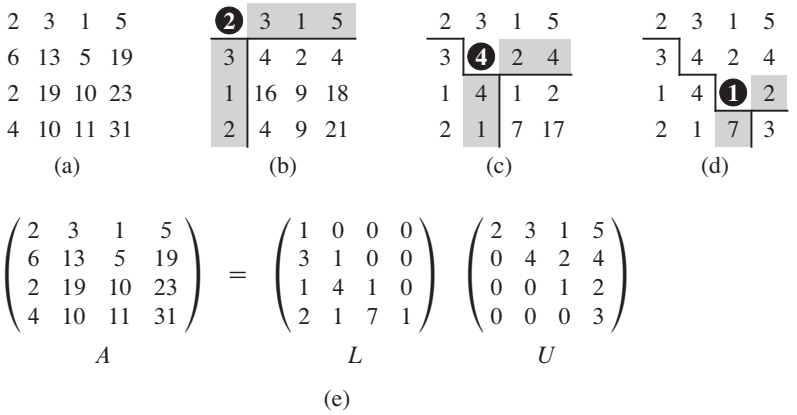


Figure 28.1 The operation of LU-DECOMPOSITION. **(a)** The matrix A. **(b)** The element $a_{11} = 2$ in the black circle is the pivot, the shaded column is v/a_{11}, and the shaded row is w^{T}. The elements of U computed thus far are above the horizontal line, and the elements of L are to the left of the vertical line. The Schur complement matrix $A' - vw^{\mathrm{T}}/a_{11}$ occupies the lower right. **(c)** We now operate on the Schur complement matrix produced from part (b). The element $a_{22} = 4$ in the black circle is the pivot, and the shaded column and row are v/a_{22} and w^{T} (in the partitioning of the Schur complement), respectively. Lines divide the matrix into the elements of U computed so far (above), the elements of L computed so far (left), and the new Schur complement (lower right). **(d)** After the next step, the matrix A is factored. (The element 3 in the new Schur complement becomes part of U when the recursion terminates.) **(e)** The factorization $A = LU$.

trix A. (We don't need to divide by a_{kk} in line 12 because we already did so when we computed l_{ik} in line 8.) Because line 12 is triply nested, LU-DECOMPOSITION runs in time $\Theta(n^3)$.

Figure 28.1 illustrates the operation of LU-DECOMPOSITION. It shows a standard optimization of the procedure in which we store the significant elements of L and U in place in the matrix A. That is, we can set up a correspondence between each element a_{ij} and either l_{ij} (if $i > j$) or u_{ij} (if $i \leq j$) and update the matrix A so that it holds both L and U when the procedure terminates. To obtain the pseudocode for this optimization from the above pseudocode, just replace each reference to l or u by a; you can easily verify that this transformation preserves correctness.

Computing an LUP decomposition

Generally, in solving a system of linear equations $Ax = b$, we must pivot on off-diagonal elements of A to avoid dividing by 0. Dividing by 0 would, of course, be disastrous. But we also want to avoid dividing by a small value—even if A is

nonsingular—because numerical instabilities can result. We therefore try to pivot on a large value.

The mathematics behind LUP decomposition is similar to that of LU decomposition. Recall that we are given an $n \times n$ nonsingular matrix A, and we wish to find a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before we partition the matrix A, as we did for LU decomposition, we move a nonzero element, say a_{k1}, from somewhere in the first column to the $(1,1)$ position of the matrix. For numerical stability, we choose a_{k1} as the element in the first column with the greatest absolute value. (The first column cannot contain only 0s, for then A would be singular, because its determinant would be 0, by Theorems D.4 and D.5.) In order to preserve the set of equations, we exchange row 1 with row k, which is equivalent to multiplying A by a permutation matrix Q on the left (Exercise D.1-4). Thus, we can write QA as

$$QA = \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix},$$

where $v = (a_{21}, a_{31}, \ldots, a_{n1})^{\mathrm{T}}$, except that a_{11} replaces a_{k1}; $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})$; and A' is an $(n-1) \times (n-1)$ matrix. Since $a_{k1} \neq 0$, we can now perform much the same linear algebra as for LU decomposition, but now guaranteeing that we do not divide by 0:

$$\begin{aligned} QA &= \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix}. \end{aligned}$$

As we saw for LU decomposition, if A is nonsingular, then the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ is nonsingular, too. Therefore, we can recursively find an LUP decomposition for it, with unit lower-triangular matrix L', upper-triangular matrix U', and permutation matrix P', such that

$$P'(A' - vw^{\mathrm{T}}/a_{k1}) = L'U' .$$

Define

$$P = \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} Q ,$$

which is a permutation matrix, since it is the product of two permutation matrices (Exercise D.1-4). We now have

$$
\begin{aligned}
PA &= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} QA \\
&= \begin{pmatrix} 1 & 0 \\ 0 & P' \end{pmatrix} \begin{pmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{k1} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & P'(A' - vw^{\mathrm{T}}/a_{k1}) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{pmatrix} \begin{pmatrix} a_{k1} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\
&= LU \ ,
\end{aligned}
$$

yielding the LUP decomposition. Because L' is unit lower-triangular, so is L, and because U' is upper-triangular, so is U.

Notice that in this derivation, unlike the one for LU decomposition, we must multiply both the column vector v/a_{k1} and the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ by the permutation matrix P'. Here is the pseudocode for LUP decomposition:

```
LUP-DECOMPOSITION(A)
 1  n = A.rows
 2  let π[1..n] be a new array
 3  for i = 1 to n
 4      π[i] = i
 5  for k = 1 to n
 6      p = 0
 7      for i = k to n
 8          if |a_ik| > p
 9              p = |a_ik|
10              k' = i
11      if p == 0
12          error "singular matrix"
13      exchange π[k] with π[k']
14      for i = 1 to n
15          exchange a_ki with a_k'i
16      for i = k + 1 to n
17          a_ik = a_ik/a_kk
18          for j = k + 1 to n
19              a_ij = a_ij − a_ik a_kj
```

Like LU-DECOMPOSITION, our LUP-DECOMPOSITION procedure replaces the recursion with an iteration loop. As an improvement over a direct implementation of the recursion, we dynamically maintain the permutation matrix P as an array π, where $\pi[i] = j$ means that the ith row of P contains a 1 in column j. We also implement the code to compute L and U "in place" in the matrix A. Thus, when the procedure terminates,

$$a_{ij} = \begin{cases} l_{ij} & \text{if } i > j \ , \\ u_{ij} & \text{if } i \le j \ . \end{cases}$$

Figure 28.2 illustrates how LUP-DECOMPOSITION factors a matrix. Lines 3–4 initialize the array π to represent the identity permutation. The outer **for** loop beginning in line 5 implements the recursion. Each time through the outer loop, lines 6–10 determine the element $a_{k'k}$ with largest absolute value of those in the current first column (column k) of the $(n - k + 1) \times (n - k + 1)$ matrix whose LUP decomposition we are finding. If all elements in the current first column are zero, lines 11–12 report that the matrix is singular. To pivot, we exchange $\pi[k']$ with $\pi[k]$ in line 13 and exchange the kth and k'th rows of A in lines 14–15, thereby making the pivot element a_{kk}. (The entire rows are swapped because in the derivation of the method above, not only is $A' - vw^{\mathrm{T}}/a_{k1}$ multiplied by P', but so is v/a_{k1}.) Finally, the Schur complement is computed by lines 16–19 in much the same way as it is computed by lines 7–12 of LU-DECOMPOSITION, except that here the operation is written to work in place.

Because of its triply nested loop structure, LUP-DECOMPOSITION has a running time of $\Theta(n^3)$, which is the same as that of LU-DECOMPOSITION. Thus, pivoting costs us at most a constant factor in time.

Exercises

28.1-1

Solve the equation

$$\begin{pmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ -6 & 5 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 3 \\ 14 \\ -7 \end{pmatrix}$$

by using forward substitution.

28.1-2

Find an LU decomposition of the matrix

$$\begin{pmatrix} 4 & -5 & 6 \\ 8 & -6 & 7 \\ 12 & -7 & 12 \end{pmatrix} .$$

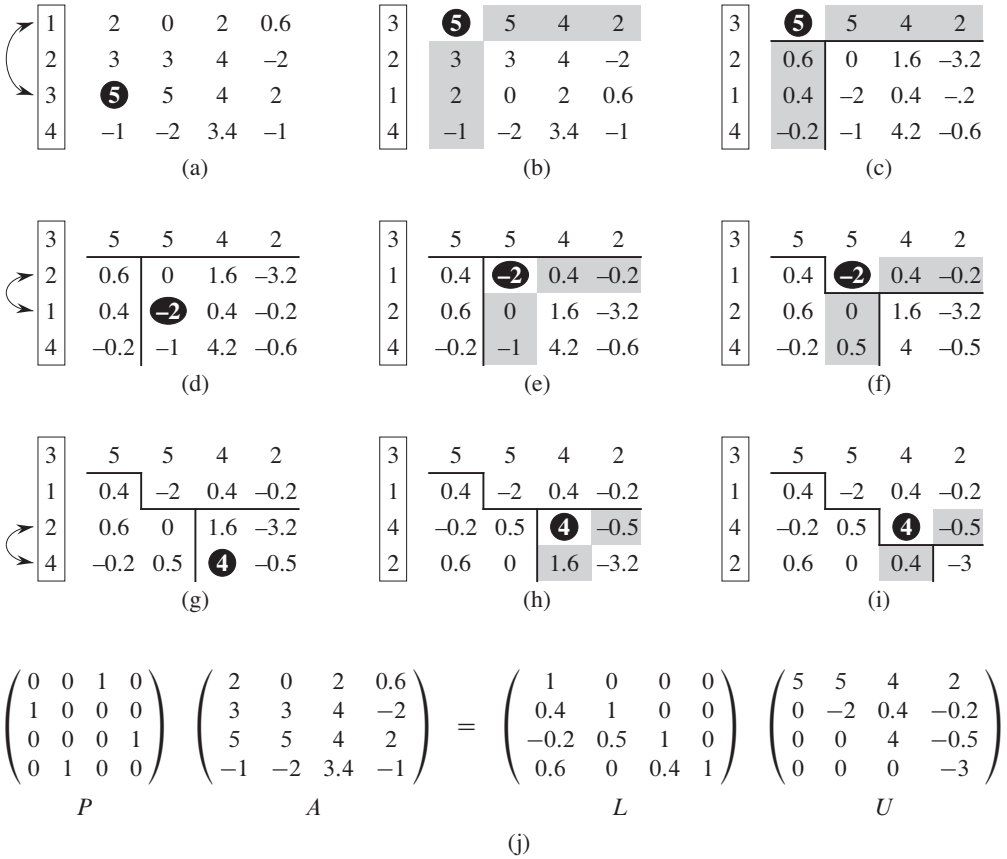


Figure 28.2 The operation of LUP-DECOMPOSITION. **(a)** The input matrix A with the identity permutation of the rows on the left. The first step of the algorithm determines that the element 5 in the black circle in the third row is the pivot for the first column. **(b)** Rows 1 and 3 are swapped and the permutation is updated. The shaded column and row represent v and w^T. **(c)** The vector v is replaced by $v/5$, and the lower right of the matrix is updated with the Schur complement. Lines divide the matrix into three regions: elements of U (above), elements of L (left), and elements of the Schur complement (lower right). **(d)–(f)** The second step. **(g)–(i)** The third step. No further changes occur on the fourth (final) step. **(j)** The LUP decomposition $PA = LU$.

28.1-3
Solve the equation

$$\begin{pmatrix} 1 & 5 & 4 \\ 2 & 0 & 3 \\ 5 & 8 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 12 \\ 9 \\ 5 \end{pmatrix}$$

by using an LUP decomposition.

28.1-4
Describe the LUP decomposition of a diagonal matrix.

28.1-5
Describe the LUP decomposition of a permutation matrix A, and prove that it is unique.

28.1-6
Show that for all $n \geq 1$, there exists a singular $n \times n$ matrix that has an LU decomposition.

28.1-7
In LU-DECOMPOSITION, is it necessary to perform the outermost **for** loop iteration when $k = n$? How about in LUP-DECOMPOSITION?

28.2 Inverting matrices

Although in practice we do not generally use matrix inverses to solve systems of linear equations, preferring instead to use more numerically stable techniques such as LUP decomposition, sometimes we need to compute a matrix inverse. In this section, we show how to use LUP decomposition to compute a matrix inverse. We also prove that matrix multiplication and computing the inverse of a matrix are equivalently hard problems, in that (subject to technical conditions) we can use an algorithm for one to solve the other in the same asymptotic running time. Thus, we can use Strassen's algorithm (see Section 4.2) for matrix multiplication to invert a matrix. Indeed, Strassen's original paper was motivated by the problem of showing that a set of a linear equations could be solved more quickly than by the usual method.

Computing a matrix inverse from an LUP decomposition

Suppose that we have an LUP decomposition of a matrix A in the form of three matrices L, U, and P such that $PA = LU$. Using LUP-SOLVE, we can solve an equation of the form $Ax = b$ in time $\Theta(n^2)$. Since the LUP decomposition depends on A but not b, we can run LUP-SOLVE on a second set of equations of the form $Ax = b'$ in additional time $\Theta(n^2)$. In general, once we have the LUP decomposition of A, we can solve, in time $\Theta(kn^2)$, k versions of the equation $Ax = b$ that differ only in b.

We can think of the equation

$$AX = I_n \,, \tag{28.10}$$

which defines the matrix X, the inverse of A, as a set of n distinct equations of the form $Ax = b$. To be precise, let X_i denote the ith column of X, and recall that the unit vector e_i is the ith column of I_n. We can then solve equation (28.10) for X by using the LUP decomposition for A to solve each equation

$$AX_i = e_i$$

separately for X_i. Once we have the LUP decomposition, we can compute each of the n columns X_i in time $\Theta(n^2)$, and so we can compute X from the LUP decomposition of A in time $\Theta(n^3)$. Since we can determine the LUP decomposition of A in time $\Theta(n^3)$, we can compute the inverse A^{-1} of a matrix A in time $\Theta(n^3)$.

Matrix multiplication and matrix inversion

We now show that the theoretical speedups obtained for matrix multiplication translate to speedups for matrix inversion. In fact, we prove something stronger: matrix inversion is equivalent to matrix multiplication, in the following sense. If $M(n)$ denotes the time to multiply two $n \times n$ matrices, then we can invert a nonsingular $n \times n$ matrix in time $O(M(n))$. Moreover, if $I(n)$ denotes the time to invert a nonsingular $n \times n$ matrix, then we can multiply two $n \times n$ matrices in time $O(I(n))$. We prove these results as two separate theorems.

Theorem 28.1 (Multiplication is no harder than inversion)
If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Proof Let A and B be $n \times n$ matrices whose matrix product C we wish to compute. We define the $3n \times 3n$ matrix D by

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix}.$$

The inverse of D is

$$D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix},$$

and thus we can compute the product AB by taking the upper right $n \times n$ submatrix of D^{-1}.

We can construct matrix D in $\Theta(n^2)$ time, which is $O(I(n))$ because we assume that $I(n) = \Omega(n^2)$, and we can invert D in $O(I(3n)) = O(I(n))$ time, by the regularity condition on $I(n)$. We thus have $M(n) = O(I(n))$. ■

Note that $I(n)$ satisfies the regularity condition whenever $I(n) = \Theta(n^c \lg^d n)$ for any constants $c > 0$ and $d \geq 0$.

The proof that matrix inversion is no harder than matrix multiplication relies on some properties of symmetric positive-definite matrices that we will prove in Section 28.3.

Theorem 28.2 (Inversion is no harder than multiplication)
Suppose we can multiply two $n \times n$ real matrices in time $M(n)$, where $M(n) = \Omega(n^2)$ and $M(n)$ satisfies the two regularity conditions $M(n+k) = O(M(n))$ for any k in the range $0 \leq k \leq n$ and $M(n/2) \leq cM(n)$ for some constant $c < 1/2$. Then we can compute the inverse of any real nonsingular $n \times n$ matrix in time $O(M(n))$.

Proof We prove the theorem here for real matrices. Exercise 28.2-6 asks you to generalize the proof for matrices whose entries are complex numbers.

We can assume that n is an exact power of 2, since we have

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}$$

for any $k > 0$. Thus, by choosing k such that $n + k$ is a power of 2, we enlarge the matrix to a size that is the next power of 2 and obtain the desired answer A^{-1} from the answer to the enlarged problem. The first regularity condition on $M(n)$ ensures that this enlargement does not cause the running time to increase by more than a constant factor.

For the moment, let us assume that the $n \times n$ matrix A is symmetric and positive-definite. We partition each of A and its inverse A^{-1} into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} \quad \text{and} \quad A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix}. \tag{28.11}$$

Then, if we let

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.12}$$

be the Schur complement of A with respect to B (we shall see more about this form of Schur complement in Section 28.3), we have

$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \tag{28.13}$$

since $AA^{-1} = I_n$, as you can verify by performing the matrix multiplication. Because A is symmetric and positive-definite, Lemmas 28.4 and 28.5 in Section 28.3 imply that B and S are both symmetric and positive-definite. By Lemma 28.3 in Section 28.3, therefore, the inverses B^{-1} and S^{-1} exist, and by Exercise D.2-6, B^{-1} and S^{-1} are symmetric, so that $(B^{-1})^{\mathrm{T}} = B^{-1}$ and $(S^{-1})^{\mathrm{T}} = S^{-1}$. Therefore, we can compute the submatrices R, T, U, and V of A^{-1} as follows, where all matrices mentioned are $n/2 \times n/2$:

1. Form the submatrices B, C, C^{T}, and D of A.
2. Recursively compute the inverse B^{-1} of B.
3. Compute the matrix product $W = CB^{-1}$, and then compute its transpose W^{T}, which equals $B^{-1}C^{\mathrm{T}}$ (by Exercise D.1-2 and $(B^{-1})^{\mathrm{T}} = B^{-1}$).
4. Compute the matrix product $X = WC^{\mathrm{T}}$, which equals $CB^{-1}C^{\mathrm{T}}$, and then compute the matrix $S = D - X = D - CB^{-1}C^{\mathrm{T}}$.
5. Recursively compute the inverse S^{-1} of S, and set V to S^{-1}.
6. Compute the matrix product $Y = S^{-1}W$, which equals $S^{-1}CB^{-1}$, and then compute its transpose Y^{T}, which equals $B^{-1}C^{\mathrm{T}}S^{-1}$ (by Exercise D.1-2, $(B^{-1})^{\mathrm{T}} = B^{-1}$, and $(S^{-1})^{\mathrm{T}} = S^{-1}$). Set T to $-Y^{\mathrm{T}}$ and U to $-Y$.
7. Compute the matrix product $Z = W^{\mathrm{T}}Y$, which equals $B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1}$, and set R to $B^{-1} + Z$.

Thus, we can invert an $n \times n$ symmetric positive-definite matrix by inverting two $n/2 \times n/2$ matrices in steps 2 and 5; performing four multiplications of $n/2 \times n/2$ matrices in steps 3, 4, 6, and 7; plus an additional cost of $O(n^2)$ for extracting submatrices from A, inserting submatrices into A^{-1}, and performing a constant number of additions, subtractions, and transposes on $n/2 \times n/2$ matrices. We get the recurrence

$$\begin{aligned} I(n) &\le 2I(n/2) + 4M(n/2) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)) \,. \end{aligned}$$

The second line holds because the second regularity condition in the statement of the theorem implies that $4M(n/2) < 2M(n)$ and because we assume that $M(n) = \Omega(n^2)$. The third line follows because the second regularity condition allows us to apply case 3 of the master theorem (Theorem 4.1).

It remains to prove that we can obtain the same asymptotic running time for matrix multiplication as for matrix inversion when A is invertible but not symmetric and positive-definite. The basic idea is that for any nonsingular matrix A, the matrix $A^{\mathrm{T}}A$ is symmetric (by Exercise D.1-2) and positive-definite (by Theorem D.6). The trick, then, is to reduce the problem of inverting A to the problem of inverting $A^{\mathrm{T}}A$.

The reduction is based on the observation that when A is an $n \times n$ nonsingular matrix, we have

$$A^{-1} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}} ,$$

since $((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})A = (A^{\mathrm{T}}A)^{-1}(A^{\mathrm{T}}A) = I_n$ and a matrix inverse is unique. Therefore, we can compute A^{-1} by first multiplying A^{T} by A to obtain $A^{\mathrm{T}}A$, then inverting the symmetric positive-definite matrix $A^{\mathrm{T}}A$ using the above divide-and-conquer algorithm, and finally multiplying the result by A^{T}. Each of these three steps takes $O(M(n))$ time, and thus we can invert any nonsingular matrix with real entries in $O(M(n))$ time. ■

The proof of Theorem 28.2 suggests a means of solving the equation $Ax = b$ by using LU decomposition without pivoting, so long as A is nonsingular. We multiply both sides of the equation by A^{T}, yielding $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b$. This transformation doesn't affect the solution x, since A^{T} is invertible, and so we can factor the symmetric positive-definite matrix $A^{\mathrm{T}}A$ by computing an LU decomposition. We then use forward and back substitution to solve for x with the right-hand side $A^{\mathrm{T}}b$. Although this method is theoretically correct, in practice the procedure LUP-DECOMPOSITION works much better. LUP decomposition requires fewer arithmetic operations by a constant factor, and it has somewhat better numerical properties.

Exercises

28.2-1

Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $S(n)$ denote the time required to square an $n \times n$ matrix. Show that multiplying and squaring matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time squaring algorithm, and an $S(n)$-time squaring algorithm implies an $O(S(n))$-time matrix-multiplication algorithm.

28.2-2
Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $L(n)$ be the time to compute the LUP decomposition of an $n \times n$ matrix. Show that multiplying matrices and computing LUP decompositions of matrices have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time LUP-decomposition algorithm, and an $L(n)$-time LUP-decomposition algorithm implies an $O(L(n))$-time matrix-multiplication algorithm.

28.2-3
Let $M(n)$ be the time to multiply two $n \times n$ matrices, and let $D(n)$ denote the time required to find the determinant of an $n \times n$ matrix. Show that multiplying matrices and computing the determinant have essentially the same difficulty: an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time determinant algorithm, and a $D(n)$-time determinant algorithm implies an $O(D(n))$-time matrix-multiplication algorithm.

28.2-4
Let $M(n)$ be the time to multiply two $n \times n$ boolean matrices, and let $T(n)$ be the time to find the transitive closure of an $n \times n$ boolean matrix. (See Section 25.2.) Show that an $M(n)$-time boolean matrix-multiplication algorithm implies an $O(M(n) \lg n)$-time transitive-closure algorithm, and a $T(n)$-time transitive-closure algorithm implies an $O(T(n))$-time boolean matrix-multiplication algorithm.

28.2-5
Does the matrix-inversion algorithm based on Theorem 28.2 work when matrix elements are drawn from the field of integers modulo 2? Explain.

28.2-6 ⋆
Generalize the matrix-inversion algorithm of Theorem 28.2 to handle matrices of complex numbers, and prove that your generalization works correctly. (*Hint:* Instead of the transpose of A, use the ***conjugate transpose*** A^*, which you obtain from the transpose of A by replacing every entry with its complex conjugate. Instead of symmetric matrices, consider ***Hermitian*** matrices, which are matrices A such that $A = A^*$.)

28.3 Symmetric positive-definite matrices and least-squares approximation

Symmetric positive-definite matrices have many interesting and desirable properties. For example, they are nonsingular, and we can perform LU decomposition on them without having to worry about dividing by 0. In this section, we shall

prove several other important properties of symmetric positive-definite matrices and show an interesting application to curve fitting by a least-squares approximation.

The first property we prove is perhaps the most basic.

Lemma 28.3
Any positive-definite matrix is nonsingular.

Proof Suppose that a matrix A is singular. Then by Corollary D.3, there exists a nonzero vector x such that $Ax = 0$. Hence, $x^{\mathrm{T}}Ax = 0$, and A cannot be positive-definite. ■

The proof that we can perform LU decomposition on a symmetric positive-definite matrix A without dividing by 0 is more involved. We begin by proving properties about certain submatrices of A. Define the kth ***leading submatrix*** of A to be the matrix A_k consisting of the intersection of the first k rows and first k columns of A.

Lemma 28.4
If A is a symmetric positive-definite matrix, then every leading submatrix of A is symmetric and positive-definite.

Proof That each leading submatrix A_k is symmetric is obvious. To prove that A_k is positive-definite, we assume that it is not and derive a contradiction. If A_k is not positive-definite, then there exists a k-vector $x_k \neq 0$ such that $x_k^{\mathrm{T}} A_k x_k \leq 0$. Let A be $n \times n$, and

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \tag{28.14}$$

for submatrices B (which is $(n-k)\times k$) and C (which is $(n-k)\times(n-k)$). Define the n-vector $x = (\, x_k^{\mathrm{T}} \quad 0\,)^{\mathrm{T}}$, where $n-k$ 0s follow x_k. Then we have

$$\begin{aligned} x^{\mathrm{T}}Ax &= (\, x_k^{\mathrm{T}} \quad 0\,)\begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix}\begin{pmatrix} x_k \\ 0 \end{pmatrix} \\ &= (\, x_k^{\mathrm{T}} \quad 0\,)\begin{pmatrix} A_k x_k \\ B x_k \end{pmatrix} \\ &= x_k^{\mathrm{T}} A_k x_k \\ &\leq 0\,, \end{aligned}$$

which contradicts A being positive-definite. ■

We now turn to some essential properties of the Schur complement. Let A be a symmetric positive-definite matrix, and let A_k be a leading $k \times k$ submatrix of A. Partition A once again according to equation (28.14). We generalize equation (28.9) to define the ***Schur complement*** S of A with respect to A_k as

$$S = C - BA_k^{-1}B^{\mathrm{T}} . \tag{28.15}$$

(By Lemma 28.4, A_k is symmetric and positive-definite; therefore, A_k^{-1} exists by Lemma 28.3, and S is well defined.) Note that our earlier definition (28.9) of the Schur complement is consistent with equation (28.15), by letting $k = 1$.

The next lemma shows that the Schur-complement matrices of symmetric positive-definite matrices are themselves symmetric and positive-definite. We used this result in Theorem 28.2, and we need its corollary to prove the correctness of LU decomposition for symmetric positive-definite matrices.

Lemma 28.5 (Schur complement lemma)
If A is a symmetric positive-definite matrix and A_k is a leading $k \times k$ submatrix of A, then the Schur complement S of A with respect to A_k is symmetric and positive-definite.

Proof Because A is symmetric, so is the submatrix C. By Exercise D.2-6, the product $BA_k^{-1}B^{\mathrm{T}}$ is symmetric, and by Exercise D.1-1, S is symmetric.

It remains to show that S is positive-definite. Consider the partition of A given in equation (28.14). For any nonzero vector x, we have $x^{\mathrm{T}}Ax > 0$ by the assumption that A is positive-definite. Let us break x into two subvectors y and z compatible with A_k and C, respectively. Because A_k^{-1} exists, we have

$$\begin{aligned}
x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\
&= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}} z \\ By + Cz \end{pmatrix} \\
&= y^{\mathrm{T}} A_k y + y^{\mathrm{T}} B^{\mathrm{T}} z + z^{\mathrm{T}} By + z^{\mathrm{T}} Cz \\
&= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}} A_k (y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z , \qquad (28.16)
\end{aligned}$$

by matrix magic. (Verify by multiplying through.) This last equation amounts to "completing the square" of the quadratic form. (See Exercise 28.3-2.)

Since $x^{\mathrm{T}}Ax > 0$ holds for any nonzero x, let us pick any nonzero z and then choose $y = -A_k^{-1}B^{\mathrm{T}}z$, which causes the first term in equation (28.16) to vanish, leaving

$$z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z = z^{\mathrm{T}}Sz$$

as the value of the expression. For any $z \neq 0$, we therefore have $z^{\mathrm{T}}Sz = x^{\mathrm{T}}Ax > 0$, and thus S is positive-definite. ∎

Corollary 28.6
LU decomposition of a symmetric positive-definite matrix never causes a division by 0.

Proof Let A be a symmetric positive-definite matrix. We shall prove something stronger than the statement of the corollary: every pivot is strictly positive. The first pivot is a_{11}. Let e_1 be the first unit vector, from which we obtain $a_{11} = e_1^{\mathrm{T}} A e_1 > 0$. Since the first step of LU decomposition produces the Schur complement of A with respect to $A_1 = (a_{11})$, Lemma 28.5 implies by induction that all pivots are positive. ■

Least-squares approximation

One important application of symmetric positive-definite matrices arises in fitting curves to given sets of data points. Suppose that we are given a set of m data points

$$(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m) ,$$

where we know that the y_i are subject to measurement errors. We would like to determine a function $F(x)$ such that the approximation errors

$$\eta_i = F(x_i) - y_i \tag{28.17}$$

are small for $i = 1, 2, \ldots, m$. The form of the function F depends on the problem at hand. Here, we assume that it has the form of a linearly weighted sum,

$$F(x) = \sum_{j=1}^{n} c_j f_j(x) ,$$

where the number of summands n and the specific ***basis functions*** f_j are chosen based on knowledge of the problem at hand. A common choice is $f_j(x) = x^{j-1}$, which means that

$$F(x) = c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1}$$

is a polynomial of degree $n - 1$ in x. Thus, given m data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, we wish to calculate n coefficients $c_1, c_2, \ldots, c_n$ that minimize the approximation errors $\eta_1, \eta_2, \ldots, \eta_m$.

By choosing $n = m$, we can calculate each y_i *exactly* in equation (28.17). Such a high-degree F "fits the noise" as well as the data, however, and generally gives poor results when used to predict y for previously unseen values of x. It is usually better to choose n significantly smaller than m and hope that by choosing the coefficients c_j well, we can obtain a function F that finds the significant patterns in the data points without paying undue attention to the noise. Some theoretical

principles exist for choosing n, but they are beyond the scope of this text. In any case, once we choose a value of n that is less than m, we end up with an overdetermined set of equations whose solution we wish to approximate. We now show how to do so.

Let

$$A = \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix}$$

denote the matrix of values of the basis functions at the given points; that is, $a_{ij} = f_j(x_i)$. Let $c = (c_k)$ denote the desired n-vector of coefficients. Then,

$$\begin{aligned} Ac &= \begin{pmatrix} f_1(x_1) & f_2(x_1) & \dots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \dots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \dots & f_n(x_m) \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \\ &= \begin{pmatrix} F(x_1) \\ F(x_2) \\ \vdots \\ F(x_m) \end{pmatrix} \end{aligned}$$

is the m-vector of "predicted values" for y. Thus,

$$\eta = Ac - y$$

is the m-vector of ***approximation errors***.

To minimize approximation errors, we choose to minimize the norm of the error vector η, which gives us a ***least-squares solution***, since

$$\|\eta\| = \left(\sum_{i=1}^{m} \eta_i^2 \right)^{1/2} .$$

Because

$$\|\eta\|^2 = \|Ac - y\|^2 = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right)^2 ,$$

we can minimize $\|\eta\|$ by differentiating $\|\eta\|^2$ with respect to each c_k and then setting the result to 0:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2\left(\sum_{j=1}^{n} a_{ij}c_j - y_i\right) a_{ik} = 0\,. \tag{28.18}$$

The n equations (28.18) for $k = 1, 2, \ldots, n$ are equivalent to the single matrix equation

$$(Ac - y)^{\mathrm{T}} A = 0$$

or, equivalently (using Exercise D.1-2), to

$$A^{\mathrm{T}}(Ac - y) = 0\,,$$

which implies

$$A^{\mathrm{T}}Ac = A^{\mathrm{T}}y\,. \tag{28.19}$$

In statistics, this is called the ***normal equation***. The matrix $A^{\mathrm{T}}A$ is symmetric by Exercise D.1-2, and if A has full column rank, then by Theorem D.6, $A^{\mathrm{T}}A$ is positive-definite as well. Hence, $(A^{\mathrm{T}}A)^{-1}$ exists, and the solution to equation (28.19) is

$$\begin{aligned} c &= \left((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\right) y \\ &= A^{+}y\,, \end{aligned} \tag{28.20}$$

where the matrix $A^{+} = ((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}})$ is the ***pseudoinverse*** of the matrix A. The pseudoinverse naturally generalizes the notion of a matrix inverse to the case in which A is not square. (Compare equation (28.20) as the approximate solution to $Ac = y$ with the solution $A^{-1}b$ as the exact solution to $Ax = b$.)

As an example of producing a least-squares fit, suppose that we have five data points

$$\begin{aligned} (x_1, y_1) &= (-1, 2)\,, \\ (x_2, y_2) &= (1, 1)\,, \\ (x_3, y_3) &= (2, 1)\,, \\ (x_4, y_4) &= (3, 0)\,, \\ (x_5, y_5) &= (5, 3)\,, \end{aligned}$$

shown as black dots in Figure 28.3. We wish to fit these points with a quadratic polynomial

$$F(x) = c_1 + c_2 x + c_3 x^2\,.$$

We start with the matrix of basis-function values

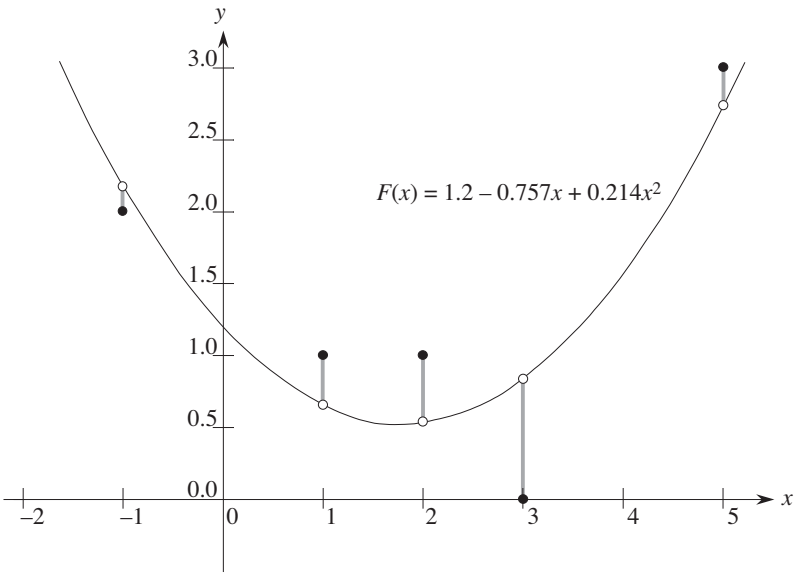


Figure 28.3 The least-squares fit of a quadratic polynomial to the set of five data points $\{(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)\}$. The black dots are the data points, and the white dots are their estimated values predicted by the polynomial $F(x) = 1.2 - 0.757x + 0.214x^2$, the quadratic polynomial that minimizes the sum of the squared errors. Each shaded line shows the error for one data point.

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \\ 1 & x_5 & x_5^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \\ 1 & 5 & 25 \end{pmatrix},$$

whose pseudoinverse is

$$A^+ = \begin{pmatrix} 0.500 & 0.300 & 0.200 & 0.100 & -0.100 \\ -0.388 & 0.093 & 0.190 & 0.193 & -0.088 \\ 0.060 & -0.036 & -0.048 & -0.036 & 0.060 \end{pmatrix}.$$

Multiplying y by A^+, we obtain the coefficient vector

$$c = \begin{pmatrix} 1.200 \\ -0.757 \\ 0.214 \end{pmatrix},$$

which corresponds to the quadratic polynomial

$$F(x) = 1.200 - 0.757x + 0.214x^2$$

as the closest-fitting quadratic to the given data, in a least-squares sense.

As a practical matter, we solve the normal equation (28.19) by multiplying y by A^{T} and then finding an LU decomposition of $A^{\mathrm{T}}A$. If A has full rank, the matrix $A^{\mathrm{T}}A$ is guaranteed to be nonsingular, because it is symmetric and positive-definite. (See Exercise D.1-2 and Theorem D.6.)

Exercises

28.3-1
Prove that every diagonal element of a symmetric positive-definite matrix is positive.

28.3-2
Let $A = \begin{pmatrix} a & b \\ b & c \end{pmatrix}$ be a 2×2 symmetric positive-definite matrix. Prove that its determinant $ac - b^2$ is positive by "completing the square" in a manner similar to that used in the proof of Lemma 28.5.

28.3-3
Prove that the maximum element in a symmetric positive-definite matrix lies on the diagonal.

28.3-4
Prove that the determinant of each leading submatrix of a symmetric positive-definite matrix is positive.

28.3-5
Let A_k denote the kth leading submatrix of a symmetric positive-definite matrix A. Prove that $\det(A_k)/\det(A_{k-1})$ is the kth pivot during LU decomposition, where, by convention, $\det(A_0) = 1$.

28.3-6
Find the function of the form

$$F(x) = c_1 + c_2 x \lg x + c_3 e^x$$

that is the best least-squares fit to the data points

$$(1, 1), (2, 1), (3, 3), (4, 8) .$$

28.3-7
Show that the pseudoinverse A^+ satisfies the following four equations:

$$\begin{aligned} AA^+A &= A\,, \\ A^+AA^+ &= A^+\,, \\ (AA^+)^{\mathrm{T}} &= AA^+\,, \\ (A^+A)^{\mathrm{T}} &= A^+A\,. \end{aligned}$$

Problems

28-1 Tridiagonal systems of linear equations
Consider the tridiagonal matrix

$$A = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{pmatrix}.$$

a. Find an LU decomposition of A.

b. Solve the equation $Ax = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \end{pmatrix}^{\mathrm{T}}$ by using forward and back substitution.

c. Find the inverse of A.

d. Show how, for any $n \times n$ symmetric positive-definite, tridiagonal matrix A and any n-vector b, to solve the equation $Ax = b$ in $O(n)$ time by performing an LU decomposition. Argue that any method based on forming A^{-1} is asymptotically more expensive in the worst case.

e. Show how, for any $n \times n$ nonsingular, tridiagonal matrix A and any n-vector b, to solve the equation $Ax = b$ in $O(n)$ time by performing an LUP decomposition.

28-2 Splines
A practical method for interpolating a set of points with a curve is to use ***cubic splines***. We are given a set $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ of $n + 1$ point-value pairs, where $x_0 < x_1 < \cdots < x_n$. We wish to fit a piecewise-cubic curve (spline) $f(x)$ to the points. That is, the curve $f(x)$ is made up of n cubic polynomials $f_i(x) = a_i + b_i x + c_i x^2 + d_i x^3$ for $i = 0, 1, \ldots, n - 1$, where if x falls in

the range $x_i \le x \le x_{i+1}$, then the value of the curve is given by $f(x) = f_i(x - x_i)$. The points x_i at which the cubic polynomials are "pasted" together are called ***knots***. For simplicity, we shall assume that $x_i = i$ for $i = 0, 1, \ldots, n$.

To ensure continuity of $f(x)$, we require that

$$\begin{aligned} f(x_i) &= f_i(0) = y_i \,, \\ f(x_{i+1}) &= f_i(1) = y_{i+1} \end{aligned}$$

for $i = 0, 1, \ldots, n-1$. To ensure that $f(x)$ is sufficiently smooth, we also insist that the first derivative be continuous at each knot:

$$f'(x_{i+1}) = f_i'(1) = f_{i+1}'(0)$$

for $i = 0, 1, \ldots, n-2$.

a. Suppose that for $i = 0, 1, \ldots, n$, we are given not only the point-value pairs $\{(x_i, y_i)\}$ but also the first derivatives $D_i = f'(x_i)$ at each knot. Express each coefficient a_i, b_i, c_i, and d_i in terms of the values y_i, y_{i+1}, D_i, and D_{i+1}. (Remember that $x_i = i$.) How quickly can we compute the $4n$ coefficients from the point-value pairs and first derivatives?

The question remains of how to choose the first derivatives of $f(x)$ at the knots. One method is to require the second derivatives to be continuous at the knots:

$$f''(x_{i+1}) = f_i''(1) = f_{i+1}''(0)$$

for $i = 0, 1, \ldots, n-2$. At the first and last knots, we assume that $f''(x_0) = f_0''(0) = 0$ and $f''(x_n) = f_{n-1}''(1) = 0$; these assumptions make $f(x)$ a ***natural*** cubic spline.

b. Use the continuity constraints on the second derivative to show that for $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \,. \tag{28.21}$$

c. Show that

$$2D_0 + D_1 = 3(y_1 - y_0) \,, \tag{28.22}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) \,. \tag{28.23}$$

d. Rewrite equations (28.21)–(28.23) as a matrix equation involving the vector $D = \langle D_0, D_1, \ldots, D_n \rangle$ of unknowns. What attributes does the matrix in your equation have?

e. Argue that a natural cubic spline can interpolate a set of $n + 1$ point-value pairs in $O(n)$ time (see Problem 28-1).

f. Show how to determine a natural cubic spline that interpolates a set of $n + 1$ points (x_i, y_i) satisfying $x_0 < x_1 < \cdots < x_n$, even when x_i is not necessarily equal to i. What matrix equation must your method solve, and how quickly does your algorithm run?

Chapter notes

Many excellent texts describe numerical and scientific computation in much greater detail than we have room for here. The following are especially readable: George and Liu [132], Golub and Van Loan [144], Press, Teukolsky, Vetterling, and Flannery [283, 284], and Strang [323, 324].

Golub and Van Loan [144] discuss numerical stability. They show why $\det(A)$ is not necessarily a good indicator of the stability of a matrix A, proposing instead to use $\|A\|_\infty \|A^{-1}\|_\infty$, where $\|A\|_\infty = \max_{1 \le i \le n} \sum_{j=1}^{n} |a_{ij}|$. They also address the question of how to compute this value without actually computing A^{-1}.

Gaussian elimination, upon which the LU and LUP decompositions are based, was the first systematic method for solving linear systems of equations. It was also one of the earliest numerical algorithms. Although it was known earlier, its discovery is commonly attributed to C. F. Gauss (1777–1855). In his famous paper [325], Strassen showed that an $n \times n$ matrix can be inverted in $O(n^{\lg 7})$ time. Winograd [358] originally proved that matrix multiplication is no harder than matrix inversion, and the converse is due to Aho, Hopcroft, and Ullman [5].

Another important matrix decomposition is the ***singular value decomposition***, or ***SVD***. The SVD factors an $m \times n$ matrix A into $A = Q_1 \Sigma Q_2^{\mathrm{T}}$, where Σ is an $m \times n$ matrix with nonzero values only on the diagonal, Q_1 is $m \times m$ with mutually orthonormal columns, and Q_2 is $n \times n$, also with mutually orthonormal columns. Two vectors are ***orthonormal*** if their inner product is 0 and each vector has a norm of 1. The books by Strang [323, 324] and Golub and Van Loan [144] contain good treatments of the SVD.

Strang [324] has an excellent presentation of symmetric positive-definite matrices and of linear algebra in general.

29 Linear Programming

Many problems take the form of maximizing or minimizing an objective, given limited resources and competing constraints. If we can specify the objective as a linear function of certain variables, and if we can specify the constraints on resources as equalities or inequalities on those variables, then we have a ***linear-programming problem***. Linear programs arise in a variety of practical applications. We begin by studying an application in electoral politics.

A political problem

Suppose that you are a politician trying to win an election. Your district has three different types of areas—urban, suburban, and rural. These areas have, respectively, 100,000, 200,000, and 50,000 registered voters. Although not all the registered voters actually go to the polls, you decide that to govern effectively, you would like at least half the registered voters in each of the three regions to vote for you. You are honorable and would never consider supporting policies in which you do not believe. You realize, however, that certain issues may be more effective in winning votes in certain places. Your primary issues are building more roads, gun control, farm subsidies, and a gasoline tax dedicated to improved public transit. According to your campaign staff's research, you can estimate how many votes you win or lose from each population segment by spending $1,000 on advertising on each issue. This information appears in the table of Figure 29.1. In this table, each entry indicates the number of thousands of either urban, suburban, or rural voters who would be won over by spending $1,000 on advertising in support of a particular issue. Negative entries denote votes that would be lost. Your task is to figure out the minimum amount of money that you need to spend in order to win 50,000 urban votes, 100,000 suburban votes, and 25,000 rural votes.

You could, by trial and error, devise a strategy that wins the required number of votes, but the strategy you come up with might not be the least expensive one. For example, you could devote $20,000 of advertising to building roads, $0 to gun control, $4,000 to farm subsidies, and $9,000 to a gasoline tax. In this case, you

policy	urban	suburban	rural
build roads	−2	5	3
gun control	8	2	−5
farm subsidies	0	0	10
gasoline tax	10	0	−2

Figure 29.1 The effects of policies on voters. Each entry describes the number of thousands of urban, suburban, or rural voters who could be won over by spending $1,000 on advertising support of a policy on a particular issue. Negative entries denote votes that would be lost.

would win $20(-2)+0(8)+4(0)+9(10) = 50$ thousand urban votes, $20(5)+0(2)+4(0)+9(0) = 100$ thousand suburban votes, and $20(3)+0(-5)+4(10)+9(-2) = 82$ thousand rural votes. You would win the exact number of votes desired in the urban and suburban areas and more than enough votes in the rural area. (In fact, in the rural area, you would receive more votes than there are voters.) In order to garner these votes, you would have paid for $20 + 0 + 4 + 9 = 33$ thousand dollars of advertising.

Naturally, you may wonder whether this strategy is the best possible. That is, could you achieve your goals while spending less on advertising? Additional trial and error might help you to answer this question, but wouldn't you rather have a systematic method for answering such questions? In order to develop one, we shall formulate this question mathematically. We introduce 4 variables:

- x_1 is the number of thousands of dollars spent on advertising on building roads,
- x_2 is the number of thousands of dollars spent on advertising on gun control,
- x_3 is the number of thousands of dollars spent on advertising on farm subsidies, and
- x_4 is the number of thousands of dollars spent on advertising on a gasoline tax.

We can write the requirement that we win at least 50,000 urban votes as

$$-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50 \, . \tag{29.1}$$

Similarly, we can write the requirements that we win at least 100,000 suburban votes and 25,000 rural votes as

$$5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100 \tag{29.2}$$

and

$$3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 \, . \tag{29.3}$$

Any setting of the variables x_1, x_2, x_3, x_4 that satisfies inequalities (29.1)–(29.3) yields a strategy that wins a sufficient number of each type of vote. In order to

keep costs as small as possible, you would like to minimize the amount spent on advertising. That is, you want to minimize the expression

$$x_1 + x_2 + x_3 + x_4 \,. \tag{29.4}$$

Although negative advertising often occurs in political campaigns, there is no such thing as negative-cost advertising. Consequently, we require that

$$x_1 \ge 0,\ x_2 \ge 0,\ x_3 \ge 0,\ \text{and}\ \ x_4 \ge 0 \,. \tag{29.5}$$

Combining inequalities (29.1)–(29.3) and (29.5) with the objective of minimizing (29.4), we obtain what is known as a "linear program." We format this problem as

$$\begin{aligned}
&\text{minimize} && x_1 + x_2 + x_3 + x_4 && && && (29.6)\\
&\text{subject to} \\
& && -2x_1 + 8x_2 + 0x_3 + 10x_4 &\ge& \ 50 && (29.7)\\
& && 5x_1 + 2x_2 + 0x_3 + 0x_4 &\ge& \ 100 && (29.8)\\
& && 3x_1 - 5x_2 + 10x_3 - 2x_4 &\ge& \ 25 && (29.9)\\
& && x_1, x_2, x_3, x_4 &\ge& \ 0 \,. && (29.10)
\end{aligned}$$

The solution of this linear program yields your optimal strategy.

General linear programs

In the general linear-programming problem, we wish to optimize a linear function subject to a set of linear inequalities. Given a set of real numbers $a_1, a_2, \ldots, a_n$ and a set of variables $x_1, x_2, \ldots, x_n$, we define a ***linear function*** f on those variables by

$$f(x_1, x_2, \ldots, x_n) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \sum_{j=1}^{n} a_j x_j \,.$$

If b is a real number and f is a linear function, then the equation

$$f(x_1, x_2, \ldots, x_n) = b$$

is a ***linear equality*** and the inequalities

$$f(x_1, x_2, \ldots, x_n) \le b$$

and

$$f(x_1, x_2, \ldots, x_n) \ge b$$

are ***linear inequalities***. We use the general term ***linear constraints*** to denote either linear equalities or linear inequalities. In linear programming, we do not allow strict inequalities. Formally, a ***linear-programming problem*** is the problem of either minimizing or maximizing a linear function subject to a finite set of linear constraints. If we are to minimize, then we call the linear program a ***minimization linear program***, and if we are to maximize, then we call the linear program a ***maximization linear program***.

The remainder of this chapter covers how to formulate and solve linear programs. Although several polynomial-time algorithms for linear programming have been developed, we will not study them in this chapter. Instead, we shall study the simplex algorithm, which is the oldest linear-programming algorithm. The simplex algorithm does not run in polynomial time in the worst case, but it is fairly efficient and widely used in practice.

An overview of linear programming

In order to describe properties of and algorithms for linear programs, we find it convenient to express them in canonical forms. We shall use two forms, ***standard*** and ***slack***, in this chapter. We will define them precisely in Section 29.1. Informally, a linear program in standard form is the maximization of a linear function subject to linear *inequalities*, whereas a linear program in slack form is the maximization of a linear function subject to linear *equalities*. We shall typically use standard form for expressing linear programs, but we find it more convenient to use slack form when we describe the details of the simplex algorithm. For now, we restrict our attention to maximizing a linear function on n variables subject to a set of m linear inequalities.

Let us first consider the following linear program with two variables:

$$\begin{aligned}
&\text{maximize} && x_1 + x_2 && && && && (29.11)\\
&\text{subject to} && && && && && \\
& && 4x_1 - x_2 && \le && 8 && (29.12)\\
& && 2x_1 + x_2 && \le && 10 && (29.13)\\
& && 5x_1 - 2x_2 && \ge && -2 && (29.14)\\
& && x_1, x_2 && \ge && 0\ . && (29.15)
\end{aligned}$$

We call any setting of the variables x_1 and x_2 that satisfies all the constraints (29.12)–(29.15) a ***feasible solution*** to the linear program. If we graph the constraints in the (x_1, x_2)-Cartesian coordinate system, as in Figure 29.2(a), we see

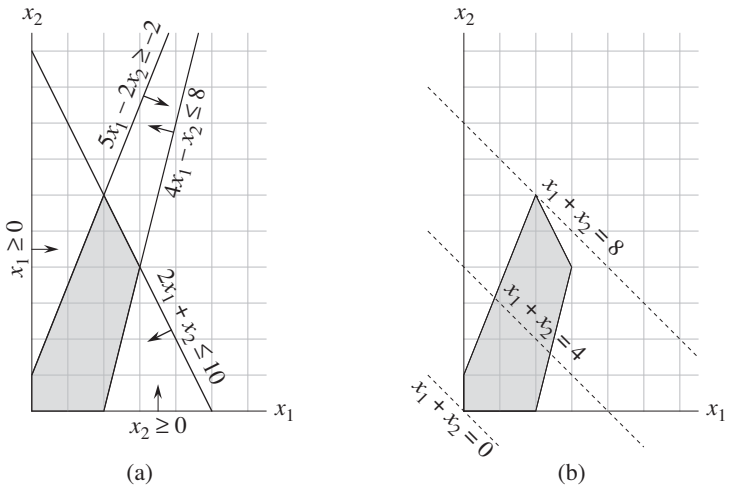


Figure 29.2 **(a)** The linear program given in (29.12)–(29.15). Each constraint is represented by a line and a direction. The intersection of the constraints, which is the feasible region, is shaded. **(b)** The dotted lines show, respectively, the points for which the objective value is 0, 4, and 8. The optimal solution to the linear program is $x_1 = 2$ and $x_2 = 6$ with objective value 8.

that the set of feasible solutions (shaded in the figure) forms a convex region[1] in the two-dimensional space. We call this convex region the ***feasible region*** and the function we wish to maximize the ***objective function***. Conceptually, we could evaluate the objective function $x_1 + x_2$ at each point in the feasible region; we call the value of the objective function at a particular point the ***objective value***. We could then identify a point that has the maximum objective value as an optimal solution. For this example (and for most linear programs), the feasible region contains an infinite number of points, and so we need to determine an efficient way to find a point that achieves the maximum objective value without explicitly evaluating the objective function at every point in the feasible region.

In two dimensions, we can optimize via a graphical procedure. The set of points for which $x_1 + x_2 = z$, for any z, is a line with a slope of -1. If we plot $x_1 + x_2 = 0$, we obtain the line with slope -1 through the origin, as in Figure 29.2(b). The intersection of this line and the feasible region is the set of feasible solutions that have an objective value of 0. In this case, that intersection of the line with the feasible region is the single point $(0, 0)$. More generally, for any z, the intersection

[1] An intuitive definition of a convex region is that it fulfills the requirement that for any two points in the region, all points on a line segment between them are also in the region.

of the line $x_1 + x_2 = z$ and the feasible region is the set of feasible solutions that have objective value z. Figure 29.2(b) shows the lines $x_1 + x_2 = 0$, $x_1 + x_2 = 4$, and $x_1 + x_2 = 8$. Because the feasible region in Figure 29.2 is bounded, there must be some maximum value z for which the intersection of the line $x_1 + x_2 = z$ and the feasible region is nonempty. Any point at which this occurs is an optimal solution to the linear program, which in this case is the point $x_1 = 2$ and $x_2 = 6$ with objective value 8.

It is no accident that an optimal solution to the linear program occurs at a vertex of the feasible region. The maximum value of z for which the line $x_1 + x_2 = z$ intersects the feasible region must be on the boundary of the feasible region, and thus the intersection of this line with the boundary of the feasible region is either a single vertex or a line segment. If the intersection is a single vertex, then there is just one optimal solution, and it is that vertex. If the intersection is a line segment, every point on that line segment must have the same objective value; in particular, both endpoints of the line segment are optimal solutions. Since each endpoint of a line segment is a vertex, there is an optimal solution at a vertex in this case as well.

Although we cannot easily graph linear programs with more than two variables, the same intuition holds. If we have three variables, then each constraint corresponds to a half-space in three-dimensional space. The intersection of these half-spaces forms the feasible region. The set of points for which the objective function obtains a given value z is now a plane (assuming no degenerate conditions). If all coefficients of the objective function are nonnegative, and if the origin is a feasible solution to the linear program, then as we move this plane away from the origin, in a direction normal to the objective function, we find points of increasing objective value. (If the origin is not feasible or if some coefficients in the objective function are negative, the intuitive picture becomes slightly more complicated.) As in two dimensions, because the feasible region is convex, the set of points that achieve the optimal objective value must include a vertex of the feasible region. Similarly, if we have n variables, each constraint defines a half-space in n-dimensional space. We call the feasible region formed by the intersection of these half-spaces a ***simplex***. The objective function is now a hyperplane and, because of convexity, an optimal solution still occurs at a vertex of the simplex.

The ***simplex algorithm*** takes as input a linear program and returns an optimal solution. It starts at some vertex of the simplex and performs a sequence of iterations. In each iteration, it moves along an edge of the simplex from a current vertex to a neighboring vertex whose objective value is no smaller than that of the current vertex (and usually is larger.) The simplex algorithm terminates when it reaches a local maximum, which is a vertex from which all neighboring vertices have a smaller objective value. Because the feasible region is convex and the objective function is linear, this local optimum is actually a global optimum. In Section 29.4,

we shall use a concept called "duality" to show that the solution returned by the simplex algorithm is indeed optimal.

Although the geometric view gives a good intuitive view of the operations of the simplex algorithm, we shall not refer to it explicitly when developing the details of the simplex algorithm in Section 29.3. Instead, we take an algebraic view. We first write the given linear program in slack form, which is a set of linear equalities. These linear equalities express some of the variables, called "basic variables," in terms of other variables, called "nonbasic variables." We move from one vertex to another by making a basic variable become nonbasic and making a nonbasic variable become basic. We call this operation a "pivot" and, viewed algebraically, it is nothing more than rewriting the linear program in an equivalent slack form.

The two-variable example described above was particularly simple. We shall need to address several more details in this chapter. These issues include identifying linear programs that have no solutions, linear programs that have no finite optimal solution, and linear programs for which the origin is not a feasible solution.

Applications of linear programming

Linear programming has a large number of applications. Any textbook on operations research is filled with examples of linear programming, and linear programming has become a standard tool taught to students in most business schools. The election scenario is one typical example. Two more examples of linear programming are the following:

- An airline wishes to schedule its flight crews. The Federal Aviation Administration imposes many constraints, such as limiting the number of consecutive hours that each crew member can work and insisting that a particular crew work only on one model of aircraft during each month. The airline wants to schedule crews on all of its flights using as few crew members as possible.
- An oil company wants to decide where to drill for oil. Siting a drill at a particular location has an associated cost and, based on geological surveys, an expected payoff of some number of barrels of oil. The company has a limited budget for locating new drills and wants to maximize the amount of oil it expects to find, given this budget.

With linear programs, we also model and solve graph and combinatorial problems, such as those appearing in this textbook. We have already seen a special case of linear programming used to solve systems of difference constraints in Section 24.4. In Section 29.2, we shall study how to formulate several graph and network-flow problems as linear programs. In Section 35.4, we shall use linear programming as a tool to find an approximate solution to another graph problem.

Algorithms for linear programming

This chapter studies the simplex algorithm. This algorithm, when implemented carefully, often solves general linear programs quickly in practice. With some carefully contrived inputs, however, the simplex algorithm can require exponential time. The first polynomial-time algorithm for linear programming was the ***ellipsoid algorithm***, which runs slowly in practice. A second class of polynomial-time algorithms are known as ***interior-point methods***. In contrast to the simplex algorithm, which moves along the exterior of the feasible region and maintains a feasible solution that is a vertex of the simplex at each iteration, these algorithms move through the interior of the feasible region. The intermediate solutions, while feasible, are not necessarily vertices of the simplex, but the final solution is a vertex. For large inputs, interior-point algorithms can run as fast as, and sometimes faster than, the simplex algorithm. The chapter notes point you to more information about these algorithms.

If we add to a linear program the additional requirement that all variables take on integer values, we have an ***integer linear program***. Exercise 34.5-3 asks you to show that just finding a feasible solution to this problem is NP-hard; since no polynomial-time algorithms are known for any NP-hard problems, there is no known polynomial-time algorithm for integer linear programming. In contrast, we can solve a general linear-programming problem in polynomial time.

In this chapter, if we have a linear program with variables $x = (x_1, x_2, \ldots, x_n)$ and wish to refer to a particular setting of the variables, we shall use the notation $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$.

29.1 Standard and slack forms

This section describes two formats, standard form and slack form, that are useful when we specify and work with linear programs. In standard form, all the constraints are inequalities, whereas in slack form, all constraints are equalities (except for those that require the variables to be nonnegative).

Standard form

In ***standard form***, we are given n real numbers $c_1, c_2, \ldots, c_n$; m real numbers $b_1, b_2, \ldots, b_m$; and mn real numbers a_{ij} for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$. We wish to find n real numbers $x_1, x_2, \ldots, x_n$ that

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.16}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.17}$$

$$x_j \geq 0 \quad \text{for } j = 1, 2, \ldots, n\,. \tag{29.18}$$

Generalizing the terminology we introduced for the two-variable linear program, we call expression (29.16) the ***objective function*** and the $n + m$ inequalities in lines (29.17) and (29.18) the ***constraints***. The n constraints in line (29.18) are the ***nonnegativity constraints***. An arbitrary linear program need not have nonnegativity constraints, but standard form requires them. Sometimes we find it convenient to express a linear program in a more compact form. If we create an $m \times n$ matrix $A = (a_{ij})$, an m-vector $b = (b_i)$, an n-vector $c = (c_j)$, and an n-vector $x = (x_j)$, then we can rewrite the linear program defined in (29.16)–(29.18) as

$$\text{maximize} \quad c^{\mathrm{T}} x \tag{29.19}$$

subject to

$$Ax \leq b \tag{29.20}$$

$$x \geq 0\,. \tag{29.21}$$

In line (29.19), $c^{\mathrm{T}} x$ is the inner product of two vectors. In inequality (29.20), Ax is a matrix-vector product, and in inequality (29.21), $x \geq 0$ means that each entry of the vector x must be nonnegative. We see that we can specify a linear program in standard form by a tuple (A, b, c), and we shall adopt the convention that A, b, and c always have the dimensions given above.

We now introduce terminology to describe solutions to linear programs. We used some of this terminology in the earlier example of a two-variable linear program. We call a setting of the variables $\bar{x}$ that satisfies all the constraints a ***feasible solution***, whereas a setting of the variables $\bar{x}$ that fails to satisfy at least one constraint is an ***infeasible solution***. We say that a solution $\bar{x}$ has ***objective value*** $c^{\mathrm{T}} \bar{x}$. A feasible solution $\bar{x}$ whose objective value is maximum over all feasible solutions is an ***optimal solution***, and we call its objective value $c^{\mathrm{T}} \bar{x}$ the ***optimal objective value***. If a linear program has no feasible solutions, we say that the linear program is ***infeasible***; otherwise it is ***feasible***. If a linear program has some feasible solutions but does not have a finite optimal objective value, we say that the linear program is ***unbounded***. Exercise 29.1-9 asks you to show that a linear program can have a finite optimal objective value even if the feasible region is not bounded.

Converting linear programs into standard form

It is always possible to convert a linear program, given as minimizing or maximizing a linear function subject to linear constraints, into standard form. A linear program might not be in standard form for any of four possible reasons:

1. The objective function might be a minimization rather than a maximization.
2. There might be variables without nonnegativity constraints.
3. There might be ***equality constraints***, which have an equal sign rather than a less-than-or-equal-to sign.
4. There might be ***inequality constraints***, but instead of having a less-than-or-equal-to sign, they have a greater-than-or-equal-to sign.

When converting one linear program L into another linear program L', we would like the property that an optimal solution to L' yields an optimal solution to L. To capture this idea, we say that two maximization linear programs L and L' are ***equivalent*** if for each feasible solution $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value z, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value z. (This definition does not imply a one-to-one correspondence between feasible solutions.) A minimization linear program L and a maximization linear program L' are equivalent if for each feasible solution $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value $-z$, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value $-z$.

We now show how to remove, one by one, each of the possible problems in the list above. After removing each one, we shall argue that the new linear program is equivalent to the old one.

To convert a minimization linear program L into an equivalent maximization linear program L', we simply negate the coefficients in the objective function. Since L and L' have identical sets of feasible solutions and, for any feasible solution, the objective value in L is the negative of the objective value in L', these two linear programs are equivalent. For example, if we have the linear program

$$\begin{aligned}
\text{minimize} \quad & -2x_1 + 3x_2 \\
\text{subject to} \quad & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \le 4 \\
& x_1 \ge 0 \ ,
\end{aligned}$$

and we negate the coefficients of the objective function, we obtain

$$
\begin{aligned}
\text{maximize} \quad & 2x_1 - 3x_2 \\
\text{subject to} \quad & \\
& x_1 + x_2 = 7 \\
& x_1 - 2x_2 \le 4 \\
& x_1 \ge 0 \ .
\end{aligned}
$$

Next, we show how to convert a linear program in which some of the variables do not have nonnegativity constraints into one in which each variable has a nonnegativity constraint. Suppose that some variable x_j does not have a nonnegativity constraint. Then, we replace each occurrence of x_j by $x'_j - x''_j$, and add the nonnegativity constraints $x'_j \ge 0$ and $x''_j \ge 0$. Thus, if the objective function has a term $c_j x_j$, we replace it by $c_j x'_j - c_j x''_j$, and if constraint i has a term $a_{ij} x_j$, we replace it by $a_{ij} x'_j - a_{ij} x''_j$. Any feasible solution $\hat{x}$ to the new linear program corresponds to a feasible solution $\bar{x}$ to the original linear program with $\bar{x}_j = \hat{x}'_j - \hat{x}''_j$ and with the same objective value. Also, any feasible solution $\bar{x}$ to the original linear program corresponds to a feasible solution $\hat{x}$ to the new linear program with $\hat{x}'_j = \bar{x}_j$ and $\hat{x}''_j = 0$ if $\bar{x}_j \ge 0$, or with $\hat{x}''_j = \bar{x}_j$ and $\hat{x}'_j = 0$ if $\bar{x}_j < 0$. The two linear programs have the same objective value regardless of the sign of $\bar{x}_j$. Thus, the two linear programs are equivalent. We apply this conversion scheme to each variable that does not have a nonnegativity constraint to yield an equivalent linear program in which all variables have nonnegativity constraints.

Continuing the example, we want to ensure that each variable has a corresponding nonnegativity constraint. Variable x_1 has such a constraint, but variable x_2 does not. Therefore, we replace x_2 by two variables x'_2 and x''_2, and we modify the linear program to obtain

$$
\begin{aligned}
\text{maximize} \quad & 2x_1 - 3x'_2 + 3x''_2 \\
\text{subject to} \quad & \\
& x_1 + x'_2 - x''_2 = 7 \\
& x_1 - 2x'_2 + 2x''_2 \le 4 \\
& x_1, x'_2, x''_2 \ge 0 \ .
\end{aligned}
\tag{29.22}
$$

Next, we convert equality constraints into inequality constraints. Suppose that a linear program has an equality constraint $f(x_1, x_2, \ldots, x_n) = b$. Since $x = y$ if and only if both $x \ge y$ and $x \le y$, we can replace this equality constraint by the pair of inequality constraints $f(x_1, x_2, \ldots, x_n) \le b$ and $f(x_1, x_2, \ldots, x_n) \ge b$. Repeating this conversion for each equality constraint yields a linear program in which all constraints are inequalities.

Finally, we can convert the greater-than-or-equal-to constraints to less-than-or-equal-to constraints by multiplying these constraints through by -1. That is, any inequality of the form

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$$

is equivalent to

$$\sum_{j=1}^{n} -a_{ij} x_j \leq -b_i \ .$$

Thus, by replacing each coefficient a_{ij} by $-a_{ij}$ and each value b_i by $-b_i$, we obtain an equivalent less-than-or-equal-to constraint.

Finishing our example, we replace the equality in constraint (29.22) by two inequalities, obtaining

$$\begin{aligned}
&\text{maximize} && 2x_1 - 3x_2' + 3x_2'' \\
&\text{subject to} \\
&&& x_1 + x_2' - x_2'' \leq 7 \\
&&& x_1 + x_2' - x_2'' \geq 7 \\
&&& x_1 - 2x_2' + 2x_2'' \leq 4 \\
&&& x_1, x_2', x_2'' \geq 0 \ .
\end{aligned} \tag{29.23}$$

Finally, we negate constraint (29.23). For consistency in variable names, we rename x_2' to x_2 and x_2'' to x_3, obtaining the standard form

$$\text{maximize} \quad 2x_1 - 3x_2 + 3x_3 \tag{29.24}$$

$$\text{subject to}$$

$$x_1 + x_2 - x_3 \leq 7 \tag{29.25}$$

$$-x_1 - x_2 + x_3 \leq -7 \tag{29.26}$$

$$x_1 - 2x_2 + 2x_3 \leq 4 \tag{29.27}$$

$$x_1, x_2, x_3 \geq 0 \ . \tag{29.28}$$

Converting linear programs into slack form

To efficiently solve a linear program with the simplex algorithm, we prefer to express it in a form in which some of the constraints are equality constraints. More precisely, we shall convert it into a form in which the nonnegativity constraints are the only inequality constraints, and the remaining constraints are equalities. Let

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \tag{29.29}$$

be an inequality constraint. We introduce a new variable s and rewrite inequality (29.29) as the two constraints

$$s = b_i - \sum_{j=1}^{n} a_{ij} x_j \,, \tag{29.30}$$

$$s \geq 0 \,. \tag{29.31}$$

We call s a ***slack variable*** because it measures the ***slack***, or difference, between the left-hand and right-hand sides of equation (29.29). (We shall soon see why we find it convenient to write the constraint with only the slack variable on the left-hand side.) Because inequality (29.29) is true if and only if both equation (29.30) and inequality (29.31) are true, we can convert each inequality constraint of a linear program in this way to obtain an equivalent linear program in which the only inequality constraints are the nonnegativity constraints. When converting from standard to slack form, we shall use x_{n+i} (instead of s) to denote the slack variable associated with the ith inequality. The ith constraint is therefore

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j \,, \tag{29.32}$$

along with the nonnegativity constraint $x_{n+i} \geq 0$.

By converting each constraint of a linear program in standard form, we obtain a linear program in a different form. For example, for the linear program described in (29.24)–(29.28), we introduce slack variables x_4, x_5, and x_6, obtaining

$$\text{maximize} \qquad 2x_1 - 3x_2 + 3x_3 \tag{29.33}$$

subject to

$$x_4 = 7 - x_1 - x_2 + x_3 \tag{29.34}$$

$$x_5 = -7 + x_1 + x_2 - x_3 \tag{29.35}$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_3 \tag{29.36}$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \,. \tag{29.37}$$

In this linear program, all the constraints except for the nonnegativity constraints are equalities, and each variable is subject to a nonnegativity constraint. We write each equality constraint with one of the variables on the left-hand side of the equality and all others on the right-hand side. Furthermore, each equation has the same set of variables on the right-hand side, and these variables are also the only ones that appear in the objective function. We call the variables on the left-hand side of the equalities ***basic variables*** and those on the right-hand side ***nonbasic variables***.

For linear programs that satisfy these conditions, we shall sometimes omit the words "maximize" and "subject to," as well as the explicit nonnegativity constraints. We shall also use the variable z to denote the value of the objective func-

tion. We call the resulting format ***slack form***. If we write the linear program given in (29.33)–(29.37) in slack form, we obtain

$$\begin{aligned}
z &= 2x_1 - 3x_2 + 3x_3 && (29.38)\\
x_4 &= 7 - x_1 - x_2 + x_3 && (29.39)\\
x_5 &= -7 + x_1 + x_2 - x_3 && (29.40)\\
x_6 &= 4 - x_1 + 2x_2 - 2x_3 \ . && (29.41)
\end{aligned}$$

As with standard form, we find it convenient to have a more concise notation for describing a slack form. As we shall see in Section 29.3, the sets of basic and nonbasic variables will change as the simplex algorithm runs. We use N to denote the set of indices of the nonbasic variables and B to denote the set of indices of the basic variables. We always have that $|N| = n$, $|B| = m$, and $N \cup B = \{1, 2, \ldots, n + m\}$. The equations are indexed by the entries of B, and the variables on the right-hand sides are indexed by the entries of N. As in standard form, we use b_i, c_j, and a_{ij} to denote constant terms and coefficients. We also use v to denote an optional constant term in the objective function. (We shall see a little later that including the constant term in the objective function makes it easy to determine the value of the objective function.) Thus we can concisely define a slack form by a tuple (N, B, A, b, c, v), denoting the slack form

$$\begin{aligned}
z &= v + \sum_{j \in N} c_j x_j && (29.42)\\
x_i &= b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \ , && (29.43)
\end{aligned}$$

in which all variables x are constrained to be nonnegative. Because we subtract the sum $\sum_{j \in N} a_{ij} x_j$ in (29.43), the values a_{ij} are actually the negatives of the coefficients as they "appear" in the slack form.

For example, in the slack form

$$\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3}\\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3}\\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3}\\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2} \ ,
\end{aligned}$$

we have $B = \{1, 2, 4\}$, $N = \{3, 5, 6\}$,

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix},$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix},$$

$c = \begin{pmatrix} c_3 & c_5 & c_6 \end{pmatrix}^{\mathrm{T}} = \begin{pmatrix} -1/6 & -1/6 & -2/3 \end{pmatrix}^{\mathrm{T}}$, and $v = 28$. Note that the indices into A, b, and c are not necessarily sets of contiguous integers; they depend on the index sets B and N. As an example of the entries of A being the negatives of the coefficients as they appear in the slack form, observe that the equation for x_1 includes the term $x_3/6$, yet the coefficient a_{13} is actually $-1/6$ rather than $+1/6$.

Exercises

29.1-1

If we express the linear program in (29.24)–(29.28) in the compact notation of (29.19)–(29.21), what are n, m, A, b, and c?

29.1-2

Give three feasible solutions to the linear program in (29.24)–(29.28). What is the objective value of each one?

29.1-3

For the slack form in (29.38)–(29.41), what are N, B, A, b, c, and v?

29.1-4

Convert the following linear program into standard form:

$$\begin{aligned} \text{minimize} \quad & 2x_1 + 7x_2 + x_3 \\ \text{subject to} \quad & \\ & x_1 - x_3 = 7 \\ & 3x_1 + x_2 \geq 24 \\ & x_2 \geq 0 \\ & x_3 \leq 0 \,. \end{aligned}$$

29.1-5

Convert the following linear program into slack form:

$$\begin{aligned}
\text{maximize} \quad & 2x_1 - 6x_3 \\
\text{subject to} \quad & \\
& x_1 + x_2 - x_3 \le 7 \\
& 3x_1 - x_2 \ge 8 \\
& -x_1 + 2x_2 + 2x_3 \ge 0 \\
& x_1, x_2, x_3 \ge 0 \; .
\end{aligned}$$

What are the basic and nonbasic variables?

29.1-6

Show that the following linear program is infeasible:

$$\begin{aligned}
\text{maximize} \quad & 3x_1 - 2x_2 \\
\text{subject to} \quad & \\
& x_1 + x_2 \le 2 \\
& -2x_1 - 2x_2 \le -10 \\
& x_1, x_2 \ge 0 \; .
\end{aligned}$$

29.1-7

Show that the following linear program is unbounded:

$$\begin{aligned}
\text{maximize} \quad & x_1 - x_2 \\
\text{subject to} \quad & \\
& -2x_1 + x_2 \le -1 \\
& -x_1 - 2x_2 \le -2 \\
& x_1, x_2 \ge 0 \; .
\end{aligned}$$

29.1-8

Suppose that we have a general linear program with n variables and m constraints, and suppose that we convert it into standard form. Give an upper bound on the number of variables and constraints in the resulting linear program.

29.1-9

Give an example of a linear program for which the feasible region is not bounded, but the optimal objective value is finite.

29.2 Formulating problems as linear programs

Although we shall focus on the simplex algorithm in this chapter, it is also important to be able to recognize when we can formulate a problem as a linear program. Once we cast a problem as a polynomial-sized linear program, we can solve it in polynomial time by the ellipsoid algorithm or interior-point methods. Several linear-programming software packages can solve problems efficiently, so that once the problem is in the form of a linear program, such a package can solve it.

We shall look at several concrete examples of linear-programming problems. We start with two problems that we have already studied: the single-source shortest-paths problem (see Chapter 24) and the maximum-flow problem (see Chapter 26). We then describe the minimum-cost-flow problem. Although the minimum-cost-flow problem has a polynomial-time algorithm that is not based on linear programming, we won't describe the algorithm. Finally, we describe the multicommodity-flow problem, for which the only known polynomial-time algorithm is based on linear programming.

When we solved graph problems in Part VI, we used attribute notation, such as $v.d$ and $(u, v).f$. Linear programs typically use subscripted variables rather than objects with attached attributes, however. Therefore, when we express variables in linear programs, we shall indicate vertices and edges through subscripts. For example, we denote the shortest-path weight for vertex v not by $v.d$ but by d_v. Similarly, we denote the flow from vertex u to vertex v not by $(u, v).f$ but by f_{uv}. For quantities that are given as inputs to problems, such as edge weights or capacities, we shall continue to use notations such as $w(u, v)$ and $c(u.v)$.

Shortest paths

We can formulate the single-source shortest-paths problem as a linear program. In this section, we shall focus on how to formulate the single-pair shortest-path problem, leaving the extension to the more general single-source shortest-paths problem as Exercise 29.2-3.

In the single-pair shortest-path problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights, a source vertex s, and destination vertex t. We wish to compute the value d_t, which is the weight of a shortest path from s to t. To express this problem as a linear program, we need to determine a set of variables and constraints that define when we have a shortest path from s to t. Fortunately, the Bellman-Ford algorithm does exactly this. When the Bellman-Ford algorithm terminates, it has computed, for each vertex v, a value d_v (using subscript notation here rather than attribute notation) such that for each edge $(u, v) \in E$, we have $d_v \leq d_u + w(u, v)$.

The source vertex initially receives a value $d_s = 0$, which never changes. Thus we obtain the following linear program to compute the shortest-path weight from s to t:

$$\begin{aligned}
&\text{maximize} && d_t && && (29.44)\\
&\text{subject to} && && && \\
& && d_v \le d_u + w(u,v) && \text{for each edge } (u,v) \in E\ , && (29.45)\\
& && d_s = 0\ . && && (29.46)
\end{aligned}$$

You might be surprised that this linear program maximizes an objective function when it is supposed to compute shortest paths. We do not want to minimize the objective function, since then setting $\bar{d}_v = 0$ for all $v \in V$ would yield an optimal solution to the linear program without solving the shortest-paths problem. We maximize because an optimal solution to the shortest-paths problem sets each $\bar{d}_v$ to $\min_{u:(u,v)\in E}\{\bar{d}_u + w(u,v)\}$, so that $\bar{d}_v$ is the largest value that is less than or equal to all of the values in the set $\{\bar{d}_u + w(u,v)\}$. We want to maximize d_v for all vertices v on a shortest path from s to t subject to these constraints on all vertices v, and maximizing d_t achieves this goal.

This linear program has $|V|$ variables d_v, one for each vertex $v \in V$. It also has $|E| + 1$ constraints: one for each edge, plus the additional constraint that the source vertex's shortest-path weight always has the value 0.

Maximum flow

Next, we express the maximum-flow problem as a linear program. Recall that we are given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \ge 0$, and two distinguished vertices: a source s and a sink t. As defined in Section 26.1, a flow is a nonnegative real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the capacity constraint and flow conservation. A maximum flow is a flow that satisfies these constraints and maximizes the flow value, which is the total flow coming out of the source minus the total flow into the source. A flow, therefore, satisfies linear constraints, and the value of a flow is a linear function. Recalling also that we assume that $c(u, v) = 0$ if $(u, v) \notin E$ and that there are no antiparallel edges, we can express the maximum-flow problem as a linear program:

$$\begin{aligned}
&\text{maximize} && \sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} && && (29.47)\\
&\text{subject to} && && && \\
& && f_{uv} \le c(u,v) && \text{for each } u, v \in V\ , && (29.48)\\
& && \sum_{v\in V} f_{vu} = \sum_{v\in V} f_{uv} && \text{for each } u \in V - \{s,t\}\ , && (29.49)\\
& && f_{uv} \ge 0 && \text{for each } u, v \in V\ . && (29.50)
\end{aligned}$$

This linear program has $|V|^2$ variables, corresponding to the flow between each pair of vertices, and it has $2|V|^2 + |V| - 2$ constraints.

It is usually more efficient to solve a smaller-sized linear program. The linear program in (29.47)–(29.50) has, for ease of notation, a flow and capacity of 0 for each pair of vertices u, v with $(u, v) \notin E$. It would be more efficient to rewrite the linear program so that it has $O(V + E)$ constraints. Exercise 29.2-5 asks you to do so.

Minimum-cost flow

In this section, we have used linear programming to solve problems for which we already knew efficient algorithms. In fact, an efficient algorithm designed specifically for a problem, such as Dijkstra's algorithm for the single-source shortest-paths problem, or the push-relabel method for maximum flow, will often be more efficient than linear programming, both in theory and in practice.

The real power of linear programming comes from the ability to solve new problems. Recall the problem faced by the politician in the beginning of this chapter. The problem of obtaining a sufficient number of votes, while not spending too much money, is not solved by any of the algorithms that we have studied in this book, yet we can solve it by linear programming. Books abound with such real-world problems that linear programming can solve. Linear programming is also particularly useful for solving variants of problems for which we may not already know of an efficient algorithm.

Consider, for example, the following generalization of the maximum-flow problem. Suppose that, in addition to a capacity $c(u, v)$ for each edge (u, v), we are given a real-valued cost $a(u, v)$. As in the maximum-flow problem, we assume that $c(u, v) = 0$ if $(u, v) \notin E$, and that there are no antiparallel edges. If we send f_{uv} units of flow over edge (u, v), we incur a cost of $a(u, v) f_{uv}$. We are also given a flow demand d. We wish to send d units of flow from s to t while minimizing the total cost $\sum_{(u,v)\in E} a(u, v) f_{uv}$ incurred by the flow. This problem is known as the ***minimum-cost-flow problem***.

Figure 29.3(a) shows an example of the minimum-cost-flow problem. We wish to send 4 units of flow from s to t while incurring the minimum total cost. Any particular legal flow, that is, a function f satisfying constraints (29.48)–(29.49), incurs a total cost of $\sum_{(u,v)\in E} a(u, v) f_{uv}$. We wish to find the particular 4-unit flow that minimizes this cost. Figure 29.3(b) shows an optimal solution, with total cost $\sum_{(u,v)\in E} a(u, v) f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

There are polynomial-time algorithms specifically designed for the minimum-cost-flow problem, but they are beyond the scope of this book. We can, however, express the minimum-cost-flow problem as a linear program. The linear program looks similar to the one for the maximum-flow problem with the additional con-

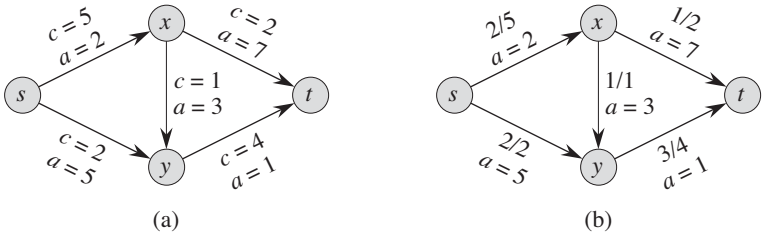


Figure 29.3 **(a)** An example of a minimum-cost-flow problem. We denote the capacities by c and the costs by a. Vertex s is the source and vertex t is the sink, and we wish to send 4 units of flow from s to t. **(b)** A solution to the minimum-cost flow problem in which 4 units of flow are sent from s to t. For each edge, the flow and capacity are written as flow/capacity.

straint that the value of the flow be exactly d units, and with the new objective function of minimizing the cost:

$$\text{minimize} \quad \sum_{(u,v)\in E} a(u,v)\,f_{uv} \tag{29.51}$$

subject to

$$\begin{aligned}
f_{uv} &\le c(u,v) && \text{for each } u, v \in V\ , \\
\sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} &= 0 && \text{for each } u \in V - \{s,t\}\ , \\
\sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} &= d\ , \\
f_{uv} &\ge 0 && \text{for each } u, v \in V\ .
\end{aligned} \tag{29.52}$$

Multicommodity flow

As a final example, we consider another flow problem. Suppose that the Lucky Puck company from Section 26.1 decides to diversify its product line and ship not only hockey pucks, but also hockey sticks and hockey helmets. Each piece of equipment is manufactured in its own factory, has its own warehouse, and must be shipped, each day, from factory to warehouse. The sticks are manufactured in Vancouver and must be shipped to Saskatoon, and the helmets are manufactured in Edmonton and must be shipped to Regina. The capacity of the shipping network does not change, however, and the different items, or ***commodities***, must share the same network.

This example is an instance of a ***multicommodity-flow problem***. In this problem, we are again given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \ge 0$. As in the maximum-flow problem, we implicitly assume that $c(u, v) = 0$ for $(u, v) \notin E$, and that the graph has no antipar-

allel edges. In addition, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where we specify commodity i by the triple $K_i = (s_i, t_i, d_i)$. Here, vertex s_i is the source of commodity i, vertex t_i is the sink of commodity i, and d_i is the demand for commodity i, which is the desired flow value for the commodity from s_i to t_i. We define a flow for commodity i, denoted by f_i, (so that f_{iuv} is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation and capacity constraints. We now define f_{uv}, the ***aggregate flow***, to be the sum of the various commodity flows, so that $f_{uv} = \sum_{i=1}^{k} f_{iuv}$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v). We are not trying to minimize any objective function in this problem; we need only determine whether such a flow exists. Thus, we write a linear program with a "null" objective function:

$$
\begin{aligned}
&\text{minimize} && 0 && && \\
&\text{subject to} && && && \\
& && \sum_{i=1}^{k} f_{iuv} && \le && c(u,v) && \text{for each } u, v \in V \ , \\
& \sum_{v \in V} f_{iuv} - \sum_{v \in V} f_{ivu} && && = && 0 && \text{for each } i = 1, 2, \ldots, k \text{ and} \\
& && && && && \text{for each } u \in V - \{s_i, t_i\} \ , \\
& \sum_{v \in V} f_{i,s_i,v} - \sum_{v \in V} f_{i,v,s_i} && && = && d_i && \text{for each } i = 1, 2, \ldots, k \ , \\
& && f_{iuv} && \ge && 0 && \text{for each } u, v \in V \text{ and} \\
& && && && && \text{for each } i = 1, 2, \ldots, k \ .
\end{aligned}
$$

The only known polynomial-time algorithm for this problem expresses it as a linear program and then solves it with a polynomial-time linear-programming algorithm.

Exercises

29.2-1

Put the single-pair shortest-path linear program from (29.44)–(29.46) into standard form.

29.2-2

Write out explicitly the linear program corresponding to finding the shortest path from node s to node y in Figure 24.2(a).

29.2-3

In the single-source shortest-paths problem, we want to find the shortest-path weights from a source vertex s to all vertices $v \in V$. Given a graph G, write a

linear program for which the solution has the property that d_v is the shortest-path weight from s to v for each vertex $v \in V$.

29.2-4
Write out explicitly the linear program corresponding to finding the maximum flow in Figure 26.1(a).

29.2-5
Rewrite the linear program for maximum flow (29.47)–(29.50) so that it uses only $O(V + E)$ constraints.

29.2-6
Write a linear program that, given a bipartite graph $G = (V, E)$, solves the maximum-bipartite-matching problem.

29.2-7
In the ***minimum-cost multicommodity-flow problem***, we are given directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$ and a cost $a(u, v)$. As in the multicommodity-flow problem, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where we specify commodity i by the triple $K_i = (s_i, t_i, d_i)$. We define the flow f_i for commodity i and the aggregate flow f_{uv} on edge (u, v) as in the multicommodity-flow problem. A feasible flow is one in which the aggregate flow on each edge (u, v) is no more than the capacity of edge (u, v). The cost of a flow is $\sum_{u,v \in V} a(u, v) f_{uv}$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

29.3 The simplex algorithm

The simplex algorithm is the classical method for solving linear programs. In contrast to most of the other algorithms in this book, its running time is not polynomial in the worst case. It does yield insight into linear programs, however, and is often remarkably fast in practice.

In addition to having a geometric interpretation, described earlier in this chapter, the simplex algorithm bears some similarity to Gaussian elimination, discussed in Section 28.1. Gaussian elimination begins with a system of linear equalities whose solution is unknown. In each iteration, we rewrite this system in an equivalent form that has some additional structure. After some number of iterations, we have rewritten the system so that the solution is simple to obtain. The simplex algorithm proceeds in a similar manner, and we can view it as Gaussian elimination for inequalities.

We now describe the main idea behind an iteration of the simplex algorithm. Associated with each iteration will be a "basic solution" that we can easily obtain from the slack form of the linear program: set each nonbasic variable to 0 and compute the values of the basic variables from the equality constraints. An iteration converts one slack form into an equivalent slack form. The objective value of the associated basic feasible solution will be no less than that at the previous iteration, and usually greater. To achieve this increase in the objective value, we choose a nonbasic variable such that if we were to increase that variable's value from 0, then the objective value would increase, too. The amount by which we can increase the variable is limited by the other constraints. In particular, we raise it until some basic variable becomes 0. We then rewrite the slack form, exchanging the roles of that basic variable and the chosen nonbasic variable. Although we have used a particular setting of the variables to guide the algorithm, and we shall use it in our proofs, the algorithm does not explicitly maintain this solution. It simply rewrites the linear program until an optimal solution becomes "obvious."

An example of the simplex algorithm

We begin with an extended example. Consider the following linear program in standard form:

$$
\begin{aligned}
&\text{maximize} && 3x_1 + x_2 + 2x_3 && && && (29.53)\\
&\text{subject to} && && && && \\
& && x_1 + x_2 + 3x_3 && \le && 30 && (29.54)\\
& && 2x_1 + 2x_2 + 5x_3 && \le && 24 && (29.55)\\
& && 4x_1 + x_2 + 2x_3 && \le && 36 && (29.56)\\
& && x_1, x_2, x_3 && \ge && 0\ . && (29.57)
\end{aligned}
$$

In order to use the simplex algorithm, we must convert the linear program into slack form; we saw how to do so in Section 29.1. In addition to being an algebraic manipulation, slack is a useful algorithmic concept. Recalling from Section 29.1 that each variable has a corresponding nonnegativity constraint, we say that an equality constraint is ***tight*** for a particular setting of its nonbasic variables if they cause the constraint's basic variable to become 0. Similarly, a setting of the nonbasic variables that would make a basic variable become negative ***violates*** that constraint. Thus, the slack variables explicitly maintain how far each constraint is from being tight, and so they help to determine how much we can increase values of nonbasic variables without violating any constraints.

Associating the slack variables x_4, x_5, and x_6 with inequalities (29.54)–(29.56), respectively, and putting the linear program into slack form, we obtain

$$
\begin{aligned}
z &= 3x_1 + x_2 + 2x_3 && (29.58)\\
x_4 &= 30 - x_1 - x_2 - 3x_3 && (29.59)\\
x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 && (29.60)\\
x_6 &= 36 - 4x_1 - x_2 - 2x_3 \, . && (29.61)
\end{aligned}
$$

The system of constraints (29.59)–(29.61) has 3 equations and 6 variables. Any setting of the variables x_1, x_2, and x_3 defines values for x_4, x_5, and x_6; therefore, we have an infinite number of solutions to this system of equations. A solution is feasible if all of $x_1, x_2, \ldots, x_6$ are nonnegative, and there can be an infinite number of feasible solutions as well. The infinite number of possible solutions to a system such as this one will be useful in later proofs. We focus on the ***basic solution***: set all the (nonbasic) variables on the right-hand side to 0 and then compute the values of the (basic) variables on the left-hand side. In this example, the basic solution is $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_6) = (0, 0, 0, 30, 24, 36)$ and it has objective value $z = (3 \cdot 0) + (1 \cdot 0) + (2 \cdot 0) = 0$. Observe that this basic solution sets $\bar{x}_i = b_i$ for each $i \in B$. An iteration of the simplex algorithm rewrites the set of equations and the objective function so as to put a different set of variables on the right-hand side. Thus, a different basic solution is associated with the rewritten problem. We emphasize that the rewrite does not in any way change the underlying linear-programming problem; the problem at one iteration has the identical set of feasible solutions as the problem at the previous iteration. The problem does, however, have a different basic solution than that of the previous iteration.

If a basic solution is also feasible, we call it a ***basic feasible solution***. As we run the simplex algorithm, the basic solution is almost always a basic feasible solution. We shall see in Section 29.5, however, that for the first few iterations of the simplex algorithm, the basic solution might not be feasible.

Our goal, in each iteration, is to reformulate the linear program so that the basic solution has a greater objective value. We select a nonbasic variable x_e whose coefficient in the objective function is positive, and we increase the value of x_e as much as possible without violating any of the constraints. The variable x_e becomes basic, and some other variable x_l becomes nonbasic. The values of other basic variables and of the objective function may also change.

To continue the example, let's think about increasing the value of x_1. As we increase x_1, the values of x_4, x_5, and x_6 all decrease. Because we have a nonnegativity constraint for each variable, we cannot allow any of them to become negative. If x_1 increases above 30, then x_4 becomes negative, and x_5 and x_6 become negative when x_1 increases above 12 and 9, respectively. The third constraint (29.61) is the tightest constraint, and it limits how much we can increase x_1. Therefore, we switch the roles of x_1 and x_6. We solve equation (29.61) for x_1 and obtain

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \, . \qquad (29.62)$$

To rewrite the other equations with x_6 on the right-hand side, we substitute for x_1 using equation (29.62). Doing so for equation (29.59), we obtain

$$\begin{aligned} x_4 &= 30 - x_1 - x_2 - 3x_3 \\ &= 30 - \left(9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}\right) - x_2 - 3x_3 \\ &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \,. \end{aligned} \tag{29.63}$$

Similarly, we combine equation (29.62) with constraint (29.60) and with objective function (29.58) to rewrite our linear program in the following form:

$$z = 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \tag{29.64}$$

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \tag{29.65}$$

$$x_4 = 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \tag{29.66}$$

$$x_5 = 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \,. \tag{29.67}$$

We call this operation a ***pivot***. As demonstrated above, a pivot chooses a nonbasic variable x_e, called the ***entering variable***, and a basic variable x_l, called the ***leaving variable***, and exchanges their roles.

The linear program described in equations (29.64)–(29.67) is equivalent to the linear program described in equations (29.58)–(29.61). We perform two operations in the simplex algorithm: rewrite equations so that variables move between the left-hand side and the right-hand side, and substitute one equation into another. The first operation trivially creates an equivalent problem, and the second, by elementary linear algebra, also creates an equivalent problem. (See Exercise 29.3-3.)

To demonstrate this equivalence, observe that our original basic solution $(0, 0, 0, 30, 24, 36)$ satisfies the new equations (29.65)–(29.67) and has objective value $27 + (1/4) \cdot 0 + (1/2) \cdot 0 - (3/4) \cdot 36 = 0$. The basic solution associated with the new linear program sets the nonbasic values to 0 and is $(9, 0, 0, 21, 6, 0)$, with objective value $z = 27$. Simple arithmetic verifies that this solution also satisfies equations (29.59)–(29.61) and, when plugged into objective function (29.58), has objective value $(3 \cdot 9) + (1 \cdot 0) + (2 \cdot 0) = 27$.

Continuing the example, we wish to find a new variable whose value we wish to increase. We do not want to increase x_6, since as its value increases, the objective value decreases. We can attempt to increase either x_2 or x_3; let us choose x_3. How far can we increase x_3 without violating any of the constraints? Constraint (29.65) limits it to 18, constraint (29.66) limits it to $42/5$, and constraint (29.67) limits it to $3/2$. The third constraint is again the tightest one, and therefore we rewrite the third constraint so that x_3 is on the left-hand side and x_5 is on the right-hand

side. We then substitute this new equation, $x_3 = 3/2 - 3x_2/8 - x_5/4 + x_6/8$, into equations (29.64)–(29.66) and obtain the new, but equivalent, system

$$z = \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \tag{29.68}$$

$$x_1 = \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \tag{29.69}$$

$$x_3 = \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \tag{29.70}$$

$$x_4 = \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} . \tag{29.71}$$

This system has the associated basic solution $(33/4, 0, 3/2, 69/4, 0, 0)$, with objective value $111/4$. Now the only way to increase the objective value is to increase x_2. The three constraints give upper bounds of 132, 4, and ∞, respectively. (We get an upper bound of ∞ from constraint (29.71) because, as we increase x_2, the value of the basic variable x_4 increases also. This constraint, therefore, places no restriction on how much we can increase x_2.) We increase x_2 to 4, and it becomes nonbasic. Then we solve equation (29.70) for x_2 and substitute in the other equations to obtain

$$z = 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \tag{29.72}$$

$$x_1 = 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \tag{29.73}$$

$$x_2 = 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \tag{29.74}$$

$$x_4 = 18 - \frac{x_3}{2} + \frac{x_5}{2} . \tag{29.75}$$

At this point, all coefficients in the objective function are negative. As we shall see later in this chapter, this situation occurs only when we have rewritten the linear program so that the basic solution is an optimal solution. Thus, for this problem, the solution $(8, 4, 0, 18, 0, 0)$, with objective value 28, is optimal. We can now return to our original linear program given in (29.53)–(29.57). The only variables in the original linear program are x_1, x_2, and x_3, and so our solution is $x_1 = 8$, $x_2 = 4$, and $x_3 = 0$, with objective value $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. Note that the values of the slack variables in the final solution measure how much slack remains in each inequality. Slack variable x_4 is 18, and in inequality (29.54), the left-hand side, with value $8 + 4 + 0 = 12$, is 18 less than the right-hand side of 30. Slack variables x_5 and x_6 are 0 and indeed, in inequalities (29.55) and (29.56), the left-hand and right-hand sides are equal. Observe also that even though the coefficients in the original slack form are integral, the coefficients in the other linear programs are not necessarily integral, and the intermediate solutions are not

necessarily integral. Furthermore, the final solution to a linear program need not be integral; it is purely coincidental that this example has an integral solution.

Pivoting

We now formalize the procedure for pivoting. The procedure PIVOT takes as input a slack form, given by the tuple (N, B, A, b, c, v), the index l of the leaving variable x_l, and the index e of the entering variable x_e. It returns the tuple $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$ describing the new slack form. (Recall again that the entries of the $m \times n$ matrices A and $\widehat{A}$ are actually the negatives of the coefficients that appear in the slack form.)

```
PIVOT(N, B, A, b, c, v, l, e)
 1  // Compute the coefficients of the equation for new basic variable x_e.
 2  let Â be a new m × n matrix
 3  b̂_e = b_l / a_le
 4  for each j ∈ N − {e}
 5      â_ej = a_lj / a_le
 6  â_el = 1 / a_le
 7  // Compute the coefficients of the remaining constraints.
 8  for each i ∈ B − {l}
 9      b̂_i = b_i − a_ie b̂_e
10      for each j ∈ N − {e}
11          â_ij = a_ij − a_ie â_ej
12      â_il = −a_ie â_el
13  // Compute the objective function.
14  v̂ = v + c_e b̂_e
15  for each j ∈ N − {e}
16      ĉ_j = c_j − c_e â_ej
17  ĉ_l = −c_e â_el
18  // Compute new sets of basic and nonbasic variables.
19  N̂ = N − {e} ∪ {l}
20  B̂ = B − {l} ∪ {e}
21  return (N̂, B̂, Â, b̂, ĉ, v̂)
```

PIVOT works as follows. Lines 3–6 compute the coefficients in the new equation for x_e by rewriting the equation that has x_l on the left-hand side to instead have x_e on the left-hand side. Lines 8–12 update the remaining equations by substituting the right-hand side of this new equation for each occurrence of x_e. Lines 14–17 do the same substitution for the objective function, and lines 19 and 20 update the

sets of nonbasic and basic variables. Line 21 returns the new slack form. As given, if $a_{le} = 0$, PIVOT would cause an error by dividing by 0, but as we shall see in the proofs of Lemmas 29.2 and 29.12, we call PIVOT only when $a_{le} \neq 0$.

We now summarize the effect that PIVOT has on the values of the variables in the basic solution.

Lemma 29.1

Consider a call to PIVOT(N, B, A, b, c, v, l, e) in which $a_{le} \neq 0$. Let the values returned from the call be $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{v})$, and let $\bar{x}$ denote the basic solution after the call. Then

1. $\bar{x}_j = 0$ for each $j \in \widehat{N}$.
2. $\bar{x}_e = b_l / a_{le}$.
3. $\bar{x}_i = b_i - a_{ie}\widehat{b}_e$ for each $i \in \widehat{B} - \{e\}$.

Proof The first statement is true because the basic solution always sets all nonbasic variables to 0. When we set each nonbasic variable to 0 in a constraint

$$x_i = \widehat{b}_i - \sum_{j \in \widehat{N}} \widehat{a}_{ij} x_j \ ,$$

we have that $\bar{x}_i = \widehat{b}_i$ for each $i \in \widehat{B}$. Since $e \in \widehat{B}$, line 3 of PIVOT gives

$$\bar{x}_e = \widehat{b}_e = b_l / a_{le} \ ,$$

which proves the second statement. Similarly, using line 9 for each $i \in \widehat{B} - \{e\}$, we have

$$\bar{x}_i = \widehat{b}_i = b_i - a_{ie}\widehat{b}_e \ ,$$

which proves the third statement. ■

The formal simplex algorithm

We are now ready to formalize the simplex algorithm, which we demonstrated by example. That example was a particularly nice one, and we could have had several other issues to address:

- How do we determine whether a linear program is feasible?
- What do we do if the linear program is feasible, but the initial basic solution is not feasible?
- How do we determine whether a linear program is unbounded?
- How do we choose the entering and leaving variables?

In Section 29.5, we shall show how to determine whether a problem is feasible, and if so, how to find a slack form in which the initial basic solution is feasible. Therefore, let us assume that we have a procedure INITIALIZE-SIMPLEX(A, b, c) that takes as input a linear program in standard form, that is, an $m \times n$ matrix $A = (a_{ij})$, an m-vector $b = (b_i)$, and an n-vector $c = (c_j)$. If the problem is infeasible, the procedure returns a message that the program is infeasible and then terminates. Otherwise, the procedure returns a slack form for which the initial basic solution is feasible.

The procedure SIMPLEX takes as input a linear program in standard form, as just described. It returns an n-vector $\bar{x} = (\bar{x}_j)$ that is an optimal solution to the linear program described in (29.19)–(29.21).

```
SIMPLEX(A, b, c)
 1  (N, B, A, b, c, v) = INITIALIZE-SIMPLEX(A, b, c)
 2  let Δ be a new vector of length n
 3  while some index j ∈ N has c_j > 0
 4      choose an index e ∈ N for which c_e > 0
 5      for each index i ∈ B
 6          if a_ie > 0
 7              Δ_i = b_i / a_ie
 8          else Δ_i = ∞
 9      choose an index l ∈ B that minimizes Δ_i
10      if Δ_l == ∞
11          return "unbounded"
12      else (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, e)
13  for i = 1 to n
14      if i ∈ B
15          x̄_i = b_i
16      else x̄_i = 0
17  return (x̄_1, x̄_2, ..., x̄_n)
```

The SIMPLEX procedure works as follows. In line 1, it calls the procedure INITIALIZE-SIMPLEX(A, b, c), described above, which either determines that the linear program is infeasible or returns a slack form for which the basic solution is feasible. The **while** loop of lines 3–12 forms the main part of the algorithm. If all coefficients in the objective function are negative, then the **while** loop terminates. Otherwise, line 4 selects a variable x_e, whose coefficient in the objective function is positive, as the entering variable. Although we may choose any such variable as the entering variable, we assume that we use some prespecified deterministic rule. Next, lines 5–9 check each constraint and pick the one that most severely limits the amount by which we can increase x_e without violating any of the nonnegativ-

ity constraints; the basic variable associated with this constraint is x_l. Again, we are free to choose one of several variables as the leaving variable, but we assume that we use some prespecified deterministic rule. If none of the constraints limits the amount by which the entering variable can increase, the algorithm returns "unbounded" in line 11. Otherwise, line 12 exchanges the roles of the entering and leaving variables by calling PIVOT(N, B, A, b, c, v, l, e), as described above. Lines 13–16 compute a solution $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ for the original linear-programming variables by setting all the nonbasic variables to 0 and each basic variable $\bar{x}_i$ to b_i, and line 17 returns these values.

To show that SIMPLEX is correct, we first show that if SIMPLEX has an initial feasible solution and eventually terminates, then it either returns a feasible solution or determines that the linear program is unbounded. Then, we show that SIMPLEX terminates. Finally, in Section 29.4 (Theorem 29.10) we show that the solution returned is optimal.

Lemma 29.2

Given a linear program (A, b, c), suppose that the call to INITIALIZE-SIMPLEX in line 1 of SIMPLEX returns a slack form for which the basic solution is feasible. Then if SIMPLEX returns a solution in line 17, that solution is a feasible solution to the linear program. If SIMPLEX returns "unbounded" in line 11, the linear program is unbounded.

Proof We use the following three-part loop invariant:

At the start of each iteration of the **while** loop of lines 3–12,

1. the slack form is equivalent to the slack form returned by the call of INITIALIZE-SIMPLEX,
2. for each $i \in B$, we have $b_i \geq 0$, and
3. the basic solution associated with the slack form is feasible.

Initialization: The equivalence of the slack forms is trivial for the first iteration. We assume, in the statement of the lemma, that the call to INITIALIZE-SIMPLEX in line 1 of SIMPLEX returns a slack form for which the basic solution is feasible. Thus, the third part of the invariant is true. Because the basic solution is feasible, each basic variable x_i is nonnegative. Furthermore, since the basic solution sets each basic variable x_i to b_i, we have that $b_i \geq 0$ for all $i \in B$. Thus, the second part of the invariant holds.

Maintenance: We shall show that each iteration of the **while** loop maintains the loop invariant, assuming that the **return** statement in line 11 does not execute. We shall handle the case in which line 11 executes when we discuss termination.

An iteration of the **while** loop exchanges the role of a basic and a nonbasic variable by calling the PIVOT procedure. By Exercise 29.3-3, the slack form is equivalent to the one from the previous iteration which, by the loop invariant, is equivalent to the initial slack form.

We now demonstrate the second part of the loop invariant. We assume that at the start of each iteration of the **while** loop, $b_i \geq 0$ for each $i \in B$, and we shall show that these inequalities remain true after the call to PIVOT in line 12. Since the only changes to the variables b_i and the set B of basic variables occur in this assignment, it suffices to show that line 12 maintains this part of the invariant. We let b_i, a_{ij}, and B refer to values before the call of PIVOT, and $\widehat{b}_i$ refer to values returned from PIVOT.

First, we observe that $\widehat{b}_e \geq 0$ because $b_l \geq 0$ by the loop invariant, $a_{le} > 0$ by lines 6 and 9 of SIMPLEX, and $\widehat{b}_e = b_l/a_{le}$ by line 3 of PIVOT.

For the remaining indices $i \in B - \{l\}$, we have that

$$\begin{aligned}\widehat{b}_i &= b_i - a_{ie}\widehat{b}_e && \text{(by line 9 of PIVOT)} \\ &= b_i - a_{ie}(b_l/a_{le}) && \text{(by line 3 of PIVOT)} \ . \end{aligned} \tag{29.76}$$

We have two cases to consider, depending on whether $a_{ie} > 0$ or $a_{ie} \leq 0$. If $a_{ie} > 0$, then since we chose l such that

$$b_l/a_{le} \leq b_i/a_{ie} \quad \text{for all } i \in B \ , \tag{29.77}$$

we have

$$\begin{aligned}\widehat{b}_i &= b_i - a_{ie}(b_l/a_{le}) && \text{(by equation (29.76))} \\ &\geq b_i - a_{ie}(b_i/a_{ie}) && \text{(by inequality (29.77))} \\ &= b_i - b_i \\ &= 0 \ , \end{aligned}$$

and thus $\widehat{b}_i \geq 0$. If $a_{ie} \leq 0$, then because a_{le}, b_i, and b_l are all nonnegative, equation (29.76) implies that $\widehat{b}_i$ must be nonnegative, too.

We now argue that the basic solution is feasible, i.e., that all variables have nonnegative values. The nonbasic variables are set to 0 and thus are nonnegative. Each basic variable x_i is defined by the equation

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \ .$$

The basic solution sets $\bar{x}_i = b_i$. Using the second part of the loop invariant, we conclude that each basic variable $\bar{x}_i$ is nonnegative.

Termination: The **while** loop can terminate in one of two ways. If it terminates because of the condition in line 3, then the current basic solution is feasible and line 17 returns this solution. The other way it terminates is by returning "unbounded" in line 11. In this case, for each iteration of the **for** loop in lines 5–8, when line 6 is executed, we find that $a_{ie} \le 0$. Consider the solution $\bar{x}$ defined as

$$\bar{x}_i = \begin{cases} \infty & \text{if } i = e \ , \\ 0 & \text{if } i \in N - \{e\} \ , \\ b_i - \sum_{j \in N} a_{ij} \bar{x}_j & \text{if } i \in B \ . \end{cases}$$

We now show that this solution is feasible, i.e., that all variables are nonnegative. The nonbasic variables other than $\bar{x}_e$ are 0, and $\bar{x}_e = \infty > 0$; thus all nonbasic variables are nonnegative. For each basic variable $\bar{x}_i$, we have

$$\begin{aligned} \bar{x}_i &= b_i - \sum_{j \in N} a_{ij} \bar{x}_j \\ &= b_i - a_{ie} \bar{x}_e \ . \end{aligned}$$

The loop invariant implies that $b_i \ge 0$, and we have $a_{ie} \le 0$ and $\bar{x}_e = \infty > 0$. Thus, $\bar{x}_i \ge 0$.

Now we show that the objective value for the solution $\bar{x}$ is unbounded. From equation (29.42), the objective value is

$$\begin{aligned} z &= v + \sum_{j \in N} c_j \bar{x}_j \\ &= v + c_e \bar{x}_e \ . \end{aligned}$$

Since $c_e > 0$ (by line 4 of SIMPLEX) and $\bar{x}_e = \infty$, the objective value is ∞, and thus the linear program is unbounded. ■

It remains to show that SIMPLEX terminates, and when it does terminate, the solution it returns is optimal. Section 29.4 will address optimality. We now discuss termination.

Termination

In the example given in the beginning of this section, each iteration of the simplex algorithm increased the objective value associated with the basic solution. As Exercise 29.3-2 asks you to show, no iteration of SIMPLEX can decrease the objective value associated with the basic solution. Unfortunately, it is possible that an iteration leaves the objective value unchanged. This phenomenon is called ***degeneracy***, and we shall now study it in greater detail.

The assignment in line 14 of PIVOT, $\hat{\nu} = \nu + c_e\hat{b}_e$, changes the objective value. Since SIMPLEX calls PIVOT only when $c_e > 0$, the only way for the objective value to remain unchanged (i.e., $\hat{\nu} = \nu$) is for $\hat{b}_e$ to be 0. This value is assigned as $\hat{b}_e = b_l/a_{le}$ in line 3 of PIVOT. Since we always call PIVOT with $a_{le} \neq 0$, we see that for $\hat{b}_e$ to equal 0, and hence the objective value to be unchanged, we must have $b_l = 0$.

Indeed, this situation can occur. Consider the linear program

$$\begin{aligned} z &= x_1 + x_2 + x_3 \\ x_4 &= 8 - x_1 - x_2 \\ x_5 &= x_2 - x_3 \,. \end{aligned}$$

Suppose that we choose x_1 as the entering variable and x_4 as the leaving variable. After pivoting, we obtain

$$\begin{aligned} z &= 8 + x_3 - x_4 \\ x_1 &= 8 - x_2 - x_4 \\ x_5 &= x_2 - x_3 \,. \end{aligned}$$

At this point, our only choice is to pivot with x_3 entering and x_5 leaving. Since $b_5 = 0$, the objective value of 8 remains unchanged after pivoting:

$$\begin{aligned} z &= 8 + x_2 - x_4 - x_5 \\ x_1 &= 8 - x_2 - x_4 \\ x_3 &= x_2 - x_5 \,. \end{aligned}$$

The objective value has not changed, but our slack form has. Fortunately, if we pivot again, with x_2 entering and x_1 leaving, the objective value increases (to 16), and the simplex algorithm can continue.

Degeneracy can prevent the simplex algorithm from terminating, because it can lead to a phenomenon known as ***cycling***: the slack forms at two different iterations of SIMPLEX are identical. Because of degeneracy, SIMPLEX could choose a sequence of pivot operations that leave the objective value unchanged but repeat a slack form within the sequence. Since SIMPLEX is a deterministic algorithm, if it cycles, then it will cycle through the same series of slack forms forever, never terminating.

Cycling is the only reason that SIMPLEX might not terminate. To show this fact, we must first develop some additional machinery.

At each iteration, SIMPLEX maintains A, b, c, and ν in addition to the sets N and B. Although we need to explicitly maintain A, b, c, and ν in order to implement the simplex algorithm efficiently, we can get by without maintaining them. In other words, the sets of basic and nonbasic variables suffice to uniquely determine the slack form. Before proving this fact, we prove a useful algebraic lemma.

Lemma 29.3
Let I be a set of indices. For each $j \in I$, let α_j and β_j be real numbers, and let x_j be a real-valued variable. Let γ be any real number. Suppose that for any settings of the x_j, we have

$$\sum_{j \in I} \alpha_j x_j = \gamma + \sum_{j \in I} \beta_j x_j \ . \tag{29.78}$$

Then $\alpha_j = \beta_j$ for each $j \in I$, and $\gamma = 0$.

Proof Since equation (29.78) holds for any values of the x_j, we can use particular values to draw conclusions about α, β, and γ. If we let $x_j = 0$ for each $j \in I$, we conclude that $\gamma = 0$. Now pick an arbitrary index $j \in I$, and set $x_j = 1$ and $x_k = 0$ for all $k \neq j$. Then we must have $\alpha_j = \beta_j$. Since we picked j as any index in I, we conclude that $\alpha_j = \beta_j$ for each $j \in I$. ∎

A particular linear program has many different slack forms; recall that each slack form has the same set of feasible and optimal solutions as the original linear program. We now show that the slack form of a linear program is uniquely determined by the set of basic variables. That is, given the set of basic variables, a unique slack form (unique set of coefficients and right-hand sides) is associated with those basic variables.

Lemma 29.4
Let (A, b, c) be a linear program in standard form. Given a set B of basic variables, the associated slack form is uniquely determined.

Proof Assume for the purpose of contradiction that there are two different slack forms with the same set B of basic variables. The slack forms must also have identical sets $N = \{1, 2, \ldots, n + m\} - B$ of nonbasic variables. We write the first slack form as

$$z = v + \sum_{j \in N} c_j x_j \tag{29.79}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \ , \tag{29.80}$$

and the second as

$$z = v' + \sum_{j \in N} c'_j x_j \tag{29.81}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B \ . \tag{29.82}$$

Consider the system of equations formed by subtracting each equation in line (29.82) from the corresponding equation in line (29.80). The resulting system is

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \quad \text{for } i \in B$$

or, equivalently,

$$\sum_{j \in N} a_{ij} x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B \ .$$

Now, for each $i \in B$, apply Lemma 29.3 with $\alpha_j = a_{ij}$, $\beta_j = a'_{ij}$, $\gamma = b_i - b'_i$, and $I = N$. Since $\alpha_i = \beta_i$, we have that $a_{ij} = a'_{ij}$ for each $j \in N$, and since $\gamma = 0$, we have that $b_i = b'_i$. Thus, for the two slack forms, A and b are identical to A' and b'. Using a similar argument, Exercise 29.3-1 shows that it must also be the case that $c = c'$ and $v = v'$, and hence that the slack forms must be identical. ■

We can now show that cycling is the only possible reason that SIMPLEX might not terminate.

Lemma 29.5
If SIMPLEX fails to terminate in at most $\binom{n+m}{m}$ iterations, then it cycles.

Proof By Lemma 29.4, the set B of basic variables uniquely determines a slack form. There are $n + m$ variables and $|B| = m$, and therefore, there are at most $\binom{n+m}{m}$ ways to choose B. Thus, there are only at most $\binom{n+m}{m}$ unique slack forms. Therefore, if SIMPLEX runs for more than $\binom{n+m}{m}$ iterations, it must cycle. ■

Cycling is theoretically possible, but extremely rare. We can prevent it by choosing the entering and leaving variables somewhat more carefully. One option is to perturb the input slightly so that it is impossible to have two solutions with the same objective value. Another option is to break ties by always choosing the variable with the smallest index, a strategy known as ***Bland's rule***. We omit the proof that these strategies avoid cycling.

Lemma 29.6
If lines 4 and 9 of SIMPLEX always break ties by choosing the variable with the smallest index, then SIMPLEX must terminate. ■

We conclude this section with the following lemma.

Lemma 29.7
Assuming that INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible, SIMPLEX either reports that a linear program is unbounded, or it terminates with a feasible solution in at most $\binom{n+m}{m}$ iterations.

Proof Lemmas 29.2 and 29.6 show that if INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible, SIMPLEX either reports that a linear program is unbounded, or it terminates with a feasible solution. By the contrapositive of Lemma 29.5, if SIMPLEX terminates with a feasible solution, then it terminates in at most $\binom{n+m}{m}$ iterations. ■

Exercises

29.3-1
Complete the proof of Lemma 29.4 by showing that it must be the case that $c = c'$ and $\nu = \nu'$.

29.3-2
Show that the call to PIVOT in line 12 of SIMPLEX never decreases the value of ν.

29.3-3
Prove that the slack form given to the PIVOT procedure and the slack form that the procedure returns are equivalent.

29.3-4
Suppose we convert a linear program (A, b, c) in standard form to slack form. Show that the basic solution is feasible if and only if $b_i \geq 0$ for $i = 1, 2, \ldots, m$.

29.3-5
Solve the following linear program using SIMPLEX:

$$
\begin{aligned}
&\text{maximize} && 18x_1 + 12.5x_2 && \\
&\text{subject to} && && \\
& && x_1 + x_2 && \leq 20 \\
& && x_1 && \leq 12 \\
& && x_2 && \leq 16 \\
& && x_1, x_2 && \geq 0 \ .
\end{aligned}
$$

29.3-6

Solve the following linear program using SIMPLEX:

$$\begin{aligned}
&\text{maximize} && 5x_1 - 3x_2 \\
&\text{subject to} \\
&&& x_1 - x_2 \le 1 \\
&&& 2x_1 + x_2 \le 2 \\
&&& x_1, x_2 \ge 0 \ .
\end{aligned}$$

29.3-7

Solve the following linear program using SIMPLEX:

$$\begin{aligned}
&\text{minimize} && x_1 + x_2 + x_3 \\
&\text{subject to} \\
&&& 2x_1 + 7.5x_2 + 3x_3 \ge 10000 \\
&&& 20x_1 + 5x_2 + 10x_3 \ge 30000 \\
&&& x_1, x_2, x_3 \ge 0 \ .
\end{aligned}$$

29.3-8

In the proof of Lemma 29.5, we argued that there are at most $\binom{m+n}{n}$ ways to choose a set B of basic variables. Give an example of a linear program in which there are strictly fewer than $\binom{m+n}{n}$ ways to choose the set B.

29.4 Duality

We have proven that, under certain assumptions, SIMPLEX terminates. We have not yet shown that it actually finds an optimal solution to a linear program, however. In order to do so, we introduce a powerful concept called ***linear-programming duality***.

Duality enables us to prove that a solution is indeed optimal. We saw an example of duality in Chapter 26 with Theorem 26.6, the max-flow min-cut theorem. Suppose that, given an instance of a maximum-flow problem, we find a flow f with value $|f|$. How do we know whether f is a maximum flow? By the max-flow min-cut theorem, if we can find a cut whose value is also $|f|$, then we have verified that f is indeed a maximum flow. This relationship provides an example of duality: given a maximization problem, we define a related minimization problem such that the two problems have the same optimal objective values.

Given a linear program in which the objective is to maximize, we shall describe how to formulate a ***dual*** linear program in which the objective is to minimize and

whose optimal value is identical to that of the original linear program. When referring to dual linear programs, we call the original linear program the ***primal***.

Given a primal linear program in standard form, as in (29.16)–(29.18), we define the dual linear program as

$$\text{minimize} \quad \sum_{i=1}^{m} b_i y_i \tag{29.83}$$

subject to

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad \text{for } j = 1, 2, \ldots, n , \tag{29.84}$$

$$y_i \geq 0 \quad \text{for } i = 1, 2, \ldots, m . \tag{29.85}$$

To form the dual, we change the maximization to a minimization, exchange the roles of coefficients on the right-hand sides and the objective function, and replace each less-than-or-equal-to by a greater-than-or-equal-to. Each of the m constraints in the primal has an associated variable y_i in the dual, and each of the n constraints in the dual has an associated variable x_j in the primal. For example, consider the linear program given in (29.53)–(29.57). The dual of this linear program is

$$\text{minimize} \quad 30y_1 + 24y_2 + 36y_3 \tag{29.86}$$

subject to

$$y_1 + 2y_2 + 4y_3 \geq 3 \tag{29.87}$$

$$y_1 + 2y_2 + y_3 \geq 1 \tag{29.88}$$

$$3y_1 + 5y_2 + 2y_3 \geq 2 \tag{29.89}$$

$$y_1, y_2, y_3 \geq 0 . \tag{29.90}$$

We shall show in Theorem 29.10 that the optimal value of the dual linear program is always equal to the optimal value of the primal linear program. Furthermore, the simplex algorithm actually implicitly solves both the primal and the dual linear programs simultaneously, thereby providing a proof of optimality.

We begin by demonstrating ***weak duality***, which states that any feasible solution to the primal linear program has a value no greater than that of any feasible solution to the dual linear program.

Lemma 29.8 (Weak linear-programming duality)
Let $\bar{x}$ be any feasible solution to the primal linear program in (29.16)–(29.18) and let $\bar{y}$ be any feasible solution to the dual linear program in (29.83)–(29.85). Then, we have

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i .$$

Proof We have

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\le \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \quad \text{(by inequalities (29.84))} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\
&\le \sum_{i=1}^{m} b_i \bar{y}_i \qquad \text{(by inequalities (29.17))} .
\end{aligned}$$

■

Corollary 29.9

Let $\bar{x}$ be a feasible solution to a primal linear program (A, b, c), and let $\bar{y}$ be a feasible solution to the corresponding dual linear program. If

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i ,$$

then $\bar{x}$ and $\bar{y}$ are optimal solutions to the primal and dual linear programs, respectively.

Proof By Lemma 29.8, the objective value of a feasible solution to the primal cannot exceed that of a feasible solution to the dual. The primal linear program is a maximization problem and the dual is a minimization problem. Thus, if feasible solutions $\bar{x}$ and $\bar{y}$ have the same objective value, neither can be improved. ■

Before proving that there always is a dual solution whose value is equal to that of an optimal primal solution, we describe how to find such a solution. When we ran the simplex algorithm on the linear program in (29.53)–(29.57), the final iteration yielded the slack form (29.72)–(29.75) with objective $z = 28 - x_3/6 - x_5/6 - 2x_6/3$, $B = \{1, 2, 4\}$, and $N = \{3, 5, 6\}$. As we shall show below, the basic solution associated with the final slack form is indeed an optimal solution to the linear program; an optimal solution to linear program (29.53)–(29.57) is therefore $(\bar{x}_1, \bar{x}_2, \bar{x}_3) = (8, 4, 0)$, with objective value $(3 \cdot 8) + (1 \cdot 4) + (2 \cdot 0) = 28$. As we also show below, we can read off an optimal dual solution: the negatives of the coefficients of the primal objective function are the values of the dual variables. More precisely, suppose that the last slack form of the primal is

$$\begin{aligned}
z &= \nu' + \sum_{j \in N} c'_j x_j \\
x_i &= b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B .
\end{aligned}$$

Then, to produce an optimal dual solution, we set

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{if } (n+i) \in N \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{29.91}$$

Thus, an optimal solution to the dual linear program defined in (29.86)–(29.90) is $\bar{y}_1 = 0$ (since $n + 1 = 4 \in B$), $\bar{y}_2 = -c'_5 = 1/6$, and $\bar{y}_3 = -c'_6 = 2/3$. Evaluating the dual objective function (29.86), we obtain an objective value of $(30 \cdot 0) + (24 \cdot (1/6)) + (36 \cdot (2/3)) = 28$, which confirms that the objective value of the primal is indeed equal to the objective value of the dual. Combining these calculations with Lemma 29.8 yields a proof that the optimal objective value of the primal linear program is 28. We now show that this approach applies in general: we can find an optimal solution to the dual and simultaneously prove that a solution to the primal is optimal.

Theorem 29.10 (Linear-programming duality)
Suppose that SIMPLEX returns values $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$ for the primal linear program (A, b, c). Let N and B denote the nonbasic and basic variables for the final slack form, let c' denote the coefficients in the final slack form, and let $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ be defined by equation (29.91). Then $\bar{x}$ is an optimal solution to the primal linear program, $\bar{y}$ is an optimal solution to the dual linear program, and

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i \ . \tag{29.92}$$

Proof By Corollary 29.9, if we can find feasible solutions $\bar{x}$ and $\bar{y}$ that satisfy equation (29.92), then $\bar{x}$ and $\bar{y}$ must be optimal primal and dual solutions. We shall now show that the solutions $\bar{x}$ and $\bar{y}$ described in the statement of the theorem satisfy equation (29.92).

Suppose that we run SIMPLEX on a primal linear program, as given in lines (29.16)–(29.18). The algorithm proceeds through a series of slack forms until it terminates with a final slack form with objective function

$$z = v' + \sum_{j \in N} c'_j x_j \ . \tag{29.93}$$

Since SIMPLEX terminated with a solution, by the condition in line 3 we know that

$$c'_j \le 0 \quad \text{for all } j \in N \ . \tag{29.94}$$

If we define

$$c'_j = 0 \quad \text{for all } j \in B\ , \tag{29.95}$$

we can rewrite equation (29.93) as

$$\begin{aligned} z &= v' + \sum_{j \in N} c'_j x_j \\ &= v' + \sum_{j \in N} c'_j x_j + \sum_{j \in B} c'_j x_j \quad (\text{because } c'_j = 0 \text{ if } j \in B) \\ &= v' + \sum_{j=1}^{n+m} c'_j x_j \qquad\qquad (\text{because } N \cup B = \{1, 2, \ldots, n+m\})\ . \end{aligned} \tag{29.96}$$

For the basic solution $\bar{x}$ associated with this final slack form, $\bar{x}_j = 0$ for all $j \in N$, and $z = v'$. Since all slack forms are equivalent, if we evaluate the original objective function on $\bar{x}$, we must obtain the same objective value:

$$\begin{aligned} \sum_{j=1}^{n} c_j \bar{x}_j &= v' + \sum_{j=1}^{n+m} c'_j \bar{x}_j && (29.97) \\ &= v' + \sum_{j \in N} c'_j \bar{x}_j + \sum_{j \in B} c'_j \bar{x}_j \\ &= v' + \sum_{j \in N} (c'_j \cdot 0) + \sum_{j \in B} (0 \cdot \bar{x}_j) && (29.98) \\ &= v'\ . \end{aligned}$$

We shall now show that $\bar{y}$, defined by equation (29.91), is feasible for the dual linear program and that its objective value $\sum_{i=1}^{m} b_i \bar{y}_i$ equals $\sum_{j=1}^{n} c_j \bar{x}_j$. Equation (29.97) says that the first and last slack forms, evaluated at $\bar{x}$, are equal. More generally, the equivalence of all slack forms implies that for *any* set of values $x = (x_1, x_2, \ldots, x_n)$, we have

$$\sum_{j=1}^{n} c_j x_j = v' + \sum_{j=1}^{n+m} c'_j x_j\ .$$

Therefore, for any particular set of values $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$, we have

$$
\begin{aligned}
&\sum_{j=1}^{n} c_j \bar{x}_j \\
&= \nu' + \sum_{j=1}^{n+m} c'_j \bar{x}_j \\
&= \nu' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{j=n+1}^{n+m} c'_j \bar{x}_j \\
&= \nu' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} c'_{n+i} \bar{x}_{n+i} \\
&= \nu' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i)\, \bar{x}_{n+i} && \text{(by equations (29.91) and (29.95))} \\
&= \nu' + \sum_{j=1}^{n} c'_j \bar{x}_j + \sum_{i=1}^{m} (-\bar{y}_i) \left(b_i - \sum_{j=1}^{n} a_{ij} \bar{x}_j \right) && \text{(by equation (29.32))} \\
&= \nu' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{i=1}^{m} \sum_{j=1}^{n} (a_{ij} \bar{x}_j)\, \bar{y}_i \\
&= \nu' + \sum_{j=1}^{n} c'_j \bar{x}_j - \sum_{i=1}^{m} b_i \bar{y}_i + \sum_{j=1}^{n} \sum_{i=1}^{m} (a_{ij} \bar{y}_i)\, \bar{x}_j \\
&= \left(\nu' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \,,
\end{aligned}
$$

so that

$$
\sum_{j=1}^{n} c_j \bar{x}_j = \left(\nu' - \sum_{i=1}^{m} b_i \bar{y}_i \right) + \sum_{j=1}^{n} \left(c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \,. \tag{29.99}
$$

Applying Lemma 29.3 to equation (29.99), we obtain

$$
\nu' - \sum_{i=1}^{m} b_i \bar{y}_i = 0 \,, \tag{29.100}
$$

$$
c'_j + \sum_{i=1}^{m} a_{ij} \bar{y}_i = c_j \quad \text{for } j = 1, 2, \ldots, n \,. \tag{29.101}
$$

By equation (29.100), we have that $\sum_{i=1}^{m} b_i \bar{y}_i = \nu'$, and hence the objective value of the dual $\left(\sum_{i=1}^{m} b_i \bar{y}_i \right)$ is equal to that of the primal (ν'). It remains to show

that the solution $\bar{y}$ is feasible for the dual problem. From inequalities (29.94) and equations (29.95), we have that $c'_j \le 0$ for all $j = 1, 2, \ldots, n+m$. Hence, for any $j = 1, 2, \ldots, n$, equations (29.101) imply that

$$\begin{aligned} c_j &= c'_j + \sum_{i=1}^{m} a_{ij}\bar{y}_i \\ &\le \sum_{i=1}^{m} a_{ij}\bar{y}_i \ , \end{aligned}$$

which satisfies the constraints (29.84) of the dual. Finally, since $c'_j \le 0$ for each $j \in N \cup B$, when we set $\bar{y}$ according to equation (29.91), we have that each $\bar{y}_i \ge 0$, and so the nonnegativity constraints are satisfied as well. ■

We have shown that, given a feasible linear program, if INITIALIZE-SIMPLEX returns a feasible solution, and if SIMPLEX terminates without returning "unbounded," then the solution returned is indeed an optimal solution. We have also shown how to construct an optimal solution to the dual linear program.

Exercises

29.4-1
Formulate the dual of the linear program given in Exercise 29.3-5.

29.4-2
Suppose that we have a linear program that is not in standard form. We could produce the dual by first converting it to standard form, and then taking the dual. It would be more convenient, however, to be able to produce the dual directly. Explain how we can directly take the dual of an arbitrary linear program.

29.4-3
Write down the dual of the maximum-flow linear program, as given in lines (29.47)–(29.50) on page 860. Explain how to interpret this formulation as a minimum-cut problem.

29.4-4
Write down the dual of the minimum-cost-flow linear program, as given in lines (29.51)–(29.52) on page 862. Explain how to interpret this problem in terms of graphs and flows.

29.4-5
Show that the dual of the dual of a linear program is the primal linear program.

29.4-6
Which result from Chapter 26 can be interpreted as weak duality for the maximum-flow problem?

29.5 The initial basic feasible solution

In this section, we first describe how to test whether a linear program is feasible, and if it is, how to produce a slack form for which the basic solution is feasible. We conclude by proving the fundamental theorem of linear programming, which says that the SIMPLEX procedure always produces the correct result.

Finding an initial solution

In Section 29.3, we assumed that we had a procedure INITIALIZE-SIMPLEX that determines whether a linear program has any feasible solutions, and if it does, gives a slack form for which the basic solution is feasible. We describe this procedure here.

A linear program can be feasible, yet the initial basic solution might not be feasible. Consider, for example, the following linear program:

$$\begin{aligned}
&\text{maximize} && 2x_1 - x_2 && && && (29.102)\\
&\text{subject to} && && && && \\
& && 2x_1 - x_2 && \le && 2 && (29.103)\\
& && x_1 - 5x_2 && \le && -4 && (29.104)\\
& && x_1, x_2 && \ge && 0\ . && (29.105)
\end{aligned}$$

If we were to convert this linear program to slack form, the basic solution would set $x_1 = 0$ and $x_2 = 0$. This solution violates constraint (29.104), and so it is not a feasible solution. Thus, INITIALIZE-SIMPLEX cannot just return the obvious slack form. In order to determine whether a linear program has any feasible solutions, we will formulate an ***auxiliary linear program***. For this auxiliary linear program, we can find (with a little work) a slack form for which the basic solution is feasible. Furthermore, the solution of this auxiliary linear program determines whether the initial linear program is feasible and if so, it provides a feasible solution with which we can initialize SIMPLEX.

Lemma 29.11
Let L be a linear program in standard form, given as in (29.16)–(29.18). Let x_0 be a new variable, and let L_{aux} be the following linear program with $n + 1$ variables:

$$\text{maximize} \qquad -x_0 \tag{29.106}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i \quad \text{for } i = 1, 2, \ldots, m\,, \tag{29.107}$$

$$x_j \geq 0 \quad \text{for } j = 0, 1, \ldots, n\,. \tag{29.108}$$

Then L is feasible if and only if the optimal objective value of L_{aux} is 0.

Proof Suppose that L has a feasible solution $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$. Then the solution $\bar{x}_0 = 0$ combined with $\bar{x}$ is a feasible solution to L_{aux} with objective value 0. Since $x_0 \geq 0$ is a constraint of L_{aux} and the objective function is to maximize $-x_0$, this solution must be optimal for L_{aux}.

Conversely, suppose that the optimal objective value of L_{aux} is 0. Then $\bar{x}_0 = 0$, and the remaining solution values of $\bar{x}$ satisfy the constraints of L. ■

We now describe our strategy to find an initial basic feasible solution for a linear program L in standard form:

INITIALIZE-SIMPLEX(A, b, c)

```
1  let k be the index of the minimum b_i
2  if b_k ≥ 0                    // is the initial basic solution feasible?
3      return ({1, 2, ..., n}, {n + 1, n + 2, ..., n + m}, A, b, c, 0)
4  form L_aux by adding −x_0 to the left-hand side of each constraint
       and setting the objective function to −x_0
5  let (N, B, A, b, c, v) be the resulting slack form for L_aux
6  l = n + k
7  // L_aux has n + 1 nonbasic variables and m basic variables.
8  (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, 0)
9  // The basic solution is now feasible for L_aux.
10 iterate the while loop of lines 3–12 of SIMPLEX until an optimal solution
       to L_aux is found
11 if the optimal solution to L_aux sets x̄_0 to 0
12     if x̄_0 is basic
13         perform one (degenerate) pivot to make it nonbasic
14     from the final slack form of L_aux, remove x_0 from the constraints and
           restore the original objective function of L, but replace each basic
           variable in this objective function by the right-hand side of its
           associated constraint
15     return the modified final slack form
16 else return "infeasible"
```

INITIALIZE-SIMPLEX works as follows. In lines 1–3, we implicitly test the basic solution to the initial slack form for L given by $N = \{1, 2, \ldots, n\}$, $B = \{n+1, n+2, \ldots, n+m\}$, $\bar{x}_i = b_i$ for all $i \in B$, and $\bar{x}_j = 0$ for all $j \in N$. (Creating the slack form requires no explicit effort, as the values of A, b, and c are the same in both slack and standard forms.) If line 2 finds this basic solution to be feasible—that is, $\bar{x}_i \geq 0$ for all $i \in N \cup B$—then line 3 returns the slack form. Otherwise, in line 4, we form the auxiliary linear program L_{aux} as in Lemma 29.11. Since the initial basic solution to L is not feasible, the initial basic solution to the slack form for L_{aux} cannot be feasible either. To find a basic feasible solution, we perform a single pivot operation. Line 6 selects $l = n + k$ as the index of the basic variable that will be the leaving variable in the upcoming pivot operation. Since the basic variables are $x_{n+1}, x_{n+2}, \ldots, x_{n+m}$, the leaving variable x_l will be the one with the most negative value. Line 8 performs that call of PIVOT, with x_0 entering and x_l leaving. We shall see shortly that the basic solution resulting from this call of PIVOT will be feasible. Now that we have a slack form for which the basic solution is feasible, we can, in line 10, repeatedly call PIVOT to fully solve the auxiliary linear program. As the test in line 11 demonstrates, if we find an optimal solution to L_{aux} with objective value 0, then in lines 12–14, we create a slack form for L for which the basic solution is feasible. To do so, we first, in lines 12–13, handle the degenerate case in which x_0 may still be basic with value $\bar{x}_0 = 0$. In this case, we perform a pivot step to remove x_0 from the basis, using any $e \in N$ such that $a_{0e} \neq 0$ as the entering variable. The new basic solution remains feasible; the degenerate pivot does not change the value of any variable. Next we delete all x_0 terms from the constraints and restore the original objective function for L. The original objective function may contain both basic and nonbasic variables. Therefore, in the objective function we replace each basic variable by the right-hand side of its associated constraint. Line 15 then returns this modified slack form. If, on the other hand, line 11 discovers that the original linear program L is infeasible, then line 16 returns this information.

We now demonstrate the operation of INITIALIZE-SIMPLEX on the linear program (29.102)–(29.105). This linear program is feasible if we can find nonnegative values for x_1 and x_2 that satisfy inequalities (29.103) and (29.104). Using Lemma 29.11, we formulate the auxiliary linear program

$$\begin{aligned}
&\text{maximize} && -x_0 && && && && && && (29.109)\\
&\text{subject to} \\
&& 2x_1 &- x_2 &- x_0 &\leq 2 && && && && (29.110)\\
&& x_1 &- 5x_2 &- x_0 &\leq -4 && && && && (29.111)\\
&& x_1, x_2, x_0 & && &\geq 0\ . && && && &
\end{aligned}$$

By Lemma 29.11, if the optimal objective value of this auxiliary linear program is 0, then the original linear program has a feasible solution. If the optimal objective

value of this auxiliary linear program is negative, then the original linear program does not have a feasible solution.

We write this linear program in slack form, obtaining

$$\begin{aligned} z &= && && && - && x_0 \\ x_3 &= && 2 && - && 2x_1 && + && x_2 && + && x_0 \\ x_4 &= && -4 && - && x_1 && + && 5x_2 && + && x_0 \ . \end{aligned}$$

We are not out of the woods yet, because the basic solution, which would set $x_4 = -4$, is not feasible for this auxiliary linear program. We can, however, with one call to PIVOT, convert this slack form into one in which the basic solution is feasible. As line 8 indicates, we choose x_0 to be the entering variable. In line 6, we choose as the leaving variable x_4, which is the basic variable whose value in the basic solution is most negative. After pivoting, we have the slack form

$$\begin{aligned} z &= && -4 && - && x_1 && + && 5x_2 && - && x_4 \\ x_0 &= && 4 && + && x_1 && - && 5x_2 && + && x_4 \\ x_3 &= && 6 && - && x_1 && - && 4x_2 && + && x_4 \ . \end{aligned}$$

The associated basic solution is $(\bar{x}_0, \bar{x}_1, \bar{x}_2, \bar{x}_3, \bar{x}_4) = (4, 0, 0, 6, 0)$, which is feasible. We now repeatedly call PIVOT until we obtain an optimal solution to L_{aux}. In this case, one call to PIVOT with x_2 entering and x_0 leaving yields

$$\begin{aligned} z &= && && - && x_0 \\ x_2 &= && \frac{4}{5} && - && \frac{x_0}{5} && + && \frac{x_1}{5} && + && \frac{x_4}{5} \\ x_3 &= && \frac{14}{5} && + && \frac{4x_0}{5} && - && \frac{9x_1}{5} && + && \frac{x_4}{5} \ . \end{aligned}$$

This slack form is the final solution to the auxiliary problem. Since this solution has $x_0 = 0$, we know that our initial problem was feasible. Furthermore, since $x_0 = 0$, we can just remove it from the set of constraints. We then restore the original objective function, with appropriate substitutions made to include only nonbasic variables. In our example, we get the objective function

$$2x_1 - x_2 = 2x_1 - \left(\frac{4}{5} - \frac{x_0}{5} + \frac{x_1}{5} + \frac{x_4}{5}\right) \ .$$

Setting $x_0 = 0$ and simplifying, we get the objective function

$$-\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \ ,$$

and the slack form

$$\begin{aligned} z &= -\frac{4}{5} + \frac{9x_1}{5} - \frac{x_4}{5} \\ x_2 &= \frac{4}{5} + \frac{x_1}{5} + \frac{x_4}{5} \\ x_3 &= \frac{14}{5} - \frac{9x_1}{5} + \frac{x_4}{5} \; . \end{aligned}$$

This slack form has a feasible basic solution, and we can return it to procedure SIMPLEX.

We now formally show the correctness of INITIALIZE-SIMPLEX.

Lemma 29.12

If a linear program L has no feasible solution, then INITIALIZE-SIMPLEX returns "infeasible." Otherwise, it returns a valid slack form for which the basic solution is feasible.

Proof First suppose that the linear program L has no feasible solution. Then by Lemma 29.11, the optimal objective value of L_{aux}, defined in (29.106)–(29.108), is nonzero, and by the nonnegativity constraint on x_0, the optimal objective value must be negative. Furthermore, this objective value must be finite, since setting $x_i = 0$, for $i = 1, 2, \ldots, n$, and $x_0 = |\min_{i=1}^{m} \{b_i\}|$ is feasible, and this solution has objective value $-|\min_{i=1}^{m} \{b_i\}|$. Therefore, line 10 of INITIALIZE-SIMPLEX finds a solution with a nonpositive objective value. Let $\bar{x}$ be the basic solution associated with the final slack form. We cannot have $\bar{x}_0 = 0$, because then L_{aux} would have objective value 0, which contradicts that the objective value is negative. Thus the test in line 11 results in line 16 returning "infeasible."

Suppose now that the linear program L does have a feasible solution. From Exercise 29.3-4, we know that if $b_i \geq 0$ for $i = 1, 2, \ldots, m$, then the basic solution associated with the initial slack form is feasible. In this case, lines 2–3 return the slack form associated with the input. (Converting the standard form to slack form is easy, since A, b, and c are the same in both.)

In the remainder of the proof, we handle the case in which the linear program is feasible but we do not return in line 3. We argue that in this case, lines 4–10 find a feasible solution to L_{aux} with objective value 0. First, by lines 1–2, we must have

$$b_k < 0 \; ,$$

and

$$b_k \leq b_i \quad \text{for each } i \in B \; . \tag{29.112}$$

In line 8, we perform one pivot operation in which the leaving variable x_l (recall that $l = n + k$, so that $b_l < 0$) is the left-hand side of the equation with minimum b_i, and the entering variable is x_0, the extra added variable. We now show

that after this pivot, all entries of b are nonnegative, and hence the basic solution to L_{aux} is feasible. Letting $\bar{x}$ be the basic solution after the call to PIVOT, and letting $\widehat{b}$ and $\widehat{B}$ be values returned by PIVOT, Lemma 29.1 implies that

$$\bar{x}_i = \begin{cases} b_i - a_{ie}\widehat{b}_e & \text{if } i \in \widehat{B} - \{e\} \,, \\ b_l / a_{le} & \text{if } i = e \,. \end{cases} \tag{29.113}$$

The call to PIVOT in line 8 has $e = 0$. If we rewrite inequalities (29.107), to include coefficients a_{i0},

$$\sum_{j=0}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \,, \tag{29.114}$$

then

$$a_{i0} = a_{ie} = -1 \quad \text{for each } i \in B \,. \tag{29.115}$$

(Note that a_{i0} is the coefficient of x_0 as it appears in inequalities (29.114), not the negation of the coefficient, because L_{aux} is in standard rather than slack form.) Since $l \in B$, we also have that $a_{le} = -1$. Thus, $b_l / a_{le} > 0$, and so $\bar{x}_e > 0$. For the remaining basic variables, we have

$$\begin{aligned} \bar{x}_i &= b_i - a_{ie}\widehat{b}_e && \text{(by equation (29.113))} \\ &= b_i - a_{ie}(b_l / a_{le}) && \text{(by line 3 of PIVOT)} \\ &= b_i - b_l && \text{(by equation (29.115) and } a_{le} = -1) \\ &\ge 0 && \text{(by inequality (29.112)) ,} \end{aligned}$$

which implies that each basic variable is now nonnegative. Hence the basic solution after the call to PIVOT in line 8 is feasible. We next execute line 10, which solves L_{aux}. Since we have assumed that L has a feasible solution, Lemma 29.11 implies that L_{aux} has an optimal solution with objective value 0. Since all the slack forms are equivalent, the final basic solution to L_{aux} must have $\bar{x}_0 = 0$, and after removing x_0 from the linear program, we obtain a slack form that is feasible for L. Line 15 then returns this slack form. ■

Fundamental theorem of linear programming

We conclude this chapter by showing that the SIMPLEX procedure works. In particular, any linear program either is infeasible, is unbounded, or has an optimal solution with a finite objective value. In each case, SIMPLEX acts appropriately.

Theorem 29.13 (Fundamental theorem of linear programming)
Any linear program L, given in standard form, either

1. has an optimal solution with a finite objective value,
2. is infeasible, or
3. is unbounded.

If L is infeasible, SIMPLEX returns "infeasible." If L is unbounded, SIMPLEX returns "unbounded." Otherwise, SIMPLEX returns an optimal solution with a finite objective value.

Proof By Lemma 29.12, if linear program L is infeasible, then SIMPLEX returns "infeasible." Now suppose that the linear program L is feasible. By Lemma 29.12, INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible. By Lemma 29.7, therefore, SIMPLEX either returns "unbounded" or terminates with a feasible solution. If it terminates with a finite solution, then Theorem 29.10 tells us that this solution is optimal. On the other hand, if SIMPLEX returns "unbounded," Lemma 29.2 tells us the linear program L is indeed unbounded. Since SIMPLEX always terminates in one of these ways, the proof is complete. ■

Exercises

29.5-1
Give detailed pseudocode to implement lines 5 and 14 of INITIALIZE-SIMPLEX.

29.5-2
Show that when the main loop of SIMPLEX is run by INITIALIZE-SIMPLEX, it can never return "unbounded."

29.5-3
Suppose that we are given a linear program L in standard form, and suppose that for both L and the dual of L, the basic solutions associated with the initial slack forms are feasible. Show that the optimal objective value of L is 0.

29.5-4
Suppose that we allow strict inequalities in a linear program. Show that in this case, the fundamental theorem of linear programming does not hold.

29.5-5

Solve the following linear program using SIMPLEX:

$$\begin{aligned}
\text{maximize} \quad & x_1 + 3x_2 \\
\text{subject to} \quad & \\
& x_1 - x_2 \le 8 \\
& -x_1 - x_2 \le -3 \\
& -x_1 + 4x_2 \le 2 \\
& x_1, x_2 \ge 0 \ .
\end{aligned}$$

29.5-6

Solve the following linear program using SIMPLEX:

$$\begin{aligned}
\text{maximize} \quad & x_1 - 2x_2 \\
\text{subject to} \quad & \\
& x_1 + 2x_2 \le 4 \\
& -2x_1 - 6x_2 \le -12 \\
& x_2 \le 1 \\
& x_1, x_2 \ge 0 \ .
\end{aligned}$$

29.5-7

Solve the following linear program using SIMPLEX:

$$\begin{aligned}
\text{maximize} \quad & x_1 + 3x_2 \\
\text{subject to} \quad & \\
& -x_1 + x_2 \le -1 \\
& -x_1 - x_2 \le -3 \\
& -x_1 + 4x_2 \le 2 \\
& x_1, x_2 \ge 0 \ .
\end{aligned}$$

29.5-8

Solve the linear program given in (29.6)–(29.10).

29.5-9

Consider the following 1-variable linear program, which we call P:

$$\begin{aligned}
\text{maximize} \quad & tx \\
\text{subject to} \quad & \\
& rx \le s \\
& x \ge 0 \ ,
\end{aligned}$$

where r, s, and t are arbitrary real numbers. Let D be the dual of P.

State for which values of r, s, and t you can assert that

1. Both P and D have optimal solutions with finite objective values.
2. P is feasible, but D is infeasible.
3. D is feasible, but P is infeasible.
4. Neither P nor D is feasible.

Problems

29-1 Linear-inequality feasibility

Given a set of m linear inequalities on n variables $x_1, x_2, \ldots, x_n$, the ***linear-inequality feasibility problem*** asks whether there is a setting of the variables that simultaneously satisfies each of the inequalities.

a. Show that if we have an algorithm for linear programming, we can use it to solve a linear-inequality feasibility problem. The number of variables and constraints that you use in the linear-programming problem should be polynomial in n and m.

b. Show that if we have an algorithm for the linear-inequality feasibility problem, we can use it to solve a linear-programming problem. The number of variables and linear inequalities that you use in the linear-inequality feasibility problem should be polynomial in n and m, the number of variables and constraints in the linear program.

29-2 Complementary slackness

Complementary slackness describes a relationship between the values of primal variables and dual constraints and between the values of dual variables and primal constraints. Let $\bar{x}$ be a feasible solution to the primal linear program given in (29.16)–(29.18), and let $\bar{y}$ be a feasible solution to the dual linear program given in (29.83)–(29.85). Complementary slackness states that the following conditions are necessary and sufficient for $\bar{x}$ and $\bar{y}$ to be optimal:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ or } \bar{x}_j = 0 \quad \text{for } j = 1, 2, \ldots, n$$

and

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \text{ or } \bar{y}_i = 0 \quad \text{for } i = 1, 2, \ldots, m\ .$$

a. Verify that complementary slackness holds for the linear program in lines (29.53)–(29.57).

b. Prove that complementary slackness holds for any primal linear program and its corresponding dual.

c. Prove that a feasible solution $\bar{x}$ to a primal linear program given in lines (29.16)–(29.18) is optimal if and only if there exist values $\bar{y} = (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_m)$ such that

1. $\bar{y}$ is a feasible solution to the dual linear program given in (29.83)–(29.85),
2. $\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j$ for all j such that $\bar{x}_j > 0$, and
3. $\bar{y}_i = 0$ for all i such that $\sum_{j=1}^{n} a_{ij}\bar{x}_j < b_i$.

29-3 Integer linear programming

An ***integer linear-programming problem*** is a linear-programming problem with the additional constraint that the variables x must take on integral values. Exercise 34.5-3 shows that just determining whether an integer linear program has a feasible solution is NP-hard, which means that there is no known polynomial-time algorithm for this problem.

a. Show that weak duality (Lemma 29.8) holds for an integer linear program.

b. Show that duality (Theorem 29.10) does not always hold for an integer linear program.

c. Given a primal linear program in standard form, let us define P to be the optimal objective value for the primal linear program, D to be the optimal objective value for its dual, IP to be the optimal objective value for the integer version of the primal (that is, the primal with the added constraint that the variables take on integer values), and ID to be the optimal objective value for the integer version of the dual. Assuming that both the primal integer program and the dual integer program are feasible and bounded, show that

$$IP \le P = D \le ID\ .$$

29-4 Farkas's lemma

Let A be an $m \times n$ matrix and c be an n-vector. Then Farkas's lemma states that exactly one of the systems

$$\begin{aligned} Ax &\leq 0\,, \\ c^{\mathrm{T}}x &> 0 \end{aligned}$$

and

$$\begin{aligned} A^{\mathrm{T}}y &= c\,, \\ y &\geq 0 \end{aligned}$$

is solvable, where x is an n-vector and y is an m-vector. Prove Farkas's lemma.

29-5 Minimum-cost circulation

In this problem, we consider a variant of the minimum-cost-flow problem from Section 29.2 in which we are not given a demand, a source, or a sink. Instead, we are given, as before, a flow network and edge costs $a(u, v)$. A flow is feasible if it satisfies the capacity constraint on every edge and flow conservation at *every* vertex. The goal is to find, among all feasible flows, the one of minimum cost. We call this problem the ***minimum-cost-circulation problem.***

a. Formulate the minimum-cost-circulation problem as a linear program.

b. Suppose that for all edges $(u, v) \in E$, we have $a(u, v) > 0$. Characterize an optimal solution to the minimum-cost-circulation problem.

c. Formulate the maximum-flow problem as a minimum-cost-circulation problem linear program. That is given a maximum-flow problem instance $G = (V, E)$ with source s, sink t and edge capacities c, create a minimum-cost-circulation problem by giving a (possibly different) network $G' = (V', E')$ with edge capacities c' and edge costs a' such that you can discern a solution to the maximum-flow problem from a solution to the minimum-cost-circulation problem.

d. Formulate the single-source shortest-path problem as a minimum-cost-circulation problem linear program.

Chapter notes

This chapter only begins to study the wide field of linear programming. A number of books are devoted exclusively to linear programming, including those by Chvátal [69], Gass [130], Karloff [197], Schrijver [303], and Vanderbei [344]. Many other books give a good coverage of linear programming, including those by Papadimitriou and Steiglitz [271] and Ahuja, Magnanti, and Orlin [7]. The coverage in this chapter draws on the approach taken by Chvátal.

The simplex algorithm for linear programming was invented by G. Dantzig in 1947. Shortly after, researchers discovered how to formulate a number of problems in a variety of fields as linear programs and solve them with the simplex algorithm. As a result, applications of linear programming flourished, along with several algorithms. Variants of the simplex algorithm remain the most popular methods for solving linear-programming problems. This history appears in a number of places, including the notes in [69] and [197].

The ellipsoid algorithm was the first polynomial-time algorithm for linear programming and is due to L. G. Khachian in 1979; it was based on earlier work by N. Z. Shor, D. B. Judin, and A. S. Nemirovskii. Grötschel, Lovász, and Schrijver [154] describe how to use the ellipsoid algorithm to solve a variety of problems in combinatorial optimization. To date, the ellipsoid algorithm does not appear to be competitive with the simplex algorithm in practice.

Karmarkar's paper [198] includes a description of the first interior-point algorithm. Many subsequent researchers designed interior-point algorithms. Good surveys appear in the article of Goldfarb and Todd [141] and the book by Ye [361].

Analysis of the simplex algorithm remains an active area of research. V. Klee and G. J. Minty constructed an example on which the simplex algorithm runs through $2^n - 1$ iterations. The simplex algorithm usually performs very well in practice and many researchers have tried to give theoretical justification for this empirical observation. A line of research begun by K. H. Borgwardt, and carried on by many others, shows that under certain probabilistic assumptions on the input, the simplex algorithm converges in expected polynomial time. Spielman and Teng [322] made progress in this area, introducing the "smoothed analysis of algorithms" and applying it to the simplex algorithm.

The simplex algorithm is known to run efficiently in certain special cases. Particularly noteworthy is the network-simplex algorithm, which is the simplex algorithm, specialized to network-flow problems. For certain network problems, including the shortest-paths, maximum-flow, and minimum-cost-flow problems, variants of the network-simplex algorithm run in polynomial time. See, for example, the article by Orlin [268] and the citations therein.

30 Polynomials and the FFT

The straightforward method of adding two polynomials of degree n takes $\Theta(n)$ time, but the straightforward method of multiplying them takes $\Theta(n^2)$ time. In this chapter, we shall show how the fast Fourier transform, or FFT, can reduce the time to multiply polynomials to $\Theta(n \lg n)$.

The most common use for Fourier transforms, and hence the FFT, is in signal processing. A signal is given in the ***time domain***: as a function mapping time to amplitude. Fourier analysis allows us to express the signal as a weighted sum of phase-shifted sinusoids of varying frequencies. The weights and phases associated with the frequencies characterize the signal in the ***frequency domain***. Among the many everyday applications of FFT's are compression techniques used to encode digital video and audio information, including MP3 files. Several fine books delve into the rich area of signal processing; the chapter notes reference a few of them.

Polynomials

A ***polynomial*** in the variable x over an algebraic field F represents a function $A(x)$ as a formal sum:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

We call the values $a_0, a_1, \ldots, a_{n-1}$ the ***coefficients*** of the polynomial. The coefficients are drawn from a field F, typically the set $\mathbb{C}$ of complex numbers. A polynomial $A(x)$ has ***degree*** k if its highest nonzero coefficient is a_k; we write that degree$(A) = k$. Any integer strictly greater than the degree of a polynomial is a ***degree-bound*** of that polynomial. Therefore, the degree of a polynomial of degree-bound n may be any integer between 0 and $n - 1$, inclusive.

We can define a variety of operations on polynomials. For ***polynomial addition***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***sum*** is a polyno-

mial $C(x)$, also of degree-bound n, such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and

$$B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

then

$$C(x) = \sum_{j=0}^{n-1} c_j x^j ,$$

where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$. For example, if we have the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, then $C(x) = 4x^3 + 7x^2 - 6x + 4$.

For ***polynomial multiplication***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***product*** $C(x)$ is a polynomial of degree-bound $2n-1$ such that $C(x) = A(x)B(x)$ for all x in the underlying field. You probably have multiplied polynomials before, by multiplying each term in $A(x)$ by each term in $B(x)$ and then combining terms with equal powers. For example, we can multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows:

$$\begin{array}{rrrrrrrrr}
 & & & & 6x^3 & + & 7x^2 & - 10x & + & 9 \\
 & & & - & 2x^3 & & & + 4x & - & 5 \\
\hline
 & & & - & 30x^3 & - & 35x^2 & + 50x & - & 45 \\
 & & 24x^4 & + & 28x^3 & - & 40x^2 & + 36x & & \\
- 12x^6 & - & 14x^5 & + & 20x^4 & - & 18x^3 & & & \\
\hline
- 12x^6 & - & 14x^5 & + & 44x^4 & - & 20x^3 & - 75x^2 & + 86x & - 45
\end{array}$$

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j , \tag{30.1}$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} . \tag{30.2}$$

Note that degree(C) = degree(A) + degree(B), implying that if A is a polynomial of degree-bound n_a and B is a polynomial of degree-bound n_b, then C is a polynomial of degree-bound $n_a + n_b - 1$. Since a polynomial of degree-bound k is also a polynomial of degree-bound $k + 1$, we will normally say that the product polynomial C is a polynomial of degree-bound $n_a + n_b$.

Chapter outline

Section 30.1 presents two ways to represent polynomials: the coefficient representation and the point-value representation. The straightforward methods for multiplying polynomials—equations (30.1) and (30.2)—take $\Theta(n^2)$ time when we represent polynomials in coefficient form, but only $\Theta(n)$ time when we represent them in point-value form. We can, however, multiply polynomials using the coefficient representation in only $\Theta(n \lg n)$ time by converting between the two representations. To see why this approach works, we must first study complex roots of unity, which we do in Section 30.2. Then, we use the FFT and its inverse, also described in Section 30.2, to perform the conversions. Section 30.3 shows how to implement the FFT quickly in both serial and parallel models.

This chapter uses complex numbers extensively, and within this chapter we use the symbol i exclusively to denote $\sqrt{-1}$.

30.1 Representing polynomials

The coefficient and point-value representations of polynomials are in a sense equivalent; that is, a polynomial in point-value form has a unique counterpart in coefficient form. In this section, we introduce the two representations and show how to combine them so that we can multiply two degree-bound n polynomials in $\Theta(n \lg n)$ time.

Coefficient representation

A ***coefficient representation*** of a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree-bound n is a vector of coefficients $a = (a_0, a_1, \ldots, a_{n-1})$. In matrix equations in this chapter, we shall generally treat vectors as column vectors.

The coefficient representation is convenient for certain operations on polynomials. For example, the operation of ***evaluating*** the polynomial $A(x)$ at a given point x_0 consists of computing the value of $A(x_0)$. We can evaluate a polynomial in $\Theta(n)$ time using ***Horner's rule***:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-2} + x_0(a_{n-1}))\cdots)) \, .$$

Similarly, adding two polynomials represented by the coefficient vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$ takes $\Theta(n)$ time: we just produce the coefficient vector $c = (c_0, c_1, \ldots, c_{n-1})$, where $c_j = a_j + b_j$ for $j = 0, 1, \ldots, n-1$.

Now, consider multiplying two degree-bound n polynomials $A(x)$ and $B(x)$ represented in coefficient form. If we use the method described by equations (30.1) and (30.2), multiplying polynomials takes time $\Theta(n^2)$, since we must multiply each coefficient in the vector a by each coefficient in the vector b. The operation of multiplying polynomials in coefficient form seems to be considerably more difficult than that of evaluating a polynomial or adding two polynomials. The resulting coefficient vector c, given by equation (30.2), is also called the ***convolution*** of the input vectors a and b, denoted $c = a \otimes b$. Since multiplying polynomials and computing convolutions are fundamental computational problems of considerable practical importance, this chapter concentrates on efficient algorithms for them.

Point-value representation

A ***point-value representation*** of a polynomial $A(x)$ of degree-bound n is a set of n ***point-value pairs***

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

such that all of the x_k are distinct and

$$y_k = A(x_k) \tag{30.3}$$

for $k = 0, 1, \ldots, n-1$. A polynomial has many different point-value representations, since we can use any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all we have to do is select n distinct points $x_0, x_1, \ldots, x_{n-1}$ and then evaluate $A(x_k)$ for $k = 0, 1, \ldots, n-1$. With Horner's method, evaluating a polynomial at n points takes time $\Theta(n^2)$. We shall see later that if we choose the points x_k cleverly, we can accelerate this computation to run in time $\Theta(n \lg n)$.

The inverse of evaluation—determining the coefficient form of a polynomial from a point-value representation—is ***interpolation***. The following theorem shows that interpolation is well defined when the desired interpolating polynomial must have a degree-bound equal to the given number of point-value pairs.

Theorem 30.1 (Uniqueness of an interpolating polynomial)
For any set $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ of n point-value pairs such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k)$ for $k = 0, 1, \ldots, n-1$.

Proof The proof relies on the existence of the inverse of a certain matrix. Equation (30.3) is equivalent to the matrix equation

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} . \tag{30.4}$$

The matrix on the left is denoted $V(x_0, x_1, \ldots, x_{n-1})$ and is known as a Vandermonde matrix. By Problem D-1, this matrix has determinant

$$\prod_{0 \le j < k \le n-1} (x_k - x_j) ,$$

and therefore, by Theorem D.5, it is invertible (that is, nonsingular) if the x_k are distinct. Thus, we can solve for the coefficients a_j uniquely given the point-value representation:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y . \qquad \blacksquare$$

The proof of Theorem 30.1 describes an algorithm for interpolation based on solving the set (30.4) of linear equations. Using the LU decomposition algorithms of Chapter 28, we can solve these equations in time $O(n^3)$.

A faster algorithm for n-point interpolation is based on ***Lagrange's formula***:

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \ne k} (x - x_j)}{\prod_{j \ne k} (x_k - x_j)} . \tag{30.5}$$

You may wish to verify that the right-hand side of equation (30.5) is a polynomial of degree-bound n that satisfies $A(x_k) = y_k$ for all k. Exercise 30.1-5 asks you how to compute the coefficients of A using Lagrange's formula in time $\Theta(n^2)$.

Thus, n-point evaluation and interpolation are well-defined inverse operations that transform between the coefficient representation of a polynomial and a point-value representation.[1] The algorithms described above for these problems take time $\Theta(n^2)$.

The point-value representation is quite convenient for many operations on polynomials. For addition, if $C(x) = A(x) + B(x)$, then $C(x_k) = A(x_k) + B(x_k)$ for any point x_k. More precisely, if we have a point-value representation for A,

[1] Interpolation is a notoriously tricky problem from the point of view of numerical stability. Although the approaches described here are mathematically correct, small differences in the inputs or round-off errors during computation can cause large differences in the result.

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} ,$$

and for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\}$$

(note that A and B are evaluated at the *same* n points), then a point-value representation for C is

$$\{(x_0, y_0 + y'_0), (x_1, y_1 + y'_1), \ldots, (x_{n-1}, y_{n-1} + y'_{n-1})\} .$$

Thus, the time to add two polynomials of degree-bound n in point-value form is $\Theta(n)$.

Similarly, the point-value representation is convenient for multiplying polynomials. If $C(x) = A(x)B(x)$, then $C(x_k) = A(x_k)B(x_k)$ for any point x_k, and we can pointwise multiply a point-value representation for A by a point-value representation for B to obtain a point-value representation for C. We must face the problem, however, that degree(C) = degree(A) + degree(B); if A and B are of degree-bound n, then C is of degree-bound $2n$. A standard point-value representation for A and B consists of n point-value pairs for each polynomial. When we multiply these together, we get n point-value pairs, but we need $2n$ pairs to interpolate a unique polynomial C of degree-bound $2n$. (See Exercise 30.1-4.) We must therefore begin with "extended" point-value representations for A and for B consisting of $2n$ point-value pairs each. Given an extended point-value representation for A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\} ,$$

and a corresponding extended point-value representation for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\} ,$$

then a point-value representation for C is

$$\{(x_0, y_0 y'_0), (x_1, y_1 y'_1), \ldots, (x_{2n-1}, y_{2n-1} y'_{2n-1})\} .$$

Given two input polynomials in extended point-value form, we see that the time to multiply them to obtain the point-value form of the result is $\Theta(n)$, much less than the time required to multiply polynomials in coefficient form.

Finally, we consider how to evaluate a polynomial given in point-value form at a new point. For this problem, we know of no simpler approach than converting the polynomial to coefficient form first, and then evaluating it at the new point.

Fast multiplication of polynomials in coefficient form

Can we use the linear-time multiplication method for polynomials in point-value form to expedite polynomial multiplication in coefficient form? The answer hinges

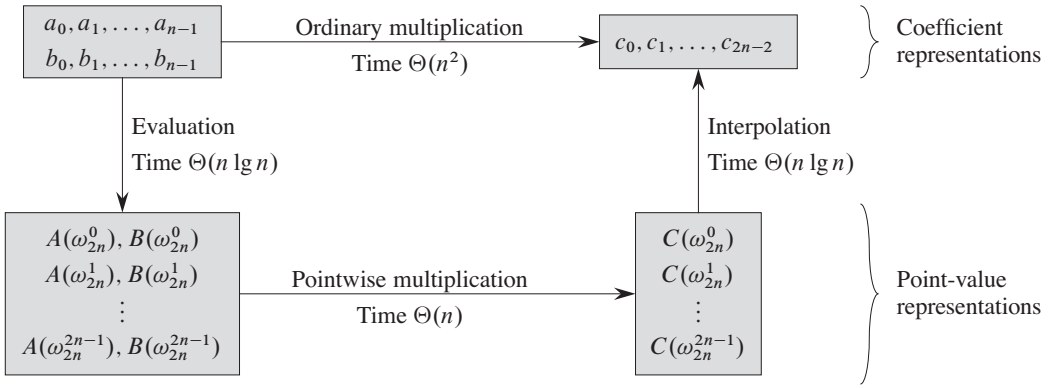


Figure 30.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, while those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots of unity.

on whether we can convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice versa (interpolate).

We can use any points we want as evaluation points, but by choosing the evaluation points carefully, we can convert between representations in only $\Theta(n \lg n)$ time. As we shall see in Section 30.2, if we choose "complex roots of unity" as the evaluation points, we can produce a point-value representation by taking the discrete Fourier transform (or DFT) of a coefficient vector. We can perform the inverse operation, interpolation, by taking the "inverse DFT" of point-value pairs, yielding a coefficient vector. Section 30.2 will show how the FFT accomplishes the DFT and inverse DFT operations in $\Theta(n \lg n)$ time.

Figure 30.1 shows this strategy graphically. One minor detail concerns degree-bounds. The product of two polynomials of degree-bound n is a polynomial of degree-bound $2n$. Before evaluating the input polynomials A and B, therefore, we first double their degree-bounds to $2n$ by adding n high-order coefficients of 0. Because the vectors have $2n$ elements, we use "complex $(2n)$th roots of unity," which are denoted by the ω_{2n} terms in Figure 30.1.

Given the FFT, we have the following $\Theta(n \lg n)$-time procedure for multiplying two polynomials $A(x)$ and $B(x)$ of degree-bound n, where the input and output representations are in coefficient form. We assume that n is a power of 2; we can always meet this requirement by adding high-order zero coefficients.

1. *Double degree-bound:* Create coefficient representations of $A(x)$ and $B(x)$ as degree-bound $2n$ polynomials by adding n high-order zero coefficients to each.

2. *Evaluate:* Compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ by applying the FFT of order $2n$ on each polynomial. These representations contain the values of the two polynomials at the $(2n)$th roots of unity.
3. *Pointwise multiply:* Compute a point-value representation for the polynomial $C(x) = A(x)B(x)$ by multiplying these values together pointwise. This representation contains the value of $C(x)$ at each $(2n)$th root of unity.
4. *Interpolate:* Create the coefficient representation of the polynomial $C(x)$ by applying the FFT on $2n$ point-value pairs to compute the inverse DFT.

Steps (1) and (3) take time $\Theta(n)$, and steps (2) and (4) take time $\Theta(n \lg n)$. Thus, once we show how to use the FFT, we will have proven the following.

Theorem 30.2
We can multiply two polynomials of degree-bound n in time $\Theta(n \lg n)$, with both the input and output representations in coefficient form. ■

Exercises

30.1-1
Multiply the polynomials $A(x) = 7x^3 - x^2 + x - 10$ and $B(x) = 8x^3 - 6x + 3$ using equations (30.1) and (30.2).

30.1-2
Another way to evaluate a polynomial $A(x)$ of degree-bound n at a given point x_0 is to divide $A(x)$ by the polynomial $(x - x_0)$, obtaining a quotient polynomial $q(x)$ of degree-bound $n - 1$ and a remainder r, such that

$$A(x) = q(x)(x - x_0) + r \ .$$

Clearly, $A(x_0) = r$. Show how to compute the remainder r and the coefficients of $q(x)$ in time $\Theta(n)$ from x_0 and the coefficients of A.

30.1-3
Derive a point-value representation for $A^{\mathrm{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$ from a point-value representation for $A(x) = \sum_{j=0}^{n-1} a_j x^j$, assuming that none of the points is 0.

30.1-4
Prove that n distinct point-value pairs are necessary to uniquely specify a polynomial of degree-bound n, that is, if fewer than n distinct point-value pairs are given, they fail to specify a unique polynomial of degree-bound n. (*Hint:* Using Theorem 30.1, what can you say about a set of $n - 1$ point-value pairs to which you add one more arbitrarily chosen point-value pair?)

30.1-5
Show how to use equation (30.5) to interpolate in time $\Theta(n^2)$. (*Hint:* First compute the coefficient representation of the polynomial $\prod_j (x - x_j)$ and then divide by $(x - x_k)$ as necessary for the numerator of each term; see Exercise 30.1-2. You can compute each of the n denominators in time $O(n)$.)

30.1-6
Explain what is wrong with the "obvious" approach to polynomial division using a point-value representation, i.e., dividing the corresponding y values. Discuss separately the case in which the division comes out exactly and the case in which it doesn't.

30.1-7
Consider two sets A and B, each having n integers in the range from 0 to $10n$. We wish to compute the ***Cartesian sum*** of A and B, defined by

$$C = \{x + y : x \in A \text{ and } y \in B\} .$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B. Show how to solve the problem in $O(n \lg n)$ time. (*Hint:* Represent A and B as polynomials of degree at most $10n$.)

30.2 The DFT and FFT

In Section 30.1, we claimed that if we use complex roots of unity, we can evaluate and interpolate polynomials in $\Theta(n \lg n)$ time. In this section, we define complex roots of unity and study their properties, define the DFT, and then show how the FFT computes the DFT and its inverse in $\Theta(n \lg n)$ time.

Complex roots of unity

A ***complex nth root of unity*** is a complex number ω such that

$$\omega^n = 1 .$$

There are exactly n complex nth roots of unity: $e^{2\pi i k/n}$ for $k = 0, 1, \ldots, n-1$. To interpret this formula, we use the definition of the exponential of a complex number:

$$e^{iu} = \cos(u) + i \sin(u) .$$

Figure 30.2 shows that the n complex roots of unity are equally spaced around the circle of unit radius centered at the origin of the complex plane. The value

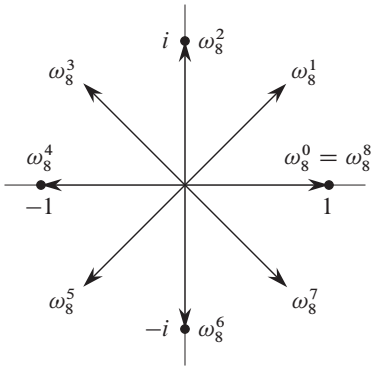


Figure 30.2 The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

is the ***principal nth root of unity***;[2] all other complex nth roots of unity are powers of ω_n.

The n complex nth roots of unity,

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1} ,$$

form a group under multiplication (see Section 31.3). This group has the same structure as the additive group $(\mathbb{Z}_n, +)$ modulo n, since $\omega_n^n = \omega_n^0 = 1$ implies that $\omega_n^j \omega_n^k = \omega_n^{j+k} = \omega_n^{(j+k) \bmod n}$. Similarly, $\omega_n^{-1} = \omega_n^{n-1}$. The following lemmas furnish some essential properties of the complex nth roots of unity.

Lemma 30.3 (Cancellation lemma)
For any integers $n \geq 0$, $k \geq 0$, and $d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k . \tag{30.7}$$

Proof The lemma follows directly from equation (30.6), since

$$\begin{aligned} \omega_{dn}^{dk} &= \left(e^{2\pi i/dn}\right)^{dk} \\ &= \left(e^{2\pi i/n}\right)^{k} \\ &= \omega_n^k . \end{aligned}$$

■

[2]Many other authors define ω_n differently: $\omega_n = e^{-2\pi i/n}$. This alternative definition tends to be used for signal-processing applications. The underlying mathematics is substantially the same with either definition of ω_n.

Corollary 30.4
For any even integer $n > 0$,

$$\omega_n^{n/2} = \omega_2 = -1 .$$

Proof The proof is left as Exercise 30.2-1. ■

Lemma 30.5 (Halving lemma)
If $n > 0$ is even, then the squares of the n complex nth roots of unity are the $n/2$ complex $(n/2)$th roots of unity.

Proof By the cancellation lemma, we have $(\omega_n^k)^2 = \omega_{n/2}^k$, for any nonnegative integer k. Note that if we square all of the complex nth roots of unity, then we obtain each $(n/2)$th root of unity exactly twice, since

$$\begin{aligned}(\omega_n^{k+n/2})^2 &= \omega_n^{2k+n} \\ &= \omega_n^{2k}\omega_n^n \\ &= \omega_n^{2k} \\ &= (\omega_n^k)^2 .\end{aligned}$$

Thus, ω_n^k and $\omega_n^{k+n/2}$ have the same square. We could also have used Corollary 30.4 to prove this property, since $\omega_n^{n/2} = -1$ implies $\omega_n^{k+n/2} = -\omega_n^k$, and thus $(\omega_n^{k+n/2})^2 = (\omega_n^k)^2$. ■

As we shall see, the halving lemma is essential to our divide-and-conquer approach for converting between coefficient and point-value representations of polynomials, since it guarantees that the recursive subproblems are only half as large.

Lemma 30.6 (Summation lemma)
For any integer $n \geq 1$ and nonzero integer k not divisible by n,

$$\sum_{j=0}^{n-1} \left(\omega_n^k\right)^j = 0 .$$

Proof Equation (A.5) applies to complex values as well as to reals, and so we have

$$\begin{aligned}\sum_{j=0}^{n-1}\left(\omega_n^k\right)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1} \\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1} \\ &= \frac{(1)^k - 1}{\omega_n^k - 1} \\ &= 0\,.\end{aligned}$$

Because we require that k is not divisible by n, and because $\omega_n^k = 1$ only when k is divisible by n, we ensure that the denominator is not 0. ■

The DFT

Recall that we wish to evaluate a polynomial

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

of degree-bound n at $\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$ (that is, at the n complex nth roots of unity).[3] We assume that A is given in coefficient form: $a = (a_0, a_1, \ldots, a_{n-1})$. Let us define the results y_k, for $k = 0, 1, \ldots, n-1$, by

$$\begin{aligned}y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj}\,. \end{aligned} \tag{30.8}$$

The vector $y = (y_0, y_1, \ldots, y_{n-1})$ is the ***discrete Fourier transform (DFT)*** of the coefficient vector $a = (a_0, a_1, \ldots, a_{n-1})$. We also write $y = \mathrm{DFT}_n(a)$.

The FFT

By using a method known as the ***fast Fourier transform (FFT)***, which takes advantage of the special properties of the complex roots of unity, we can compute $\mathrm{DFT}_n(a)$ in time $\Theta(n \lg n)$, as opposed to the $\Theta(n^2)$ time of the straightforward method. We assume throughout that n is an exact power of 2. Although strategies

[3] The length n is actually what we referred to as $2n$ in Section 30.1, since we double the degree-bound of the given polynomials prior to evaluation. In the context of polynomial multiplication, therefore, we are actually working with complex $(2n)$th roots of unity.

for dealing with non-power-of-2 sizes are known, they are beyond the scope of this book.

The FFT method employs a divide-and-conquer strategy, using the even-indexed and odd-indexed coefficients of $A(x)$ separately to define the two new polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ of degree-bound $n/2$:

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{[1]}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

Note that $A^{[0]}$ contains all the even-indexed coefficients of A (the binary representation of the index ends in 0) and $A^{[1]}$ contains all the odd-indexed coefficients (the binary representation of the index ends in 1). It follows that

$$A(x) = A^{[0]}(x^2) + x A^{[1]}(x^2) , \tag{30.9}$$

so that the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ reduces to

1. evaluating the degree-bound $n/2$ polynomials $A^{[0]}(x)$ and $A^{[1]}(x)$ at the points

   $$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 , \tag{30.10}$$

   and then
2. combining the results according to equation (30.9).

By the halving lemma, the list of values (30.10) consists not of n distinct values but only of the $n/2$ complex $(n/2)$th roots of unity, with each root occurring exactly twice. Therefore, we recursively evaluate the polynomials $A^{[0]}$ and $A^{[1]}$ of degree-bound $n/2$ at the $n/2$ complex $(n/2)$th roots of unity. These subproblems have exactly the same form as the original problem, but are half the size. We have now successfully divided an n-element DFT_n computation into two $n/2$-element $\mathrm{DFT}_{n/2}$ computations. This decomposition is the basis for the following recursive FFT algorithm, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is a power of 2.

```
RECURSIVE-FFT(a)
 1  n = a.length                    // n is a power of 2
 2  if n == 1
 3      return a
 4  ω_n = e^{2πi/n}
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_{n-2})
 7  a^[1] = (a_1, a_3, ..., a_{n-1})
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11      y_k = y_k^[0] + ω y_k^[1]
12      y_{k+(n/2)} = y_k^[0] - ω y_k^[1]
13      ω = ω ω_n
14  return y                        // y is assumed to be a column vector
```

The RECURSIVE-FFT procedure works as follows. Lines 2–3 represent the basis of the recursion; the DFT of one element is the element itself, since in this case

$$\begin{aligned} y_0 &= a_0 \omega_1^0 \\ &= a_0 \cdot 1 \\ &= a_0 \, . \end{aligned}$$

Lines 6–7 define the coefficient vectors for the polynomials $A^{[0]}$ and $A^{[1]}$. Lines 4, 5, and 13 guarantee that ω is updated properly so that whenever lines 11–12 are executed, we have $\omega = \omega_n^k$. (Keeping a running value of ω from iteration to iteration saves time over computing ω_n^k from scratch each time through the **for** loop.) Lines 8–9 perform the recursive $\mathrm{DFT}_{n/2}$ computations, setting, for $k = 0, 1, \ldots, n/2 - 1$,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_{n/2}^k) \, , \\ y_k^{[1]} &= A^{[1]}(\omega_{n/2}^k) \, , \end{aligned}$$

or, since $\omega_{n/2}^k = \omega_n^{2k}$ by the cancellation lemma,

$$\begin{aligned} y_k^{[0]} &= A^{[0]}(\omega_n^{2k}) \, , \\ y_k^{[1]} &= A^{[1]}(\omega_n^{2k}) \, . \end{aligned}$$

Lines 11–12 combine the results of the recursive $\text{DFT}_{n/2}$ calculations. For $y_0, y_1, \ldots, y_{n/2-1}$, line 11 yields

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \\ &= A(\omega_n^k) \qquad \text{(by equation (30.9))} . \end{aligned}$$

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, letting $k = 0, 1, \ldots, n/2 - 1$, line 12 yields

$$\begin{aligned} y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && \text{(since } \omega_n^{k+(n/2)} = -\omega_n^k\text{)} \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && \text{(since } \omega_n^{2k+n} = \omega_n^{2k}\text{)} \\ &= A(\omega_n^{k+(n/2)}) && \text{(by equation (30.9))} . \end{aligned}$$

Thus, the vector y returned by RECURSIVE-FFT is indeed the DFT of the input vector a.

Lines 11 and 12 multiply each value $y_k^{[1]}$ by ω_n^k, for $k = 0, 1, \ldots, n/2 - 1$. Line 11 adds this product to $y_k^{[0]}$, and line 12 subtracts it. Because we use each factor ω_n^k in both its positive and negative forms, we call the factors ω_n^k ***twiddle factors***.

To determine the running time of procedure RECURSIVE-FFT, we note that exclusive of the recursive calls, each invocation takes time $\Theta(n)$, where n is the length of the input vector. The recurrence for the running time is therefore

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) . \end{aligned}$$

Thus, we can evaluate a polynomial of degree-bound n at the complex nth roots of unity in time $\Theta(n \lg n)$ using the fast Fourier transform.

Interpolation at the complex roots of unity

We now complete the polynomial multiplication scheme by showing how to interpolate the complex roots of unity by a polynomial, which enables us to convert from point-value form back to coefficient form. We interpolate by writing the DFT as a matrix equation and then looking at the form of the matrix inverse.

From equation (30.4), we can write the DFT as the matrix product $y = V_n a$, where V_n is a Vandermonde matrix containing the appropriate powers of ω_n:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} .$$

The (k, j) entry of V_n is ω_n^{kj}, for $j, k = 0, 1, \ldots, n-1$. The exponents of the entries of V_n form a multiplication table.

For the inverse operation, which we write as $a = \mathrm{DFT}_n^{-1}(y)$, we proceed by multiplying y by the matrix V_n^{-1}, the inverse of V_n.

Theorem 30.7

For $j, k = 0, 1, \ldots, n-1$, the (j, k) entry of V_n^{-1} is ω_n^{-kj}/n.

Proof We show that $V_n^{-1} V_n = I_n$, the $n \times n$ identity matrix. Consider the (j, j') entry of $V_n^{-1} V_n$:

$$\begin{aligned} [V_n^{-1} V_n]_{jj'} &= \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'}) \\ &= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n \ . \end{aligned}$$

This summation equals 1 if $j' = j$, and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that we rely on $-(n-1) \le j' - j \le n-1$, so that $j' - j$ is not divisible by n, in order for the summation lemma to apply. ■

Given the inverse matrix V_n^{-1}, we have that $\mathrm{DFT}_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{30.11}$$

for $j = 0, 1, \ldots, n-1$. By comparing equations (30.8) and (30.11), we see that by modifying the FFT algorithm to switch the roles of a and y, replace ω_n by ω_n^{-1}, and divide each element of the result by n, we compute the inverse DFT (see Exercise 30.2-4). Thus, we can compute DFT_n^{-1} in $\Theta(n \lg n)$ time as well.

We see that, by using the FFT and the inverse FFT, we can transform a polynomial of degree-bound n back and forth between its coefficient representation and a point-value representation in time $\Theta(n \lg n)$. In the context of polynomial multiplication, we have shown the following.

Theorem 30.8 (Convolution theorem)
For any two vectors a and b of length n, where n is a power of 2,

$$a \otimes b = \mathrm{DFT}_{2n}^{-1}(\mathrm{DFT}_{2n}(a) \cdot \mathrm{DFT}_{2n}(b)) ,$$

where the vectors a and b are padded with 0s to length $2n$ and $\cdot$ denotes the componentwise product of two $2n$-element vectors. ■

Exercises

30.2-1
Prove Corollary 30.4.

30.2-2
Compute the DFT of the vector $(0, 1, 2, 3)$.

30.2-3
Do Exercise 30.1-1 by using the $\Theta(n \lg n)$-time scheme.

30.2-4
Write pseudocode to compute DFT_n^{-1} in $\Theta(n \lg n)$ time.

30.2-5
Describe the generalization of the FFT procedure to the case in which n is a power of 3. Give a recurrence for the running time, and solve the recurrence.

30.2-6 ★
Suppose that instead of performing an n-element FFT over the field of complex numbers (where n is even), we use the ring $\mathbb{Z}_m$ of integers modulo m, where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. Use $\omega = 2^t$ instead of ω_n as a principal nth root of unity, modulo m. Prove that the DFT and the inverse DFT are well defined in this system.

30.2-7
Given a list of values $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions), show how to find the coefficients of a polynomial $P(x)$ of degree-bound $n + 1$ that has zeros only at $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions). Your procedure should run in time $O(n \lg^2 n)$. (*Hint:* The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

30.2-8 ★
The ***chirp transform*** of a vector $a = (a_0, a_1, \ldots, a_{n-1})$ is the vector $y = (y_0, y_1, \ldots, y_{n-1})$, where $y_k = \sum_{j=0}^{n-1} a_j z^{kj}$ and z is any complex number. The

DFT is therefore a special case of the chirp transform, obtained by taking $z = \omega_n$. Show how to evaluate the chirp transform in time $O(n \lg n)$ for any complex number z. (*Hint:* Use the equation

$$y_k = z^{k^2/2} \sum_{j=0}^{n-1} \left(a_j z^{j^2/2}\right)\left(z^{-(k-j)^2/2}\right)$$

to view the chirp transform as a convolution.)

30.3 Efficient FFT implementations

Since the practical applications of the DFT, such as signal processing, demand the utmost speed, this section examines two efficient FFT implementations. First, we shall examine an iterative version of the FFT algorithm that runs in $\Theta(n \lg n)$ time but can have a lower constant hidden in the Θ-notation than the recursive version in Section 30.2. (Depending on the exact implementation, the recursive version may use the hardware cache more efficiently.) Then, we shall use the insights that led us to the iterative implementation to design an efficient parallel FFT circuit.

An iterative FFT implementation

We first note that the **for** loop of lines 10–13 of RECURSIVE-FFT involves computing the value $\omega_n^k\, y_k^{[1]}$ twice. In compiler terminology, we call such a value a ***common subexpression***. We can change the loop to compute it only once, storing it in a temporary variable t.

```
for k = 0 to n/2 − 1
    t = ω y_k^[1]
    y_k = y_k^[0] + t
    y_{k+(n/2)} = y_k^[0] − t
    ω = ω ω_n
```

The operation in this loop, multiplying the twiddle factor $\omega = \omega_n^k$ by $y_k^{[1]}$, storing the product into t, and adding and subtracting t from $y_k^{[0]}$, is known as a ***butterfly operation*** and is shown schematically in Figure 30.3.

We now show how to make the FFT algorithm iterative rather than recursive in structure. In Figure 30.4, we have arranged the input vectors to the recursive calls in an invocation of RECURSIVE-FFT in a tree structure, where the initial call is for $n = 8$. The tree has one node for each call of the procedure, labeled

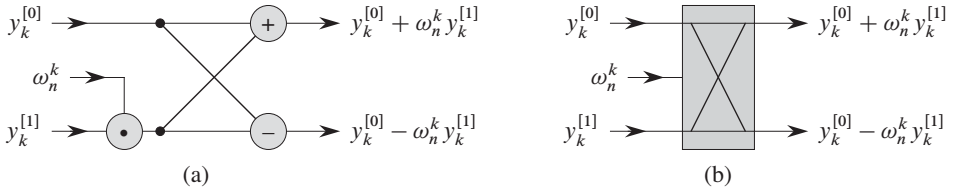


Figure 30.3 A butterfly operation. **(a)** The two input values enter from the left, the twiddle factor ω_n^k is multiplied by $y_k^{[1]}$, and the sum and difference are output on the right. **(b)** A simplified drawing of a butterfly operation. We will use this representation in a parallel FFT circuit.

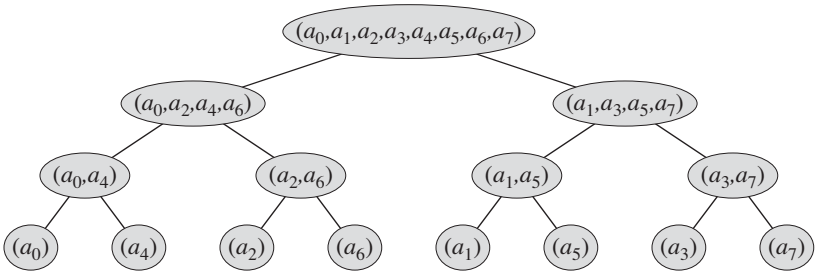


Figure 30.4 The tree of input vectors to the recursive calls of the RECURSIVE-FFT procedure. The initial invocation is for $n = 8$.

by the corresponding input vector. Each RECURSIVE-FFT invocation makes two recursive calls, unless it has received a 1-element vector. The first call appears in the left child, and the second call appears in the right child.

Looking at the tree, we observe that if we could arrange the elements of the initial vector a into the order in which they appear in the leaves, we could trace the execution of the RECURSIVE-FFT procedure, but bottom up instead of top down. First, we take the elements in pairs, compute the DFT of each pair using one butterfly operation, and replace the pair with its DFT. The vector then holds $n/2$ 2-element DFTs. Next, we take these $n/2$ DFTs in pairs and compute the DFT of the four vector elements they come from by executing two butterfly operations, replacing two 2-element DFTs with one 4-element DFT. The vector then holds $n/4$ 4-element DFTs. We continue in this manner until the vector holds two $(n/2)$-element DFTs, which we combine using $n/2$ butterfly operations into the final n-element DFT.

To turn this bottom-up approach into code, we use an array $A[0\,..\,n-1]$ that initially holds the elements of the input vector a in the order in which they appear

in the leaves of the tree of Figure 30.4. (We shall show later how to determine this order, which is known as a bit-reversal permutation.) Because we have to combine DFTs on each level of the tree, we introduce a variable s to count the levels, ranging from 1 (at the bottom, when we are combining pairs to form 2-element DFTs) to $\lg n$ (at the top, when we are combining two $(n/2)$-element DFTs to produce the final result). The algorithm therefore has the following structure:

```
1  for s = 1 to lg n
2      for k = 0 to n − 1 by 2^s
3          combine the two 2^(s−1)-element DFTs in
               A[k . . k + 2^(s−1) − 1] and A[k + 2^(s−1) . . k + 2^s − 1]
               into one 2^s-element DFT in A[k . . k + 2^s − 1]
```

We can express the body of the loop (line 3) as more precise pseudocode. We copy the **for** loop from the RECURSIVE-FFT procedure, identifying $y^{[0]}$ with $A[k \,..\, k + 2^{s-1} - 1]$ and $y^{[1]}$ with $A[k + 2^{s-1} \,..\, k + 2^s - 1]$. The twiddle factor used in each butterfly operation depends on the value of s; it is a power of ω_m, where $m = 2^s$. (We introduce the variable m solely for the sake of readability.) We introduce another temporary variable u that allows us to perform the butterfly operation in place. When we replace line 3 of the overall structure by the loop body, we get the following pseudocode, which forms the basis of the parallel implementation we shall present later. The code first calls the auxiliary procedure BIT-REVERSE-COPY(a, A) to copy vector a into array A in the initial order in which we need the values.

ITERATIVE-FFT(a)

```
 1  BIT-REVERSE-COPY(a, A)
 2  n = a.length                    // n is a power of 2
 3  for s = 1 to lg n
 4      m = 2^s
 5      ω_m = e^(2πi/m)
 6      for k = 0 to n − 1 by m
 7          ω = 1
 8          for j = 0 to m/2 − 1
 9              t = ω A[k + j + m/2]
10              u = A[k + j]
11              A[k + j] = u + t
12              A[k + j + m/2] = u − t
13              ω = ω ω_m
14  return A
```

How does BIT-REVERSE-COPY get the elements of the input vector a into the desired order in the array A? The order in which the leaves appear in Figure 30.4

is a ***bit-reversal permutation***. That is, if we let $\text{rev}(k)$ be the $\lg n$-bit integer formed by reversing the bits of the binary representation of k, then we want to place vector element a_k in array position $A[\text{rev}(k)]$. In Figure 30.4, for example, the leaves appear in the order $0, 4, 2, 6, 1, 5, 3, 7$; this sequence in binary is $000, 100, 010, 110, 001, 101, 011, 111$, and when we reverse the bits of each value we get the sequence $000, 001, 010, 011, 100, 101, 110, 111$. To see that we want a bit-reversal permutation in general, we note that at the top level of the tree, indices whose low-order bit is 0 go into the left subtree and indices whose low-order bit is 1 go into the right subtree. Stripping off the low-order bit at each level, we continue this process down the tree, until we get the order given by the bit-reversal permutation at the leaves.

Since we can easily compute the function $\text{rev}(k)$, the BIT-REVERSE-COPY procedure is simple:

```
BIT-REVERSE-COPY(a, A)
1  n = a.length
2  for k = 0 to n − 1
3      A[rev(k)] = a_k
```

The iterative FFT implementation runs in time $\Theta(n \lg n)$. The call to BIT-REVERSE-COPY(a, A) certainly runs in $O(n \lg n)$ time, since we iterate n times and can reverse an integer between 0 and $n - 1$, with $\lg n$ bits, in $O(\lg n)$ time. (In practice, because we usually know the initial value of n in advance, we would probably code a table mapping k to $\text{rev}(k)$, making BIT-REVERSE-COPY run in $\Theta(n)$ time with a low hidden constant. Alternatively, we could use the clever amortized reverse binary counter scheme described in Problem 17-1.) To complete the proof that ITERATIVE-FFT runs in time $\Theta(n \lg n)$, we show that $L(n)$, the number of times the body of the innermost loop (lines 8–13) executes, is $\Theta(n \lg n)$. The **for** loop of lines 6–13 iterates $n/m = n/2^s$ times for each value of s, and the innermost loop of lines 8–13 iterates $m/2 = 2^{s-1}$ times. Thus,

$$\begin{aligned} L(n) &= \sum_{s=1}^{\lg n} \frac{n}{2^s} \cdot 2^{s-1} \\ &= \sum_{s=1}^{\lg n} \frac{n}{2} \\ &= \Theta(n \lg n) \, . \end{aligned}$$

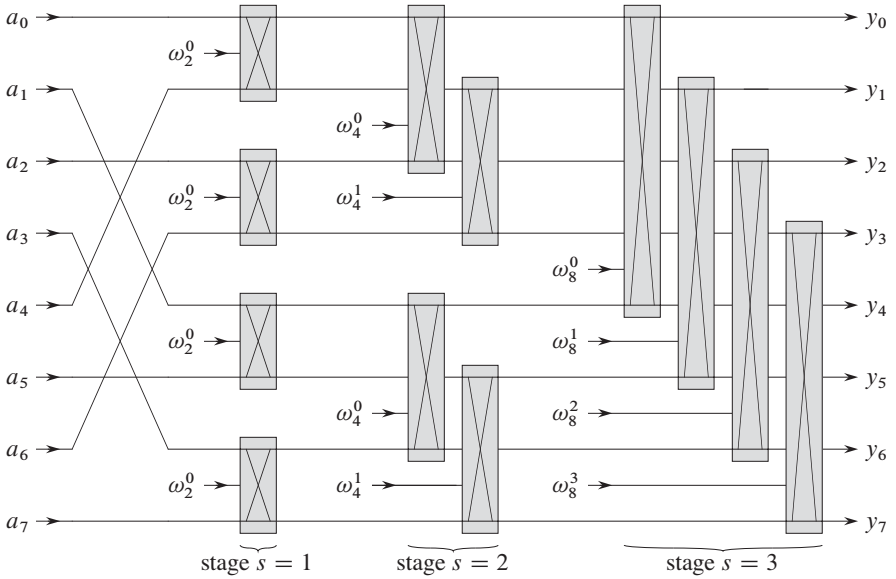


Figure 30.5 A circuit that computes the FFT in parallel, here shown on $n = 8$ inputs. Each butterfly operation takes as input the values on two wires, along with a twiddle factor, and it produces as outputs the values on two wires. The stages of butterflies are labeled to correspond to iterations of the outermost loop of the ITERATIVE-FFT procedure. Only the top and bottom wires passing through a butterfly interact with it; wires that pass through the middle of a butterfly do not affect that butterfly, nor are their values changed by that butterfly. For example, the top butterfly in stage 2 has nothing to do with wire 1 (the wire whose output is labeled y_1); its inputs and outputs are only on wires 0 and 2 (labeled y_0 and y_2, respectively). This circuit has depth $\Theta(\lg n)$ and performs $\Theta(n \lg n)$ butterfly operations altogether.

A parallel FFT circuit

We can exploit many of the properties that allowed us to implement an efficient iterative FFT algorithm to produce an efficient parallel algorithm for the FFT. We will express the parallel FFT algorithm as a circuit. Figure 30.5 shows a parallel FFT circuit, which computes the FFT on n inputs, for $n = 8$. The circuit begins with a bit-reverse permutation of the inputs, followed by $\lg n$ stages, each stage consisting of $n/2$ butterflies executed in parallel. The ***depth*** of the circuit—the maximum number of computational elements between any output and any input that can reach it—is therefore $\Theta(\lg n)$.

The leftmost part of the parallel FFT circuit performs the bit-reverse permutation, and the remainder mimics the iterative ITERATIVE-FFT procedure. Because each iteration of the outermost **for** loop performs $n/2$ independent butterfly operations, the circuit performs them in parallel. The value of s in each iteration within

ITERATIVE-FFT corresponds to a stage of butterflies shown in Figure 30.5. For $s = 1, 2, \ldots, \lg n$, stage s consists of $n/2^s$ groups of butterflies (corresponding to each value of k in ITERATIVE-FFT), with 2^{s-1} butterflies per group (corresponding to each value of j in ITERATIVE-FFT). The butterflies shown in Figure 30.5 correspond to the butterfly operations of the innermost loop (lines 9–12 of ITERATIVE-FFT). Note also that the twiddle factors used in the butterflies correspond to those used in ITERATIVE-FFT: in stage s, we use $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, where $m = 2^s$.

Exercises

30.3-1
Show how ITERATIVE-FFT computes the DFT of the input vector $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3-2
Show how to implement an FFT algorithm with the bit-reversal permutation occurring at the end, rather than at the beginning, of the computation. (*Hint:* Consider the inverse DFT.)

30.3-3
How many times does ITERATIVE-FFT compute twiddle factors in each stage? Rewrite ITERATIVE-FFT to compute twiddle factors only 2^{s-1} times in stage s.

30.3-4 ⋆
Suppose that the adders within the butterfly operations of the FFT circuit sometimes fail in such a manner that they always produce a zero output, independent of their inputs. Suppose that exactly one adder has failed, but that you don't know which one. Describe how you can identify the failed adder by supplying inputs to the overall FFT circuit and observing the outputs. How efficient is your method?

Problems

30-1 Divide-and-conquer multiplication

a. Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (*Hint:* One of the multiplications is $(a + b) \cdot (c + d)$.)

b. Give two divide-and-conquer algorithms for multiplying two polynomials of degree-bound n in $\Theta(n^{\lg 3})$ time. The first algorithm should divide the input polynomial coefficients into a high half and a low half, and the second algorithm should divide them according to whether their index is odd or even.

c. Show how to multiply two n-bit integers in $O(n^{\lg 3})$ steps, where each step operates on at most a constant number of 1-bit values.

30-2 Toeplitz matrices

A ***Toeplitz matrix*** is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$.

a. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

b. Describe how to represent a Toeplitz matrix so that you can add two $n \times n$ Toeplitz matrices in $O(n)$ time.

c. Give an $O(n \lg n)$-time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b).

d. Give an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

30-3 Multidimensional fast Fourier transform

We can generalize the 1-dimensional discrete Fourier transform defined by equation (30.8) to d dimensions. The input is a d-dimensional array $A = (a_{j_1, j_2, \ldots, j_d})$ whose dimensions are $n_1, n_2, \ldots, n_d$, where $n_1 n_2 \cdots n_d = n$. We define the d-dimensional discrete Fourier transform by the equation

$$y_{k_1,k_2,\ldots,k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,\ldots,j_d} \omega_{n_1}^{j_1 k_1} \omega_{n_2}^{j_2 k_2} \cdots \omega_{n_d}^{j_d k_d}$$

for $0 \le k_1 < n_1, 0 \le k_2 < n_2, \ldots, 0 \le k_d < n_d$.

a. Show that we can compute a d-dimensional DFT by computing 1-dimensional DFTs on each dimension in turn. That is, we first compute n/n_1 separate 1-dimensional DFTs along dimension 1. Then, using the result of the DFTs along dimension 1 as the input, we compute n/n_2 separate 1-dimensional DFTs along dimension 2. Using this result as the input, we compute n/n_3 separate 1-dimensional DFTs along dimension 3, and so on, through dimension d.

b. Show that the ordering of dimensions does not matter, so that we can compute a d-dimensional DFT by computing the 1-dimensional DFTs in any order of the d dimensions.

c. Show that if we compute each 1-dimensional DFT by computing the fast Fourier transform, the total time to compute a d-dimensional DFT is $O(n \lg n)$, independent of d.

30-4 Evaluating all derivatives of a polynomial at a point

Given a polynomial $A(x)$ of degree-bound n, we define its tth derivative by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0 , \\ \frac{d}{dx} A^{(t-1)}(x) & \text{if } 1 \le t \le n-1 , \\ 0 & \text{if } t \ge n . \end{cases}$$

From the coefficient representation $(a_0, a_1, \ldots, a_{n-1})$ of $A(x)$ and a given point x_0, we wish to determine $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n-1$.

a. Given coefficients $b_0, b_1, \ldots, b_{n-1}$ such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j ,$$

show how to compute $A^{(t)}(x_0)$, for $t = 0, 1, \ldots, n-1$, in $O(n)$ time.

b. Explain how to find $b_0, b_1, \ldots, b_{n-1}$ in $O(n \lg n)$ time, given $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$.

c. Prove that

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right) ,$$

where $f(j) = a_j \cdot j!$ and

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{if } -(n-1) \le l \le 0 , \\ 0 & \text{if } 1 \le l \le n-1 . \end{cases}$$

d. Explain how to evaluate $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$ in $O(n \lg n)$ time. Conclude that we can evaluate all nontrivial derivatives of $A(x)$ at x_0 in $O(n \lg n)$ time.

30-5 Polynomial evaluation at multiple points

We have seen how to evaluate a polynomial of degree-bound n at a single point in $O(n)$ time using Horner's rule. We have also discovered how to evaluate such a polynomial at all n complex roots of unity in $O(n \lg n)$ time using the FFT. We shall now show how to evaluate a polynomial of degree-bound n at n arbitrary points in $O(n \lg^2 n)$ time.

To do so, we shall assume that we can compute the polynomial remainder when one such polynomial is divided by another in $O(n \lg n)$ time, a result that we state without proof. For example, the remainder of $3x^3 + x^2 - 3x + 1$ when divided by $x^2 + x + 2$ is

$$(3x^3 + x^2 - 3x + 1) \bmod (x^2 + x + 2) = -7x + 5 .$$

Given the coefficient representation of a polynomial $A(x) = \sum_{k=0}^{n-1} a_k x^k$ and n points $x_0, x_1, \ldots, x_{n-1}$, we wish to compute the n values $A(x_0), A(x_1), \ldots, A(x_{n-1})$. For $0 \le i \le j \le n-1$, define the polynomials $P_{ij}(x) = \prod_{k=i}^{j} (x - x_k)$ and $Q_{ij}(x) = A(x) \bmod P_{ij}(x)$. Note that $Q_{ij}(x)$ has degree at most $j - i$.

a. Prove that $A(x) \bmod (x - z) = A(z)$ for any point z.

b. Prove that $Q_{kk}(x) = A(x_k)$ and that $Q_{0,n-1}(x) = A(x)$.

c. Prove that for $i \le k \le j$, we have $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ and $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Give an $O(n \lg^2 n)$-time algorithm to evaluate $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

30-6 FFT using modular arithmetic

As defined, the discrete Fourier transform requires us to compute with complex numbers, which can result in a loss of precision due to round-off errors. For some problems, the answer is known to contain only integers, and by using a variant of the FFT based on modular arithmetic, we can guarantee that the answer is calculated exactly. An example of such a problem is that of multiplying two polynomials with integer coefficients. Exercise 30.2-6 gives one approach, using a modulus of length $\Omega(n)$ bits to handle a DFT on n points. This problem gives another approach, which uses a modulus of the more reasonable length $O(\lg n)$; it requires that you understand the material of Chapter 31. Let n be a power of 2.

a. Suppose that we search for the smallest k such that $p = kn + 1$ is prime. Give a simple heuristic argument why we might expect k to be approximately $\ln n$. (The value of k might be much larger or smaller, but we can reasonably expect to examine $O(\lg n)$ candidate values of k on average.) How does the expected length of p compare to the length of n?

Let g be a generator of $\mathbb{Z}_p^*$, and let $w = g^k \bmod p$.

b. Argue that the DFT and the inverse DFT are well-defined inverse operations modulo p, where w is used as a principal nth root of unity.

c. Show how to make the FFT and its inverse work modulo p in time $O(n \lg n)$, where operations on words of $O(\lg n)$ bits take unit time. Assume that the algorithm is given p and w.

d. Compute the DFT modulo $p = 17$ of the vector $(0, 5, 3, 7, 7, 2, 1, 6)$. Note that $g = 3$ is a generator of $\mathbb{Z}_{17}^*$.

Chapter notes

Van Loan's book [343] provides an outstanding treatment of the fast Fourier transform. Press, Teukolsky, Vetterling, and Flannery [283, 284] have a good description of the fast Fourier transform and its applications. For an excellent introduction to signal processing, a popular FFT application area, see the texts by Oppenheim and Schafer [266] and Oppenheim and Willsky [267]. The Oppenheim and Schafer book also shows how to handle cases in which n is not an integer power of 2.

Fourier analysis is not limited to 1-dimensional data. It is widely used in image processing to analyze data in 2 or more dimensions. The books by Gonzalez and Woods [146] and Pratt [281] discuss multidimensional Fourier transforms and their use in image processing, and books by Tolimieri, An, and Lu [338] and Van Loan [343] discuss the mathematics of multidimensional fast Fourier transforms.

Cooley and Tukey [76] are widely credited with devising the FFT in the 1960s. The FFT had in fact been discovered many times previously, but its importance was not fully realized before the advent of modern digital computers. Although Press, Teukolsky, Vetterling, and Flannery attribute the origins of the method to Runge and König in 1924, an article by Heideman, Johnson, and Burrus [163] traces the history of the FFT as far back as C. F. Gauss in 1805.

Frigo and Johnson [117] developed a fast and flexible implementation of the FFT, called FFTW ("fastest Fourier transform in the West"). FFTW is designed for situations requiring multiple DFT computations on the same problem size. Before actually computing the DFTs, FFTW executes a "planner," which, by a series of trial runs, determines how best to decompose the FFT computation for the given problem size on the host machine. FFTW adapts to use the hardware cache efficiently, and once subproblems are small enough, FFTW solves them with optimized, straight-line code. Furthermore, FFTW has the unusual advantage of taking $\Theta(n \lg n)$ time for any problem size n, even when n is a large prime.

Although the standard Fourier transform assumes that the input represents points that are uniformly spaced in the time domain, other techniques can approximate the FFT on "nonequispaced" data. The article by Ware [348] provides an overview.

31 Number-Theoretic Algorithms

Number theory was once viewed as a beautiful but largely useless subject in pure mathematics. Today number-theoretic algorithms are used widely, due in large part to the invention of cryptographic schemes based on large prime numbers. These schemes are feasible because we can find large primes easily, and they are secure because we do not know how to factor the product of large primes (or solve related problems, such as computing discrete logarithms) efficiently. This chapter presents some of the number theory and related algorithms that underlie such applications.

Section 31.1 introduces basic concepts of number theory, such as divisibility, modular equivalence, and unique factorization. Section 31.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers. Section 31.3 reviews concepts of modular arithmetic. Section 31.4 then studies the set of multiples of a given number a, modulo n, and shows how to find all solutions to the equation $ax \equiv b \pmod{n}$ by using Euclid's algorithm. The Chinese remainder theorem is presented in Section 31.5. Section 31.6 considers powers of a given number a, modulo n, and presents a repeated-squaring algorithm for efficiently computing $a^b \bmod n$, given a, b, and n. This operation is at the heart of efficient primality testing and of much modern cryptography. Section 31.7 then describes the RSA public-key cryptosystem. Section 31.8 examines a randomized primality test. We can use this test to find large primes efficiently, which we need to do in order to create keys for the RSA cryptosystem. Finally, Section 31.9 reviews a simple but effective heuristic for factoring small integers. It is a curious fact that factoring is one problem people may wish to be intractable, since the security of RSA depends on the difficulty of factoring large integers.

Size of inputs and cost of arithmetic computations

Because we shall be working with large integers, we need to adjust how we think about the size of an input and about the cost of elementary arithmetic operations.

In this chapter, a "large input" typically means an input containing "large integers" rather than an input containing "many integers" (as for sorting). Thus,

we shall measure the size of an input in terms of the *number of bits* required to represent that input, not just the number of integers in the input. An algorithm with integer inputs $a_1, a_2, \ldots, a_k$ is a ***polynomial-time algorithm*** if it runs in time polynomial in $\lg a_1, \lg a_2, \ldots, \lg a_k$, that is, polynomial in the lengths of its binary-encoded inputs.

In most of this book, we have found it convenient to think of the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. By counting the number of such arithmetic operations that an algorithm performs, we have a basis for making a reasonable estimate of the algorithm's actual running time on a computer. Elementary operations can be time-consuming, however, when their inputs are large. It thus becomes convenient to measure how many ***bit operations*** a number-theoretic algorithm requires. In this model, multiplying two β-bit integers by the ordinary method uses $\Theta(\beta^2)$ bit operations. Similarly, we can divide a β-bit integer by a shorter integer or take the remainder of a β-bit integer when divided by a shorter integer in time $\Theta(\beta^2)$ by simple algorithms. (See Exercise 31.1-12.) Faster methods are known. For example, a simple divide-and-conquer method for multiplying two β-bit integers has a running time of $\Theta(\beta^{\lg 3})$, and the fastest known method has a running time of $\Theta(\beta \lg \beta \lg\lg \beta)$. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we shall use this bound as a basis for our analyses.

We shall generally analyze algorithms in this chapter in terms of both the number of arithmetic operations and the number of bit operations they require.

31.1 Elementary number-theoretic notions

This section provides a brief review of notions from elementary number theory concerning the set $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ of integers and the set $\mathbb{N} = \{0, 1, 2, \ldots\}$ of natural numbers.

Divisibility and divisors

The notion of one integer being divisible by another is key to the theory of numbers. The notation $d \mid a$ (read "d ***divides*** a") means that $a = kd$ for some integer k. Every integer divides 0. If $a > 0$ and $d \mid a$, then $|d| \le |a|$. If $d \mid a$, then we also say that a is a ***multiple*** of d. If d does not divide a, we write $d \nmid a$.

If $d \mid a$ and $d \ge 0$, we say that d is a ***divisor*** of a. Note that $d \mid a$ if and only if $-d \mid a$, so that no generality is lost by defining the divisors to be nonnegative, with the understanding that the negative of any divisor of a also divides a. A

divisor of a nonzero integer a is at least 1 but not greater than $|a|$. For example, the divisors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

Every positive integer a is divisible by the ***trivial divisors*** 1 and a. The nontrivial divisors of a are the ***factors*** of a. For example, the factors of 20 are 2, 4, 5, and 10.

Prime and composite numbers

An integer $a > 1$ whose only divisors are the trivial divisors 1 and a is a ***prime number*** or, more simply, a ***prime***. Primes have many special properties and play a critical role in number theory. The first 20 primes, in order, are

$$2,\ 3,\ 5,\ 7,\ 11,\ 13,\ 17,\ 19,\ 23,\ 29,\ 31,\ 37,\ 41,\ 43,\ 47,\ 53,\ 59,\ 61,\ 67,\ 71\ .$$

Exercise 31.1-2 asks you to prove that there are infinitely many primes. An integer $a > 1$ that is not prime is a ***composite number*** or, more simply, a ***composite***. For example, 39 is composite because $3 \mid 39$. We call the integer 1 a ***unit***, and it is neither prime nor composite. Similarly, the integer 0 and all negative integers are neither prime nor composite.

The division theorem, remainders, and modular equivalence

Given an integer n, we can partition the integers into those that are multiples of n and those that are not multiples of n. Much number theory is based upon refining this partition by classifying the nonmultiples of n according to their remainders when divided by n. The following theorem provides the basis for this refinement. We omit the proof (but see, for example, Niven and Zuckerman [265]).

Theorem 31.1 (Division theorem)
For any integer a and any positive integer n, there exist unique integers q and r such that $0 \le r < n$ and $a = qn + r$. ■

The value $q = \lfloor a/n \rfloor$ is the ***quotient*** of the division. The value $r = a \bmod n$ is the ***remainder*** (or ***residue***) of the division. We have that $n \mid a$ if and only if $a \bmod n = 0$.

We can partition the integers into n equivalence classes according to their remainders modulo n. The ***equivalence class modulo n*** containing an integer a is

$$[a]_n = \{a + kn : k \in \mathbb{Z}\}\ .$$

For example, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$; we can also denote this set by $[-4]_7$ and $[10]_7$. Using the notation defined on page 54, we can say that writing $a \in [b]_n$ is the same as writing $a \equiv b \pmod{n}$. The set of all such equivalence classes is

$$\mathbb{Z}_n = \{[a]_n : 0 \le a \le n-1\} \ . \tag{31.1}$$

When you see the definition

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\} \ , \tag{31.2}$$

you should read it as equivalent to equation (31.1) with the understanding that 0 represents $[0]_n$, 1 represents $[1]_n$, and so on; each class is represented by its smallest nonnegative element. You should keep the underlying equivalence classes in mind, however. For example, if we refer to -1 as a member of $\mathbb{Z}_n$, we are really referring to $[n-1]_n$, since $-1 \equiv n-1 \pmod{n}$.

Common divisors and greatest common divisors

If d is a divisor of a and d is also a divisor of b, then d is a ***common divisor*** of a and b. For example, the divisors of 30 are 1, 2, 3, 5, 6, 10, 15, and 30, and so the common divisors of 24 and 30 are 1, 2, 3, and 6. Note that 1 is a common divisor of any two integers.

An important property of common divisors is that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (a+b) \text{ and } d \mid (a-b) \ . \tag{31.3}$$

More generally, we have that

$$d \mid a \text{ and } d \mid b \text{ implies } d \mid (ax + by) \tag{31.4}$$

for any integers x and y. Also, if $a \mid b$, then either $|a| \le |b|$ or $b = 0$, which implies that

$$a \mid b \text{ and } b \mid a \text{ implies } a = \pm b \ . \tag{31.5}$$

The ***greatest common divisor*** of two integers a and b, not both zero, is the largest of the common divisors of a and b; we denote it by $\gcd(a, b)$. For example, $\gcd(24, 30) = 6$, $\gcd(5, 7) = 1$, and $\gcd(0, 9) = 9$. If a and b are both nonzero, then $\gcd(a, b)$ is an integer between 1 and $\min(|a|, |b|)$. We define $\gcd(0, 0)$ to be 0; this definition is necessary to make standard properties of the gcd function (such as equation (31.9) below) universally valid.

The following are elementary properties of the gcd function:

$$\gcd(a, b) = \gcd(b, a) \ , \tag{31.6}$$

$$\gcd(a, b) = \gcd(-a, b) \ , \tag{31.7}$$

$$\gcd(a, b) = \gcd(|a|, |b|) \ , \tag{31.8}$$

$$\gcd(a, 0) = |a| \ , \tag{31.9}$$

$$\gcd(a, ka) = |a| \qquad \text{for any } k \in \mathbb{Z} \ . \tag{31.10}$$

The following theorem provides an alternative and useful characterization of $\gcd(a, b)$.

Theorem 31.2
If a and b are any integers, not both zero, then $\gcd(a, b)$ is the smallest positive element of the set $\{ax + by : x, y \in \mathbb{Z}\}$ of linear combinations of a and b.

Proof Let s be the smallest positive such linear combination of a and b, and let $s = ax + by$ for some $x, y \in \mathbb{Z}$. Let $q = \lfloor a/s \rfloor$. Equation (3.8) then implies

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a\,(1 - qx) + b\,(-qy)\ , \end{aligned}$$

and so $a \bmod s$ is a linear combination of a and b as well. But, since $0 \le a \bmod s < s$, we have that $a \bmod s = 0$, because s is the smallest positive such linear combination. Therefore, we have that $s \mid a$ and, by analogous reasoning, $s \mid b$. Thus, s is a common divisor of a and b, and so $\gcd(a, b) \ge s$. Equation (31.4) implies that $\gcd(a, b) \mid s$, since $\gcd(a, b)$ divides both a and b and s is a linear combination of a and b. But $\gcd(a, b) \mid s$ and $s > 0$ imply that $\gcd(a, b) \le s$. Combining $\gcd(a, b) \ge s$ and $\gcd(a, b) \le s$ yields $\gcd(a, b) = s$. We conclude that s is the greatest common divisor of a and b. ■

Corollary 31.3
For any integers a and b, if $d \mid a$ and $d \mid b$, then $d \mid \gcd(a, b)$.

Proof This corollary follows from equation (31.4), because $\gcd(a, b)$ is a linear combination of a and b by Theorem 31.2. ■

Corollary 31.4
For all integers a and b and any nonnegative integer n,

$$\gcd(an, bn) = n \gcd(a, b)\ .$$

Proof If $n = 0$, the corollary is trivial. If $n > 0$, then $\gcd(an, bn)$ is the smallest positive element of the set $\{anx + bny : x, y \in \mathbb{Z}\}$, which is n times the smallest positive element of the set $\{ax + by : x, y \in \mathbb{Z}\}$. ■

Corollary 31.5
For all positive integers n, a, and b, if $n \mid ab$ and $\gcd(a, n) = 1$, then $n \mid b$.

Proof We leave the proof as Exercise 31.1-5. ■

Relatively prime integers

Two integers a and b are ***relatively prime*** if their only common divisor is 1, that is, if $\gcd(a, b) = 1$. For example, 8 and 15 are relatively prime, since the divisors of 8 are 1, 2, 4, and 8, and the divisors of 15 are 1, 3, 5, and 15. The following theorem states that if two integers are each relatively prime to an integer p, then their product is relatively prime to p.

Theorem 31.6
For any integers a, b, and p, if both $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then $\gcd(ab, p) = 1$.

Proof It follows from Theorem 31.2 that there exist integers x, y, x', and y' such that

$$\begin{aligned} ax + py &= 1 , \\ bx' + py' &= 1 . \end{aligned}$$

Multiplying these equations and rearranging, we have

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 .$$

Since 1 is thus a positive linear combination of ab and p, an appeal to Theorem 31.2 completes the proof. ■

Integers $n_1, n_2, \ldots, n_k$ are ***pairwise relatively prime*** if, whenever $i \neq j$, we have $\gcd(n_i, n_j) = 1$.

Unique factorization

An elementary but important fact about divisibility by primes is the following.

Theorem 31.7
For all primes p and all integers a and b, if $p \mid ab$, then $p \mid a$ or $p \mid b$ (or both).

Proof Assume for the purpose of contradiction that $p \mid ab$, but that $p \nmid a$ and $p \nmid b$. Thus, $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, since the only divisors of p are 1 and p, and we assume that p divides neither a nor b. Theorem 31.6 then implies that $\gcd(ab, p) = 1$, contradicting our assumption that $p \mid ab$, since $p \mid ab$ implies $\gcd(ab, p) = p$. This contradiction completes the proof. ■

A consequence of Theorem 31.7 is that we can uniquely factor any composite integer into a product of primes.

Theorem 31.8 (Unique factorization)
There is exactly one way to write any composite integer a as a product of the form

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} ,$$

where the p_i are prime, $p_1 < p_2 < \cdots < p_r$, and the e_i are positive integers.

Proof We leave the proof as Exercise 31.1-11. ■

As an example, the number 6000 is uniquely factored into primes as $2^4 \cdot 3 \cdot 5^3$.

Exercises

31.1-1
Prove that if $a > b > 0$ and $c = a + b$, then $c \bmod a = b$.

31.1-2
Prove that there are infinitely many primes. (*Hint:* Show that none of the primes $p_1, p_2, \ldots, p_k$ divide $(p_1 p_2 \cdots p_k) + 1$.)

31.1-3
Prove that if $a \mid b$ and $b \mid c$, then $a \mid c$.

31.1-4
Prove that if p is prime and $0 < k < p$, then $\gcd(k, p) = 1$.

31.1-5
Prove Corollary 31.5.

31.1-6
Prove that if p is prime and $0 < k < p$, then $p \mid \binom{p}{k}$. Conclude that for all integers a and b and all primes p,

$$(a + b)^p \equiv a^p + b^p \pmod{p} .$$

31.1-7
Prove that if a and b are any positive integers such that $a \mid b$, then

$$(x \bmod b) \bmod a = x \bmod a$$

for any x. Prove, under the same assumptions, that

$$x \equiv y \pmod{b} \text{ implies } x \equiv y \pmod{a}$$

for any integers x and y.

31.1-8
For any integer $k > 0$, an integer n is a ***kth power*** if there exists an integer a such that $a^k = n$. Furthermore, $n > 1$ is a ***nontrivial power*** if it is a kth power for some integer $k > 1$. Show how to determine whether a given β-bit integer n is a nontrivial power in time polynomial in β.

31.1-9
Prove equations (31.6)–(31.10).

31.1-10
Show that the gcd operator is associative. That is, prove that for all integers a, b, and c,

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) .$$

31.1-11 ⋆
Prove Theorem 31.8.

31.1-12
Give efficient algorithms for the operations of dividing a β-bit integer by a shorter integer and of taking the remainder of a β-bit integer when divided by a shorter integer. Your algorithms should run in time $\Theta(\beta^2)$.

31.1-13
Give an efficient algorithm to convert a given β-bit (binary) integer to a decimal representation. Argue that if multiplication or division of integers whose length is at most β takes time $M(\beta)$, then we can convert binary to decimal in time $\Theta(M(\beta) \lg \beta)$. (*Hint:* Use a divide-and-conquer approach, obtaining the top and bottom halves of the result with separate recursions.)

31.2 Greatest common divisor

In this section, we describe Euclid's algorithm for efficiently computing the greatest common divisor of two integers. When we analyze the running time, we shall see a surprising connection with the Fibonacci numbers, which yield a worst-case input for Euclid's algorithm.

We restrict ourselves in this section to nonnegative integers. This restriction is justified by equation (31.8), which states that $\gcd(a, b) = \gcd(|a|, |b|)$.

In principle, we can compute $\gcd(a, b)$ for positive integers a and b from the prime factorizations of a and b. Indeed, if

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.11}$$

$$b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r} , \tag{31.12}$$

with zero exponents being used to make the set of primes $p_1, p_2, \ldots, p_r$ the same for both a and b, then, as Exercise 31.2-1 asks you to show,

$$\gcd(a, b) = p_1^{\min(e_1, f_1)} p_2^{\min(e_2, f_2)} \cdots p_r^{\min(e_r, f_r)} . \tag{31.13}$$

As we shall show in Section 31.9, however, the best algorithms to date for factoring do not run in polynomial time. Thus, this approach to computing greatest common divisors seems unlikely to yield an efficient algorithm.

Euclid's algorithm for computing greatest common divisors relies on the following theorem.

Theorem 31.9 (GCD recursion theorem)
For any nonnegative integer a and any positive integer b,

$$\gcd(a, b) = \gcd(b, a \bmod b) .$$

Proof We shall show that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ divide each other, so that by equation (31.5) they must be equal (since they are both nonnegative).

We first show that $\gcd(a, b) \mid \gcd(b, a \bmod b)$. If we let $d = \gcd(a, b)$, then $d \mid a$ and $d \mid b$. By equation (3.8), $a \bmod b = a - qb$, where $q = \lfloor a/b \rfloor$. Since $a \bmod b$ is thus a linear combination of a and b, equation (31.4) implies that $d \mid (a \bmod b)$. Therefore, since $d \mid b$ and $d \mid (a \bmod b)$, Corollary 31.3 implies that $d \mid \gcd(b, a \bmod b)$ or, equivalently, that

$$\gcd(a, b) \mid \gcd(b, a \bmod b). \tag{31.14}$$

Showing that $\gcd(b, a \bmod b) \mid \gcd(a, b)$ is almost the same. If we now let $d = \gcd(b, a \bmod b)$, then $d \mid b$ and $d \mid (a \bmod b)$. Since $a = qb + (a \bmod b)$, where $q = \lfloor a/b \rfloor$, we have that a is a linear combination of b and $(a \bmod b)$. By equation (31.4), we conclude that $d \mid a$. Since $d \mid b$ and $d \mid a$, we have that $d \mid \gcd(a, b)$ by Corollary 31.3 or, equivalently, that

$$\gcd(b, a \bmod b) \mid \gcd(a, b). \tag{31.15}$$

Using equation (31.5) to combine equations (31.14) and (31.15) completes the proof. ■

Euclid's algorithm

The *Elements* of Euclid (circa 300 B.C.) describes the following gcd algorithm, although it may be of even earlier origin. We express Euclid's algorithm as a recursive program based directly on Theorem 31.9. The inputs a and b are arbitrary nonnegative integers.

```
EUCLID(a, b)
1  if b == 0
2        return a
3  else return EUCLID(b, a mod b)
```

As an example of the running of EUCLID, consider the computation of gcd(30, 21):

$$\begin{aligned} \text{EUCLID}(30,21) &= \text{EUCLID}(21,9) \\ &= \text{EUCLID}(9,3) \\ &= \text{EUCLID}(3,0) \\ &= 3 \,. \end{aligned}$$

This computation calls EUCLID recursively three times.

The correctness of EUCLID follows from Theorem 31.9 and the property that if the algorithm returns a in line 2, then $b = 0$, so that equation (31.9) implies that $\gcd(a,b) = \gcd(a,0) = a$. The algorithm cannot recurse indefinitely, since the second argument strictly decreases in each recursive call and is always nonnegative. Therefore, EUCLID always terminates with the correct answer.

The running time of Euclid's algorithm

We analyze the worst-case running time of EUCLID as a function of the size of a and b. We assume with no loss of generality that $a > b \geq 0$. To justify this assumption, observe that if $b > a \geq 0$, then $\text{EUCLID}(a,b)$ immediately makes the recursive call $\text{EUCLID}(b,a)$. That is, if the first argument is less than the second argument, EUCLID spends one recursive call swapping its arguments and then proceeds. Similarly, if $b = a > 0$, the procedure terminates after one recursive call, since $a \bmod b = 0$.

The overall running time of EUCLID is proportional to the number of recursive calls it makes. Our analysis makes use of the Fibonacci numbers F_k, defined by the recurrence (3.22).

Lemma 31.10

If $a > b \geq 1$ and the call $\text{EUCLID}(a,b)$ performs $k \geq 1$ recursive calls, then $a \geq F_{k+2}$ and $b \geq F_{k+1}$.

Proof The proof proceeds by induction on k. For the basis of the induction, let $k = 1$. Then, $b \geq 1 = F_2$, and since $a > b$, we must have $a \geq 2 = F_3$. Since $b > (a \bmod b)$, in each recursive call the first argument is strictly larger than the second; the assumption that $a > b$ therefore holds for each recursive call.

Assume inductively that the lemma holds if $k - 1$ recursive calls are made; we shall then prove that the lemma holds for k recursive calls. Since $k > 0$, we have $b > 0$, and EUCLID(a, b) calls EUCLID$(b, a \bmod b)$ recursively, which in turn makes $k - 1$ recursive calls. The inductive hypothesis then implies that $b \geq F_{k+1}$ (thus proving part of the lemma), and $a \bmod b \geq F_k$. We have

$$\begin{aligned} b + (a \bmod b) &= b + (a - b \lfloor a/b \rfloor) \\ &\leq a , \end{aligned}$$

since $a > b > 0$ implies $\lfloor a/b \rfloor \geq 1$. Thus,

$$\begin{aligned} a &\geq b + (a \bmod b) \\ &\geq F_{k+1} + F_k \\ &= F_{k+2} . \end{aligned}$$

■

The following theorem is an immediate corollary of this lemma.

Theorem 31.11 (Lamé's theorem)
For any integer $k \geq 1$, if $a > b \geq 1$ and $b < F_{k+1}$, then the call EUCLID(a, b) makes fewer than k recursive calls. ■

We can show that the upper bound of Theorem 31.11 is the best possible by showing that the call EUCLID(F_{k+1}, F_k) makes exactly $k - 1$ recursive calls when $k \geq 2$. We use induction on k. For the base case, $k = 2$, and the call EUCLID(F_3, F_2) makes exactly one recursive call, to EUCLID$(1, 0)$. (We have to start at $k = 2$, because when $k = 1$ we do not have $F_2 > F_1$.) For the inductive step, assume that EUCLID(F_k, F_{k-1}) makes exactly $k - 2$ recursive calls. For $k > 2$, we have $F_k > F_{k-1} > 0$ and $F_{k+1} = F_k + F_{k-1}$, and so by Exercise 31.1-1, we have $F_{k+1} \bmod F_k = F_{k-1}$. Thus, we have

$$\begin{aligned} \gcd(F_{k+1}, F_k) &= \gcd(F_k, F_{k+1} \bmod F_k) \\ &= \gcd(F_k, F_{k-1}) . \end{aligned}$$

Therefore, the call EUCLID(F_{k+1}, F_k) recurses one time more than the call EUCLID(F_k, F_{k-1}), or exactly $k - 1$ times, meeting the upper bound of Theorem 31.11.

Since F_k is approximately $\phi^k / \sqrt{5}$, where ϕ is the golden ratio $(1 + \sqrt{5})/2$ defined by equation (3.24), the number of recursive calls in EUCLID is $O(\lg b)$. (See

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	−11	14
78	21	3	3	3	−11
21	15	1	3	−2	3
15	6	2	3	1	−2
6	3	2	3	0	1
3	0	—	3	1	0

Figure 31.1 How EXTENDED-EUCLID computes gcd(99, 78). Each line shows one level of the recursion: the values of the inputs a and b, the computed value $\lfloor a/b \rfloor$, and the values d, x, and y returned. The triple (d, x, y) returned becomes the triple (d', x', y') used at the next higher level of recursion. The call EXTENDED-EUCLID(99, 78) returns $(3, -11, 14)$, so that $\gcd(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

Exercise 31.2-5 for a tighter bound.) Therefore, if we call EUCLID on two β-bit numbers, then it performs $O(\beta)$ arithmetic operations and $O(\beta^3)$ bit operations (assuming that multiplication and division of β-bit numbers take $O(\beta^2)$ bit operations). Problem 31-2 asks you to show an $O(\beta^2)$ bound on the number of bit operations.

The extended form of Euclid's algorithm

We now rewrite Euclid's algorithm to compute additional useful information. Specifically, we extend the algorithm to compute the integer coefficients x and y such that

$$d = \gcd(a, b) = ax + by \ . \tag{31.16}$$

Note that x and y may be zero or negative. We shall find these coefficients useful later for computing modular multiplicative inverses. The procedure EXTENDED-EUCLID takes as input a pair of nonnegative integers and returns a triple of the form (d, x, y) that satisfies equation (31.16).

```
EXTENDED-EUCLID(a, b)
1  if b == 0
2      return (a, 1, 0)
3  else (d', x', y') = EXTENDED-EUCLID(b, a mod b)
4      (d, x, y) = (d', y', x' − ⌊a/b⌋ y')
5      return (d, x, y)
```

Figure 31.1 illustrates how EXTENDED-EUCLID computes gcd(99, 78).

The EXTENDED-EUCLID procedure is a variation of the EUCLID procedure. Line 1 is equivalent to the test "b == 0" in line 1 of EUCLID. If $b = 0$, then

EXTENDED-EUCLID returns not only $d = a$ in line 2, but also the coefficients $x = 1$ and $y = 0$, so that $a = ax + by$. If $b \neq 0$, EXTENDED-EUCLID first computes (d', x', y') such that $d' = \gcd(b, a \bmod b)$ and

$$d' = bx' + (a \bmod b)y' . \tag{31.17}$$

As for EUCLID, we have in this case $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. To obtain x and y such that $d = ax + by$, we start by rewriting equation (31.17) using the equation $d = d'$ and equation (3.8):

$$\begin{aligned} d &= bx' + (a - b\lfloor a/b \rfloor)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') . \end{aligned}$$

Thus, choosing $x = y'$ and $y = x' - \lfloor a/b \rfloor y'$ satisfies the equation $d = ax + by$, proving the correctness of EXTENDED-EUCLID.

Since the number of recursive calls made in EUCLID is equal to the number of recursive calls made in EXTENDED-EUCLID, the running times of EUCLID and EXTENDED-EUCLID are the same, to within a constant factor. That is, for $a > b > 0$, the number of recursive calls is $O(\lg b)$.

Exercises

31.2-1
Prove that equations (31.11) and (31.12) imply equation (31.13).

31.2-2
Compute the values (d, x, y) that the call EXTENDED-EUCLID$(899, 493)$ returns.

31.2-3
Prove that for all integers a, k, and n,

$$\gcd(a, n) = \gcd(a + kn, n) .$$

31.2-4
Rewrite EUCLID in an iterative form that uses only a constant amount of memory (that is, stores only a constant number of integer values).

31.2-5
If $a > b \geq 0$, show that the call EUCLID(a, b) makes at most $1 + \log_\phi b$ recursive calls. Improve this bound to $1 + \log_\phi(b/\gcd(a, b))$.

31.2-6
What does EXTENDED-EUCLID(F_{k+1}, F_k) return? Prove your answer correct.

31.2-7
Define the gcd function for more than two arguments by the recursive equation $\gcd(a_0, a_1, \ldots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \ldots, a_n))$. Show that the gcd function returns the same answer independent of the order in which its arguments are specified. Also show how to find integers $x_0, x_1, \ldots, x_n$ such that $\gcd(a_0, a_1, \ldots, a_n) = a_0x_0 + a_1x_1 + \cdots + a_nx_n$. Show that the number of divisions performed by your algorithm is $O(n + \lg(\max\{a_0, a_1, \ldots, a_n\}))$.

31.2-8
Define $\operatorname{lcm}(a_1, a_2, \ldots, a_n)$ to be the ***least common multiple*** of the n integers $a_1, a_2, \ldots, a_n$, that is, the smallest nonnegative integer that is a multiple of each a_i. Show how to compute $\operatorname{lcm}(a_1, a_2, \ldots, a_n)$ efficiently using the (two-argument) gcd operation as a subroutine.

31.2-9
Prove that n_1, n_2, n_3, and n_4 are pairwise relatively prime if and only if

$$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1 .$$

More generally, show that $n_1, n_2, \ldots, n_k$ are pairwise relatively prime if and only if a set of $\lceil \lg k \rceil$ pairs of numbers derived from the n_i are relatively prime.

31.3 Modular arithmetic

Informally, we can think of modular arithmetic as arithmetic as usual over the integers, except that if we are working modulo n, then every result x is replaced by the element of $\{0, 1, \ldots, n-1\}$ that is equivalent to x, modulo n (that is, x is replaced by $x \bmod n$). This informal model suffices if we stick to the operations of addition, subtraction, and multiplication. A more formal model for modular arithmetic, which we now give, is best described within the framework of group theory.

Finite groups

A ***group*** $(S, \oplus)$ is a set S together with a binary operation $\oplus$ defined on S for which the following properties hold:

1. **Closure:** For all $a, b \in S$, we have $a \oplus b \in S$.
2. **Identity:** There exists an element $e \in S$, called the ***identity*** of the group, such that $e \oplus a = a \oplus e = a$ for all $a \in S$.
3. **Associativity:** For all $a, b, c \in S$, we have $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.

4. **Inverses:** For each $a \in S$, there exists a unique element $b \in S$, called the ***inverse*** of a, such that $a \oplus b = b \oplus a = e$.

As an example, consider the familiar group $(\mathbb{Z}, +)$ of the integers $\mathbb{Z}$ under the operation of addition: 0 is the identity, and the inverse of a is $-a$. If a group $(S, \oplus)$ satisfies the ***commutative law*** $a \oplus b = b \oplus a$ for all $a, b \in S$, then it is an ***abelian group***. If a group $(S, \oplus)$ satisfies $|S| < \infty$, then it is a ***finite group***.

The groups defined by modular addition and multiplication

We can form two finite abelian groups by using addition and multiplication modulo n, where n is a positive integer. These groups are based on the equivalence classes of the integers modulo n, defined in Section 31.1.

To define a group on $\mathbb{Z}_n$, we need to have suitable binary operations, which we obtain by redefining the ordinary operations of addition and multiplication. We can easily define addition and multiplication operations for $\mathbb{Z}_n$, because the equivalence class of two integers uniquely determines the equivalence class of their sum or product. That is, if $a \equiv a' \pmod{n}$ and $b \equiv b' \pmod{n}$, then

$$\begin{aligned} a + b &\equiv a' + b' \pmod{n}, \\ ab &\equiv a'b' \pmod{n}. \end{aligned}$$

Thus, we define addition and multiplication modulo n, denoted $+_n$ and $\cdot_n$, by

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n, \\ [a]_n \cdot_n [b]_n &= [ab]_n. \end{aligned} \tag{31.18}$$

(We can define subtraction similarly on $\mathbb{Z}_n$ by $[a]_n -_n [b]_n = [a - b]_n$, but division is more complicated, as we shall see.) These facts justify the common and convenient practice of using the smallest nonnegative element of each equivalence class as its representative when performing computations in $\mathbb{Z}_n$. We add, subtract, and multiply as usual on the representatives, but we replace each result x by the representative of its class, that is, by $x \bmod n$.

Using this definition of addition modulo n, we define the ***additive group modulo n*** as $(\mathbb{Z}_n, +_n)$. The size of the additive group modulo n is $|\mathbb{Z}_n| = n$. Figure 31.2(a) gives the operation table for the group $(\mathbb{Z}_6, +_6)$.

Theorem 31.12
The system $(\mathbb{Z}_n, +_n)$ is a finite abelian group.

Proof Equation (31.18) shows that $(\mathbb{Z}_n, +_n)$ is closed. Associativity and commutativity of $+_n$ follow from the associativity and commutativity of $+$:

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

(a)

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

(b)

Figure 31.2 Two finite groups. Equivalence classes are denoted by their representative elements. **(a)** The group $(\mathbb{Z}_6, +_6)$. **(b)** The group $(\mathbb{Z}_{15}^*, \cdot_{15})$.

$$\begin{aligned}
([a]_n +_n [b]_n) +_n [c]_n &= [a+b]_n +_n [c]_n \\
&= [(a+b)+c]_n \\
&= [a+(b+c)]_n \\
&= [a]_n +_n [b+c]_n \\
&= [a]_n +_n ([b]_n +_n [c]_n) ,
\end{aligned}$$

$$\begin{aligned}
[a]_n +_n [b]_n &= [a+b]_n \\
&= [b+a]_n \\
&= [b]_n +_n [a]_n .
\end{aligned}$$

The identity element of $(\mathbb{Z}_n, +_n)$ is 0 (that is, $[0]_n$). The (additive) inverse of an element a (that is, of $[a]_n$) is the element $-a$ (that is, $[-a]_n$ or $[n-a]_n$), since $[a]_n +_n [-a]_n = [a-a]_n = [0]_n$. ■

Using the definition of multiplication modulo n, we define the ***multiplicative group modulo n*** as $(\mathbb{Z}_n^*, \cdot_n)$. The elements of this group are the set $\mathbb{Z}_n^*$ of elements in $\mathbb{Z}_n$ that are relatively prime to n, so that each one has a unique inverse, modulo n:

$$\mathbb{Z}_n^* = \{[a]_n \in \mathbb{Z}_n : \gcd(a, n) = 1\} .$$

To see that $\mathbb{Z}_n^*$ is well defined, note that for $0 \le a < n$, we have $a \equiv (a + kn) \pmod{n}$ for all integers k. By Exercise 31.2-3, therefore, $\gcd(a, n) = 1$ implies $\gcd(a + kn, n) = 1$ for all integers k. Since $[a]_n = \{a + kn : k \in \mathbb{Z}\}$, the set $\mathbb{Z}_n^*$ is well defined. An example of such a group is

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\} ,$$

where the group operation is multiplication modulo 15. (Here we denote an element $[a]_{15}$ as a; for example, we denote $[7]_{15}$ as 7.) Figure 31.2(b) shows the group $(\mathbb{Z}_{15}^*, \cdot_{15})$. For example, $8 \cdot 11 \equiv 13 \pmod{15}$, working in $\mathbb{Z}_{15}^*$. The identity for this group is 1.

Theorem 31.13
The system $(\mathbb{Z}_n^*, \cdot_n)$ is a finite abelian group.

Proof Theorem 31.6 implies that $(\mathbb{Z}_n^*, \cdot_n)$ is closed. Associativity and commutativity can be proved for $\cdot_n$ as they were for $+_n$ in the proof of Theorem 31.12. The identity element is $[1]_n$. To show the existence of inverses, let a be an element of $\mathbb{Z}_n^*$ and let (d, x, y) be returned by EXTENDED-EUCLID(a, n). Then, $d = 1$, since $a \in \mathbb{Z}_n^*$, and

$$ax + ny = 1 \tag{31.19}$$

or, equivalently,

$$ax \equiv 1 \pmod{n} .$$

Thus, $[x]_n$ is a multiplicative inverse of $[a]_n$, modulo n. Furthermore, we claim that $[x]_n \in \mathbb{Z}_n^*$. To see why, equation (31.19) demonstrates that the smallest positive linear combination of x and n must be 1. Therefore, Theorem 31.2 implies that $\gcd(x, n) = 1$. We defer the proof that inverses are uniquely defined until Corollary 31.26. ∎

As an example of computing multiplicative inverses, suppose that $a = 5$ and $n = 11$. Then EXTENDED-EUCLID(a, n) returns $(d, x, y) = (1, -2, 1)$, so that $1 = 5 \cdot (-2) + 11 \cdot 1$. Thus, $[-2]_{11}$ (i.e., $[9]_{11}$) is the multiplicative inverse of $[5]_{11}$.

When working with the groups $(\mathbb{Z}_n, +_n)$ and $(\mathbb{Z}_n^*, \cdot_n)$ in the remainder of this chapter, we follow the convenient practice of denoting equivalence classes by their representative elements and denoting the operations $+_n$ and $\cdot_n$ by the usual arithmetic notations $+$ and $\cdot$ (or juxtaposition, so that $ab = a \cdot b$) respectively. Also, equivalences modulo n may also be interpreted as equations in $\mathbb{Z}_n$. For example, the following two statements are equivalent:

$$\begin{aligned} ax &\equiv b \pmod{n} , \\ [a]_n \cdot_n [x]_n &= [b]_n . \end{aligned}$$

As a further convenience, we sometimes refer to a group $(S, \oplus)$ merely as S when the operation $\oplus$ is understood from context. We may thus refer to the groups $(\mathbb{Z}_n, +_n)$ and $(\mathbb{Z}_n^*, \cdot_n)$ as $\mathbb{Z}_n$ and $\mathbb{Z}_n^*$, respectively.

We denote the (multiplicative) inverse of an element a by $(a^{-1} \bmod n)$. Division in $\mathbb{Z}_n^*$ is defined by the equation $a/b \equiv ab^{-1} \pmod{n}$. For example, in $\mathbb{Z}_{15}^*$

we have that $7^{-1} \equiv 13 \pmod{15}$, since $7 \cdot 13 = 91 \equiv 1 \pmod{15}$, so that $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.

The size of $\mathbb{Z}_n^*$ is denoted $\phi(n)$. This function, known as ***Euler's phi function***, satisfies the equation

$$\phi(n) = n \prod_{p \,:\, p \text{ is prime and } p \mid n} \left(1 - \frac{1}{p}\right) , \tag{31.20}$$

so that p runs over all the primes dividing n (including n itself, if n is prime). We shall not prove this formula here. Intuitively, we begin with a list of the n remainders $\{0, 1, \ldots, n-1\}$ and then, for each prime p that divides n, cross out every multiple of p in the list. For example, since the prime divisors of 45 are 3 and 5,

$$\begin{aligned} \phi(45) &= 45\left(1 - \frac{1}{3}\right)\left(1 - \frac{1}{5}\right) \\ &= 45\left(\frac{2}{3}\right)\left(\frac{4}{5}\right) \\ &= 24 . \end{aligned}$$

If p is prime, then $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$, and

$$\begin{aligned} \phi(p) &= p\left(1 - \frac{1}{p}\right) \\ &= p - 1 . \end{aligned} \tag{31.21}$$

If n is composite, then $\phi(n) < n - 1$, although it can be shown that

$$\phi(n) > \frac{n}{e^{\gamma} \ln\ln n + \frac{3}{\ln\ln n}} \tag{31.22}$$

for $n \geq 3$, where $\gamma = 0.5772156649\ldots$ is ***Euler's constant***. A somewhat simpler (but looser) lower bound for $n > 5$ is

$$\phi(n) > \frac{n}{6 \ln\ln n} . \tag{31.23}$$

The lower bound (31.22) is essentially the best possible, since

$$\liminf_{n \to \infty} \frac{\phi(n)}{n / \ln\ln n} = e^{-\gamma} . \tag{31.24}$$

Subgroups

If $(S, \oplus)$ is a group, $S' \subseteq S$, and $(S', \oplus)$ is also a group, then $(S', \oplus)$ is a ***subgroup*** of $(S, \oplus)$. For example, the even integers form a subgroup of the integers under the operation of addition. The following theorem provides a useful tool for recognizing subgroups.

Theorem 31.14 (A nonempty closed subset of a finite group is a subgroup)
If $(S, \oplus)$ is a finite group and S' is any nonempty subset of S such that $a \oplus b \in S'$ for all $a, b \in S'$, then $(S', \oplus)$ is a subgroup of $(S, \oplus)$.

Proof We leave the proof as Exercise 31.3-3. ■

For example, the set $\{0, 2, 4, 6\}$ forms a subgroup of $\mathbb{Z}_8$, since it is nonempty and closed under the operation $+$ (that is, it is closed under $+_8$).

The following theorem provides an extremely useful constraint on the size of a subgroup; we omit the proof.

Theorem 31.15 (Lagrange's theorem)
If $(S, \oplus)$ is a finite group and $(S', \oplus)$ is a subgroup of $(S, \oplus)$, then $|S'|$ is a divisor of $|S|$. ■

A subgroup S' of a group S is a ***proper*** subgroup if $S' \neq S$. We shall use the following corollary in our analysis in Section 31.8 of the Miller-Rabin primality test procedure.

Corollary 31.16
If S' is a proper subgroup of a finite group S, then $|S'| \leq |S|/2$. ■

Subgroups generated by an element

Theorem 31.14 gives us an easy way to produce a subgroup of a finite group $(S, \oplus)$: choose an element a and take all elements that can be generated from a using the group operation. Specifically, define $a^{(k)}$ for $k \geq 1$ by

$$a^{(k)} = \bigoplus_{i=1}^{k} a = \underbrace{a \oplus a \oplus \cdots \oplus a}_{k} .$$

For example, if we take $a = 2$ in the group $\mathbb{Z}_6$, the sequence $a^{(1)}, a^{(2)}, a^{(3)}, \ldots$ is

$2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots$.

In the group $\mathbb{Z}_n$, we have $a^{(k)} = ka \bmod n$, and in the group $\mathbb{Z}_n^*$, we have $a^{(k)} = a^k \bmod n$. We define the ***subgroup generated by a***, denoted $\langle a \rangle$ or $(\langle a \rangle, \oplus)$, by

$$\langle a \rangle = \{a^{(k)} : k \geq 1\} .$$

We say that a ***generates*** the subgroup $\langle a \rangle$ or that a is a ***generator*** of $\langle a \rangle$. Since S is finite, $\langle a \rangle$ is a finite subset of S, possibly including all of S. Since the associativity of $\oplus$ implies

$$a^{(i)} \oplus a^{(j)} = a^{(i+j)} ,$$

$\langle a \rangle$ is closed and therefore, by Theorem 31.14, $\langle a \rangle$ is a subgroup of S. For example, in $\mathbb{Z}_6$, we have

$$\begin{aligned} \langle 0 \rangle &= \{0\} , \\ \langle 1 \rangle &= \{0, 1, 2, 3, 4, 5\} , \\ \langle 2 \rangle &= \{0, 2, 4\} . \end{aligned}$$

Similarly, in $\mathbb{Z}_7^*$, we have

$$\begin{aligned} \langle 1 \rangle &= \{1\} , \\ \langle 2 \rangle &= \{1, 2, 4\} , \\ \langle 3 \rangle &= \{1, 2, 3, 4, 5, 6\} . \end{aligned}$$

The ***order*** of a (in the group S), denoted $\text{ord}(a)$, is defined as the smallest positive integer t such that $a^{(t)} = e$.

Theorem 31.17
For any finite group $(S, \oplus)$ and any $a \in S$, the order of a is equal to the size of the subgroup it generates, or $\text{ord}(a) = |\langle a \rangle|$.

Proof Let $t = \text{ord}(a)$. Since $a^{(t)} = e$ and $a^{(t+k)} = a^{(t)} \oplus a^{(k)} = a^{(k)}$ for $k \geq 1$, if $i > t$, then $a^{(i)} = a^{(j)}$ for some $j < i$. Thus, as we generate elements by a, we see no new elements after $a^{(t)}$. Thus, $\langle a \rangle = \{a^{(1)}, a^{(2)}, \ldots, a^{(t)}\}$, and so $|\langle a \rangle| \leq t$. To show that $|\langle a \rangle| \geq t$, we show that each element of the sequence $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ is distinct. Suppose for the purpose of contradiction that $a^{(i)} = a^{(j)}$ for some i and j satisfying $1 \leq i < j \leq t$. Then, $a^{(i+k)} = a^{(j+k)}$ for $k \geq 0$. But this equality implies that $a^{(i+(t-j))} = a^{(j+(t-j))} = e$, a contradiction, since $i + (t - j) < t$ but t is the least positive value such that $a^{(t)} = e$. Therefore, each element of the sequence $a^{(1)}, a^{(2)}, \ldots, a^{(t)}$ is distinct, and $|\langle a \rangle| \geq t$. We conclude that $\text{ord}(a) = |\langle a \rangle|$. ■

Corollary 31.18
The sequence $a^{(1)}, a^{(2)}, \ldots$ is periodic with period $t = \text{ord}(a)$; that is, $a^{(i)} = a^{(j)}$ if and only if $i \equiv j \pmod{t}$. ■

Consistent with the above corollary, we define $a^{(0)}$ as e and $a^{(i)}$ as $a^{(i \bmod t)}$, where $t = \text{ord}(a)$, for all integers i.

Corollary 31.19
If $(S, \oplus)$ is a finite group with identity e, then for all $a \in S$,

$$a^{(|S|)} = e .$$

Proof Lagrange's theorem (Theorem 31.15) implies that $\mathrm{ord}(a) \mid |S|$, and so $|S| \equiv 0 \pmod{t}$, where $t = \mathrm{ord}(a)$. Therefore, $a^{(|S|)} = a^{(0)} = e$. ■

Exercises

31.3-1
Draw the group operation tables for the groups $(\mathbb{Z}_4, +_4)$ and $(\mathbb{Z}_5^*, \cdot_5)$. Show that these groups are isomorphic by exhibiting a one-to-one correspondence α between their elements such that $a + b \equiv c \pmod{4}$ if and only if $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

31.3-2
List all subgroups of $\mathbb{Z}_9$ and of $\mathbb{Z}_{13}^*$.

31.3-3
Prove Theorem 31.14.

31.3-4
Show that if p is prime and e is a positive integer, then

$$\phi(p^e) = p^{e-1}(p-1) .$$

31.3-5
Show that for any integer $n > 1$ and for any $a \in \mathbb{Z}_n^*$, the function $f_a : \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ defined by $f_a(x) = ax \bmod n$ is a permutation of $\mathbb{Z}_n^*$.

31.4 Solving modular linear equations

We now consider the problem of finding solutions to the equation

$$ax \equiv b \pmod{n} , \tag{31.25}$$

where $a > 0$ and $n > 0$. This problem has several applications; for example, we shall use it as part of the procedure for finding keys in the RSA public-key cryptosystem in Section 31.7. We assume that a, b, and n are given, and we wish to find all values of x, modulo n, that satisfy equation (31.25). The equation may have zero, one, or more than one such solution.

Let $\langle a \rangle$ denote the subgroup of $\mathbb{Z}_n$ generated by a. Since $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, equation (31.25) has a solution if and only if $[b] \in \langle a \rangle$. Lagrange's theorem (Theorem 31.15) tells us that $|\langle a \rangle|$ must be a divisor of n. The following theorem gives us a precise characterization of $\langle a \rangle$.

Theorem 31.20
For any positive integers a and n, if $d = \gcd(a, n)$, then

$$\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\} \tag{31.26}$$

in $\mathbb{Z}_n$, and thus

$$|\langle a \rangle| = n/d \ .$$

Proof We begin by showing that $d \in \langle a \rangle$. Recall that EXTENDED-EUCLID(a, n) produces integers x' and y' such that $ax' + ny' = d$. Thus, $ax' \equiv d \pmod{n}$, so that $d \in \langle a \rangle$. In other words, d is a multiple of a in $\mathbb{Z}_n$.

Since $d \in \langle a \rangle$, it follows that every multiple of d belongs to $\langle a \rangle$, because any multiple of a multiple of a is itself a multiple of a. Thus, $\langle a \rangle$ contains every element in $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$. That is, $\langle d \rangle \subseteq \langle a \rangle$.

We now show that $\langle a \rangle \subseteq \langle d \rangle$. If $m \in \langle a \rangle$, then $m = ax \bmod n$ for some integer x, and so $m = ax + ny$ for some integer y. However, $d \mid a$ and $d \mid n$, and so $d \mid m$ by equation (31.4). Therefore, $m \in \langle d \rangle$.

Combining these results, we have that $\langle a \rangle = \langle d \rangle$. To see that $|\langle a \rangle| = n/d$, observe that there are exactly n/d multiples of d between 0 and $n-1$, inclusive. ■

Corollary 31.21
The equation $ax \equiv b \pmod{n}$ is solvable for the unknown x if and only if $d \mid b$, where $d = \gcd(a, n)$.

Proof The equation $ax \equiv b \pmod{n}$ is solvable if and only if $[b] \in \langle a \rangle$, which is the same as saying

$$(b \bmod n) \in \{0, d, 2d, \ldots, ((n/d) - 1)d\} \ ,$$

by Theorem 31.20. If $0 \le b < n$, then $b \in \langle a \rangle$ if and only if $d \mid b$, since the members of $\langle a \rangle$ are precisely the multiples of d. If $b < 0$ or $b \ge n$, the corollary then follows from the observation that $d \mid b$ if and only if $d \mid (b \bmod n)$, since b and $b \bmod n$ differ by a multiple of n, which is itself a multiple of d. ■

Corollary 31.22
The equation $ax \equiv b \pmod{n}$ either has d distinct solutions modulo n, where $d = \gcd(a, n)$, or it has no solutions.

Proof If $ax \equiv b \pmod{n}$ has a solution, then $b \in \langle a \rangle$. By Theorem 31.17, $\mathrm{ord}(a) = |\langle a \rangle|$, and so Corollary 31.18 and Theorem 31.20 imply that the sequence $ai \bmod n$, for $i = 0, 1, \ldots$, is periodic with period $|\langle a \rangle| = n/d$. If $b \in \langle a \rangle$, then b appears exactly d times in the sequence $ai \bmod n$, for $i = 0, 1, \ldots, n-1$, since

the length-(n/d) block of values $\langle a \rangle$ repeats exactly d times as i increases from 0 to $n-1$. The indices x of the d positions for which $ax \bmod n = b$ are the solutions of the equation $ax \equiv b \pmod{n}$. ■

Theorem 31.23
Let $d = \gcd(a, n)$, and suppose that $d = ax' + ny'$ for some integers x' and y' (for example, as computed by EXTENDED-EUCLID). If $d \mid b$, then the equation $ax \equiv b \pmod{n}$ has as one of its solutions the value x_0, where

$$x_0 = x'(b/d) \bmod n \ .$$

Proof We have

$$\begin{aligned} ax_0 &\equiv ax'(b/d) \pmod{n} \\ &\equiv d(b/d) \pmod{n} \qquad \text{(because } ax' \equiv d \pmod{n}\text{)} \\ &\equiv b \pmod{n} \ , \end{aligned}$$

and thus x_0 is a solution to $ax \equiv b \pmod{n}$. ■

Theorem 31.24
Suppose that the equation $ax \equiv b \pmod{n}$ is solvable (that is, $d \mid b$, where $d = \gcd(a, n)$) and that x_0 is any solution to this equation. Then, this equation has exactly d distinct solutions, modulo n, given by $x_i = x_0 + i(n/d)$ for $i = 0, 1, \ldots, d-1$.

Proof Because $n/d > 0$ and $0 \le i(n/d) < n$ for $i = 0, 1, \ldots, d-1$, the values $x_0, x_1, \ldots, x_{d-1}$ are all distinct, modulo n. Since x_0 is a solution of $ax \equiv b \pmod{n}$, we have $ax_0 \bmod n \equiv b \pmod{n}$. Thus, for $i = 0, 1, \ldots, d-1$, we have

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \quad \text{(because } d \mid a \text{ implies that } ain/d \text{ is a multiple of } n\text{)} \\ &\equiv b \pmod{n} \ , \end{aligned}$$

and hence $ax_i \equiv b \pmod{n}$, making x_i a solution, too. By Corollary 31.22, the equation $ax \equiv b \pmod{n}$ has exactly d solutions, so that $x_0, x_1, \ldots, x_{d-1}$ must be all of them. ■

We have now developed the mathematics needed to solve the equation $ax \equiv b \pmod{n}$; the following algorithm prints all solutions to this equation. The inputs a and n are arbitrary positive integers, and b is an arbitrary integer.

```
MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)
1  (d, x', y') = EXTENDED-EUCLID(a, n)
2  if d | b
3       x_0 = x'(b/d) mod n
4       for i = 0 to d − 1
5            print (x_0 + i(n/d)) mod n
6  else print "no solutions"
```

As an example of the operation of this procedure, consider the equation $14x \equiv 30 \pmod{100}$ (here, $a = 14$, $b = 30$, and $n = 100$). Calling EXTENDED-EUCLID in line 1, we obtain $(d, x', y') = (2, -7, 1)$. Since $2 \mid 30$, lines 3–5 execute. Line 3 computes $x_0 = (-7)(15) \bmod 100 = 95$. The loop on lines 4–5 prints the two solutions 95 and 45.

The procedure MODULAR-LINEAR-EQUATION-SOLVER works as follows. Line 1 computes $d = \gcd(a, n)$, along with two values x' and y' such that $d = ax' + ny'$, demonstrating that x' is a solution to the equation $ax' \equiv d \pmod{n}$. If d does not divide b, then the equation $ax \equiv b \pmod{n}$ has no solution, by Corollary 31.21. Line 2 checks to see whether $d \mid b$; if not, line 6 reports that there are no solutions. Otherwise, line 3 computes a solution x_0 to $ax \equiv b \pmod{n}$, in accordance with Theorem 31.23. Given one solution, Theorem 31.24 states that adding multiples of (n/d), modulo n, yields the other $d - 1$ solutions. The **for** loop of lines 4–5 prints out all d solutions, beginning with x_0 and spaced n/d apart, modulo n.

MODULAR-LINEAR-EQUATION-SOLVER performs $O(\lg n + \gcd(a, n))$ arithmetic operations, since EXTENDED-EUCLID performs $O(\lg n)$ arithmetic operations, and each iteration of the **for** loop of lines 4–5 performs a constant number of arithmetic operations.

The following corollaries of Theorem 31.24 give specializations of particular interest.

Corollary 31.25

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv b \pmod{n}$ has a unique solution, modulo n. ■

If $b = 1$, a common case of considerable interest, the x we are looking for is a ***multiplicative inverse*** of a, modulo n.

Corollary 31.26

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax \equiv 1 \pmod{n}$ has a unique solution, modulo n. Otherwise, it has no solution. ■

Thanks to Corollary 31.26, we can use the notation $a^{-1} \bmod n$ to refer to *the* multiplicative inverse of a, modulo n, when a and n are relatively prime. If $\gcd(a, n) = 1$, then the unique solution to the equation $ax \equiv 1 \pmod{n}$ is the integer x returned by EXTENDED-EUCLID, since the equation

$$\gcd(a, n) = 1 = ax + ny$$

implies $ax \equiv 1 \pmod{n}$. Thus, we can compute $a^{-1} \bmod n$ efficiently using EXTENDED-EUCLID.

Exercises

31.4-1
Find all solutions to the equation $35x \equiv 10 \pmod{50}$.

31.4-2
Prove that the equation $ax \equiv ay \pmod{n}$ implies $x \equiv y \pmod{n}$ whenever $\gcd(a, n) = 1$. Show that the condition $\gcd(a, n) = 1$ is necessary by supplying a counterexample with $\gcd(a, n) > 1$.

31.4-3
Consider the following change to line 3 of the procedure MODULAR-LINEAR-EQUATION-SOLVER:

3 $\quad x_0 = x'(b/d) \bmod (n/d)$

Will this work? Explain why or why not.

31.4-4 ⋆
Let p be prime and $f(x) \equiv f_0 + f_1 x + \cdots + f_t x^t \pmod{p}$ be a polynomial of degree t, with coefficients f_i drawn from $\mathbb{Z}_p$. We say that $a \in \mathbb{Z}_p$ is a ***zero*** of f if $f(a) \equiv 0 \pmod{p}$. Prove that if a is a zero of f, then $f(x) \equiv (x - a)g(x) \pmod{p}$ for some polynomial $g(x)$ of degree $t - 1$. Prove by induction on t that if p is prime, then a polynomial $f(x)$ of degree t can have at most t distinct zeros modulo p.

31.5 The Chinese remainder theorem

Around A.D. 100, the Chinese mathematician Sun-Tsŭ solved the problem of finding those integers x that leave remainders 2, 3, and 2 when divided by 3, 5, and 7 respectively. One such solution is $x = 23$; all solutions are of the form $23 + 105k$

for arbitrary integers k. The "Chinese remainder theorem" provides a correspondence between a system of equations modulo a set of pairwise relatively prime moduli (for example, 3, 5, and 7) and an equation modulo their product (for example, 105).

The Chinese remainder theorem has two major applications. Let the integer n be factored as $n = n_1 n_2 \cdots n_k$, where the factors n_i are pairwise relatively prime. First, the Chinese remainder theorem is a descriptive "structure theorem" that describes the structure of $\mathbb{Z}_n$ as identical to that of the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$ with componentwise addition and multiplication modulo n_i in the ith component. Second, this description helps us to design efficient algorithms, since working in each of the systems $\mathbb{Z}_{n_i}$ can be more efficient (in terms of bit operations) than working modulo n.

Theorem 31.27 (Chinese remainder theorem)
Let $n = n_1 n_2 \cdots n_k$, where the n_i are pairwise relatively prime. Consider the correspondence

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) , \tag{31.27}$$

where $a \in \mathbb{Z}_n$, $a_i \in \mathbb{Z}_{n_i}$, and

$$a_i = a \bmod n_i$$

for $i = 1, 2, \ldots, k$. Then, mapping (31.27) is a one-to-one correspondence (bijection) between $\mathbb{Z}_n$ and the Cartesian product $\mathbb{Z}_{n_1} \times \mathbb{Z}_{n_2} \times \cdots \times \mathbb{Z}_{n_k}$. Operations performed on the elements of $\mathbb{Z}_n$ can be equivalently performed on the corresponding k-tuples by performing the operations independently in each coordinate position in the appropriate system. That is, if

$$\begin{aligned} a &\leftrightarrow (a_1, a_2, \ldots, a_k) , \\ b &\leftrightarrow (b_1, b_2, \ldots, b_k) , \end{aligned}$$

then

$$(a + b) \bmod n \leftrightarrow ((a_1 + b_1) \bmod n_1, \ldots, (a_k + b_k) \bmod n_k) , \tag{31.28}$$

$$(a - b) \bmod n \leftrightarrow ((a_1 - b_1) \bmod n_1, \ldots, (a_k - b_k) \bmod n_k) , \tag{31.29}$$

$$(ab) \bmod n \leftrightarrow (a_1 b_1 \bmod n_1, \ldots, a_k b_k \bmod n_k) . \tag{31.30}$$

Proof Transforming between the two representations is fairly straightforward. Going from a to $(a_1, a_2, \ldots, a_k)$ is quite easy and requires only k "mod" operations.

Computing a from inputs $(a_1, a_2, \ldots, a_k)$ is a bit more complicated. We begin by defining $m_i = n/n_i$ for $i = 1, 2, \ldots, k$; thus m_i is the product of all of the n_j's other than n_i: $m_i = n_1 n_2 \cdots n_{i-1} n_{i+1} \cdots n_k$. We next define

$$c_i = m_i(m_i^{-1} \bmod n_i) \tag{31.31}$$

for $i = 1, 2, \ldots, k$. Equation (31.31) is always well defined: since m_i and n_i are relatively prime (by Theorem 31.6), Corollary 31.26 guarantees that $m_i^{-1} \bmod n_i$ exists. Finally, we can compute a as a function of $a_1, a_2, \ldots, a_k$ as follows:

$$a \equiv (a_1c_1 + a_2c_2 + \cdots + a_kc_k) \pmod{n} . \tag{31.32}$$

We now show that equation (31.32) ensures that $a \equiv a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. Note that if $j \neq i$, then $m_j \equiv 0 \pmod{n_i}$, which implies that $c_j \equiv m_j \equiv 0 \pmod{n_i}$. Note also that $c_i \equiv 1 \pmod{n_i}$, from equation (31.31). We thus have the appealing and useful correspondence

$$c_i \leftrightarrow (0, 0, \ldots, 0, 1, 0, \ldots, 0) ,$$

a vector that has 0s everywhere except in the ith coordinate, where it has a 1; the c_i thus form a "basis" for the representation, in a certain sense. For each i, therefore, we have

$$\begin{aligned} a &\equiv a_ic_i && \pmod{n_i} \\ &\equiv a_im_i(m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &\equiv a_i && \pmod{n_i} , \end{aligned}$$

which is what we wished to show: our method of computing a from the a_i's produces a result a that satisfies the constraints $a \equiv a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. The correspondence is one-to-one, since we can transform in both directions. Finally, equations (31.28)–(31.30) follow directly from Exercise 31.1-7, since $x \bmod n_i = (x \bmod n) \bmod n_i$ for any x and $i = 1, 2, \ldots, k$. ■

We shall use the following corollaries later in this chapter.

Corollary 31.28

If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1n_2 \cdots n_k$, then for any integers $a_1, a_2, \ldots, a_k$, the set of simultaneous equations

$$x \equiv a_i \pmod{n_i} ,$$

for $i = 1, 2, \ldots, k$, has a unique solution modulo n for the unknown x. ■

Corollary 31.29

If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1n_2 \cdots n_k$, then for all integers x and a,

$$x \equiv a \pmod{n_i}$$

for $i = 1, 2, \ldots, k$ if and only if

$$x \equiv a \pmod{n} .$$ ■

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	40	15	55	30	5	45	20	60	35	10	50	25
1	26	1	41	16	56	31	6	46	21	61	36	11	51
2	52	27	2	42	17	57	32	7	47	22	62	37	12
3	13	53	28	3	43	18	58	33	8	48	23	63	38
4	39	14	54	29	4	44	19	59	34	9	49	24	64

Figure 31.3 An illustration of the Chinese remainder theorem for $n_1 = 5$ and $n_2 = 13$. For this example, $c_1 = 26$ and $c_2 = 40$. In row i, column j is shown the value of a, modulo 65, such that $a \bmod 5 = i$ and $a \bmod 13 = j$. Note that row 0, column 0 contains a 0. Similarly, row 4, column 12 contains a 64 (equivalent to -1). Since $c_1 = 26$, moving down a row increases a by 26. Similarly, $c_2 = 40$ means that moving right by a column increases a by 40. Increasing a by 1 corresponds to moving diagonally downward and to the right, wrapping around from the bottom to the top and from the right to the left.

As an example of the application of the Chinese remainder theorem, suppose we are given the two equations

$$\begin{aligned} a &\equiv 2 \pmod 5 , \\ a &\equiv 3 \pmod{13} , \end{aligned}$$

so that $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, and $n_2 = m_1 = 13$, and we wish to compute $a \bmod 65$, since $n = n_1 n_2 = 65$. Because $13^{-1} \equiv 2 \pmod 5$ and $5^{-1} \equiv 8 \pmod{13}$, we have

$$\begin{aligned} c_1 &= 13(2 \bmod 5) = 26 , \\ c_2 &= 5(8 \bmod 13) = 40 , \end{aligned}$$

and

$$\begin{aligned} a &\equiv 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &\equiv 52 + 120 \pmod{65} \\ &\equiv 42 \pmod{65} . \end{aligned}$$

See Figure 31.3 for an illustration of the Chinese remainder theorem, modulo 65.

Thus, we can work modulo n by working modulo n directly or by working in the transformed representation using separate modulo n_i computations, as convenient. The computations are entirely equivalent.

Exercises

31.5-1

Find all solutions to the equations $x \equiv 4 \pmod 5$ and $x \equiv 5 \pmod{11}$.

31.5-2
Find all integers x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.

31.5-3
Argue that, under the definitions of Theorem 31.27, if $\gcd(a, n) = 1$, then

$$(a^{-1} \bmod n) \leftrightarrow ((a_1^{-1} \bmod n_1), (a_2^{-1} \bmod n_2), \ldots, (a_k^{-1} \bmod n_k)) .$$

31.5-4
Under the definitions of Theorem 31.27, prove that for any polynomial f, the number of roots of the equation $f(x) \equiv 0 \pmod{n}$ equals the product of the number of roots of each of the equations $f(x) \equiv 0 \pmod{n_1}$, $f(x) \equiv 0 \pmod{n_2}$, ..., $f(x) \equiv 0 \pmod{n_k}$.

31.6 Powers of an element

Just as we often consider the multiples of a given element a, modulo n, we consider the sequence of powers of a, modulo n, where $a \in \mathbb{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \ldots, \tag{31.33}$$

modulo n. Indexing from 0, the 0th value in this sequence is $a^0 \bmod n = 1$, and the ith value is $a^i \bmod n$. For example, the powers of 3 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$3^i \bmod 7$	1	3	2	6	4	5	1	3	2	6	4	5	$\cdots$

whereas the powers of 2 modulo 7 are

i	0	1	2	3	4	5	6	7	8	9	10	11	$\cdots$
$2^i \bmod 7$	1	2	4	1	2	4	1	2	4	1	2	4	$\cdots$

In this section, let $\langle a \rangle$ denote the subgroup of $\mathbb{Z}_n^*$ generated by a by repeated multiplication, and let $\mathrm{ord}_n(a)$ (the "order of a, modulo n") denote the order of a in $\mathbb{Z}_n^*$. For example, $\langle 2 \rangle = \{1, 2, 4\}$ in $\mathbb{Z}_7^*$, and $\mathrm{ord}_7(2) = 3$. Using the definition of the Euler phi function $\phi(n)$ as the size of $\mathbb{Z}_n^*$ (see Section 31.3), we now translate Corollary 31.19 into the notation of $\mathbb{Z}_n^*$ to obtain Euler's theorem and specialize it to $\mathbb{Z}_p^*$, where p is prime, to obtain Fermat's theorem.

Theorem 31.30 (Euler's theorem)
For any integer $n > 1$,

$$a^{\phi(n)} \equiv 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^* .$$

■

Theorem 31.31 (Fermat's theorem)
If p is prime, then

$a^{p-1} \equiv 1 \pmod{p}$ for all $a \in \mathbb{Z}_p^*$.

Proof By equation (31.21), $\phi(p) = p - 1$ if p is prime. ■

Fermat's theorem applies to every element in $\mathbb{Z}_p$ except 0, since $0 \notin \mathbb{Z}_p^*$. For all $a \in \mathbb{Z}_p$, however, we have $a^p \equiv a \pmod{p}$ if p is prime.

If $\text{ord}_n(g) = |\mathbb{Z}_n^*|$, then every element in $\mathbb{Z}_n^*$ is a power of g, modulo n, and g is a ***primitive root*** or a ***generator*** of $\mathbb{Z}_n^*$. For example, 3 is a primitive root, modulo 7, but 2 is not a primitive root, modulo 7. If $\mathbb{Z}_n^*$ possesses a primitive root, the group $\mathbb{Z}_n^*$ is ***cyclic***. We omit the proof of the following theorem, which is proven by Niven and Zuckerman [265].

Theorem 31.32
The values of $n > 1$ for which $\mathbb{Z}_n^*$ is cyclic are 2, 4, p^e, and $2p^e$, for all primes $p > 2$ and all positive integers e. ■

If g is a primitive root of $\mathbb{Z}_n^*$ and a is any element of $\mathbb{Z}_n^*$, then there exists a z such that $g^z \equiv a \pmod{n}$. This z is a ***discrete logarithm*** or an ***index*** of a, modulo n, to the base g; we denote this value as $\text{ind}_{n,g}(a)$.

Theorem 31.33 (Discrete logarithm theorem)
If g is a primitive root of $\mathbb{Z}_n^*$, then the equation $g^x \equiv g^y \pmod{n}$ holds if and only if the equation $x \equiv y \pmod{\phi(n)}$ holds.

Proof Suppose first that $x \equiv y \pmod{\phi(n)}$. Then, $x = y + k\phi(n)$ for some integer k. Therefore,

$$\begin{aligned} g^x &\equiv g^{y+k\phi(n)} && \pmod{n} \\ &\equiv g^y \cdot (g^{\phi(n)})^k && \pmod{n} \\ &\equiv g^y \cdot 1^k && \pmod{n} \qquad \text{(by Euler's theorem)} \\ &\equiv g^y && \pmod{n}\,. \end{aligned}$$

Conversely, suppose that $g^x \equiv g^y \pmod{n}$. Because the sequence of powers of g generates every element of $\langle g \rangle$ and $|\langle g \rangle| = \phi(n)$, Corollary 31.18 implies that the sequence of powers of g is periodic with period $\phi(n)$. Therefore, if $g^x \equiv g^y \pmod{n}$, then we must have $x \equiv y \pmod{\phi(n)}$. ■

We now turn our attention to the square roots of 1, modulo a prime power. The following theorem will be useful in our development of a primality-testing algorithm in Section 31.8.

Theorem 31.34
If p is an odd prime and $e \geq 1$, then the equation

$$x^2 \equiv 1 \pmod{p^e} \tag{31.34}$$

has only two solutions, namely $x = 1$ and $x = -1$.

Proof Equation (31.34) is equivalent to

$$p^e \mid (x-1)(x+1) .$$

Since $p > 2$, we can have $p \mid (x-1)$ or $p \mid (x+1)$, but not both. (Otherwise, by property (31.3), p would also divide their difference $(x+1)-(x-1) = 2$.) If $p \nmid (x-1)$, then $\gcd(p^e, x-1) = 1$, and by Corollary 31.5, we would have $p^e \mid (x+1)$. That is, $x \equiv -1 \pmod{p^e}$. Symmetrically, if $p \nmid (x+1)$, then $\gcd(p^e, x+1) = 1$, and Corollary 31.5 implies that $p^e \mid (x-1)$, so that $x \equiv 1 \pmod{p^e}$. Therefore, either $x \equiv -1 \pmod{p^e}$ or $x \equiv 1 \pmod{p^e}$. ■

A number x is a ***nontrivial square root of 1, modulo n***, if it satisfies the equation $x^2 \equiv 1 \pmod{n}$ but x is equivalent to neither of the two "trivial" square roots: 1 or -1, modulo n. For example, 6 is a nontrivial square root of 1, modulo 35. We shall use the following corollary to Theorem 31.34 in the correctness proof in Section 31.8 for the Miller-Rabin primality-testing procedure.

Corollary 31.35
If there exists a nontrivial square root of 1, modulo n, then n is composite.

Proof By the contrapositive of Theorem 31.34, if there exists a nontrivial square root of 1, modulo n, then n cannot be an odd prime or a power of an odd prime. If $x^2 \equiv 1 \pmod{2}$, then $x \equiv 1 \pmod{2}$, and so all square roots of 1, modulo 2, are trivial. Thus, n cannot be prime. Finally, we must have $n > 1$ for a nontrivial square root of 1 to exist. Therefore, n must be composite. ■

Raising to powers with repeated squaring

A frequently occurring operation in number-theoretic computations is raising one number to a power modulo another number, also known as ***modular exponentiation***. More precisely, we would like an efficient way to compute $a^b \bmod n$, where a and b are nonnegative integers and n is a positive integer. Modular exponentiation is an essential operation in many primality-testing routines and in the RSA public-key cryptosystem. The method of ***repeated squaring*** solves this problem efficiently using the binary representation of b.

Let $\langle b_k, b_{k-1}, \ldots, b_1, b_0 \rangle$ be the binary representation of b. (That is, the binary representation is $k+1$ bits long, b_k is the most significant bit, and b_0 is the least

i	9	8	7	6	5	4	3	2	1	0
b_i	1	0	0	0	1	1	0	0	0	0
c	1	2	4	8	17	35	70	140	280	560
d	7	49	157	526	160	241	298	166	67	1

Figure 31.4 The results of MODULAR-EXPONENTIATION when computing $a^b \pmod n$, where $a = 7$, $b = 560 = \langle 1000110000 \rangle$, and $n = 561$. The values are shown after each execution of the **for** loop. The final result is 1.

significant bit.) The following procedure computes $a^c \bmod n$ as c is increased by doublings and incrementations from 0 to b.

MODULAR-EXPONENTIATION(a, b, n)

```
 1  c = 0
 2  d = 1
 3  let ⟨b_k, b_{k-1}, ..., b_0⟩ be the binary representation of b
 4  for i = k downto 0
 5      c = 2c
 6      d = (d · d) mod n
 7      if b_i == 1
 8          c = c + 1
 9          d = (d · a) mod n
10  return d
```

The essential use of squaring in line 6 of each iteration explains the name "repeated squaring." As an example, for $a = 7$, $b = 560$, and $n = 561$, the algorithm computes the sequence of values modulo 561 shown in Figure 31.4; the sequence of exponents used appears in the row of the table labeled by c.

The variable c is not really needed by the algorithm but is included for the following two-part loop invariant:

> Just prior to each iteration of the **for** loop of lines 4–9,
>
> 1. The value of c is the same as the prefix $\langle b_k, b_{k-1}, \ldots, b_{i+1} \rangle$ of the binary representation of b, and
> 2. $d = a^c \bmod n$.

We use this loop invariant as follows:

Initialization: Initially, $i = k$, so that the prefix $\langle b_k, b_{k-1}, \ldots, b_{i+1} \rangle$ is empty, which corresponds to $c = 0$. Moreover, $d = 1 = a^0 \bmod n$.

Maintenance: Let c' and d' denote the values of c and d at the end of an iteration of the **for** loop, and thus the values prior to the next iteration. Each iteration updates $c' = 2c$ (if $b_i = 0$) or $c' = 2c + 1$ (if $b_i = 1$), so that c will be correct prior to the next iteration. If $b_i = 0$, then $d' = d^2 \bmod n = (a^c)^2 \bmod n = a^{2c} \bmod n = a^{c'} \bmod n$. If $b_i = 1$, then $d' = d^2 a \bmod n = (a^c)^2 a \bmod n = a^{2c+1} \bmod n = a^{c'} \bmod n$. In either case, $d = a^c \bmod n$ prior to the next iteration.

Termination: At termination, $i = -1$. Thus, $c = b$, since c has the value of the prefix $\langle b_k, b_{k-1}, \ldots, b_0 \rangle$ of b's binary representation. Hence $d = a^c \bmod n = a^b \bmod n$.

If the inputs a, b, and n are β-bit numbers, then the total number of arithmetic operations required is $O(\beta)$ and the total number of bit operations required is $O(\beta^3)$.

Exercises

31.6-1
Draw a table showing the order of every element in $\mathbb{Z}_{11}^*$. Pick the smallest primitive root g and compute a table giving $\text{ind}_{11,g}(x)$ for all $x \in \mathbb{Z}_{11}^*$.

31.6-2
Give a modular exponentiation algorithm that examines the bits of b from right to left instead of left to right.

31.6-3
Assuming that you know $\phi(n)$, explain how to compute $a^{-1} \bmod n$ for any $a \in \mathbb{Z}_n^*$ using the procedure MODULAR-EXPONENTIATION.

31.7 The RSA public-key cryptosystem

With a public-key cryptosystem, we can encrypt messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode them. A public-key cryptosystem also enables a party to append an unforgeable "digital signature" to the end of an electronic message. Such a signature is the electronic version of a handwritten signature on a paper document. It can be easily checked by anyone, forged by no one, yet loses its validity if any bit of the message is altered. It therefore provides authentication of both the identity of the signer and the contents of the signed message. It is the perfect tool

for electronically signed business contracts, electronic checks, electronic purchase orders, and other electronic communications that parties wish to authenticate.

The RSA public-key cryptosystem relies on the dramatic difference between the ease of finding large prime numbers and the difficulty of factoring the product of two large prime numbers. Section 31.8 describes an efficient procedure for finding large prime numbers, and Section 31.9 discusses the problem of factoring large integers.

Public-key cryptosystems

In a public-key cryptosystem, each participant has both a ***public key*** and a ***secret key***. Each key is a piece of information. For example, in the RSA cryptosystem, each key consists of a pair of integers. The participants "Alice" and "Bob" are traditionally used in cryptography examples; we denote their public and secret keys as P_A, S_A for Alice and P_B, S_B for Bob.

Each participant creates his or her own public and secret keys. Secret keys are kept secret, but public keys can be revealed to anyone or even published. In fact, it is often convenient to assume that everyone's public key is available in a public directory, so that any participant can easily obtain the public key of any other participant.

The public and secret keys specify functions that can be applied to any message. Let $\mathcal{D}$ denote the set of permissible messages. For example, $\mathcal{D}$ might be the set of all finite-length bit sequences. In the simplest, and original, formulation of public-key cryptography, we require that the public and secret keys specify one-to-one functions from $\mathcal{D}$ to itself. We denote the function corresponding to Alice's public key P_A by $P_A()$ and the function corresponding to her secret key S_A by $S_A()$. The functions $P_A()$ and $S_A()$ are thus permutations of $\mathcal{D}$. We assume that the functions $P_A()$ and $S_A()$ are efficiently computable given the corresponding key P_A or S_A.

The public and secret keys for any participant are a "matched pair" in that they specify functions that are inverses of each other. That is,

$$M = S_A(P_A(M)) , \tag{31.35}$$

$$M = P_A(S_A(M)) \tag{31.36}$$

for any message $M \in \mathcal{D}$. Transforming M with the two keys P_A and S_A successively, in either order, yields the message M back.

In a public-key cryptosystem, we require that no one but Alice be able to compute the function $S_A()$ in any practical amount of time. This assumption is crucial to keeping encrypted mail sent to Alice private and to knowing that Alice's digital signatures are authentic. Alice must keep S_A secret; if she does not, she loses her uniqueness and the cryptosystem cannot provide her with unique capabilities. The assumption that only Alice can compute $S_A()$ must hold even though everyone

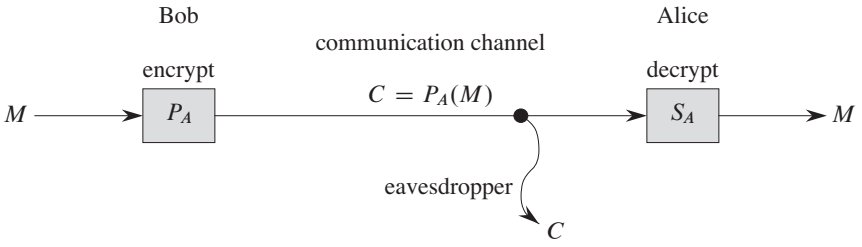


Figure 31.5 Encryption in a public key system. Bob encrypts the message M using Alice's public key P_A and transmits the resulting ciphertext $C = P_A(M)$ over a communication channel to Alice. An eavesdropper who captures the transmitted ciphertext gains no information about M. Alice receives C and decrypts it using her secret key to obtain the original message $M = S_A(C)$.

knows P_A and can compute $P_A()$, the inverse function to $S_A()$, efficiently. In order to design a workable public-key cryptosystem, we must figure out how to create a system in which we can reveal a transformation $P_A()$ without thereby revealing how to compute the corresponding inverse transformation $S_A()$. This task appears formidable, but we shall see how to accomplish it.

In a public-key cryptosystem, encryption works as shown in Figure 31.5. Suppose Bob wishes to send Alice a message M encrypted so that it will look like unintelligible gibberish to an eavesdropper. The scenario for sending the message goes as follows.

- Bob obtains Alice's public key P_A (from a public directory or directly from Alice).
- Bob computes the ***ciphertext*** $C = P_A(M)$ corresponding to the message M and sends C to Alice.
- When Alice receives the ciphertext C, she applies her secret key S_A to retrieve the original message: $S_A(C) = S_A(P_A(M)) = M$.

Because $S_A()$ and $P_A()$ are inverse functions, Alice can compute M from C. Because only Alice is able to compute $S_A()$, Alice is the only one who can compute M from C. Because Bob encrypts M using $P_A()$, only Alice can understand the transmitted message.

We can just as easily implement digital signatures within our formulation of a public-key cryptosystem. (There are other ways of approaching the problem of constructing digital signatures, but we shall not go into them here.) Suppose now that Alice wishes to send Bob a digitally signed response M'. Figure 31.6 shows how the digital-signature scenario proceeds.

- Alice computes her ***digital signature*** σ for the message M' using her secret key S_A and the equation $\sigma = S_A(M')$.

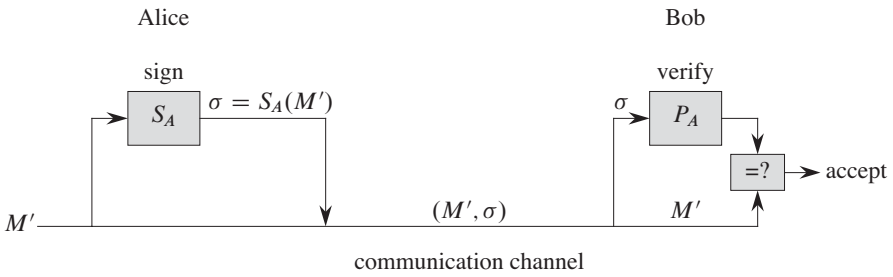


Figure 31.6 Digital signatures in a public-key system. Alice signs the message M' by appending her digital signature $\sigma = S_A(M')$ to it. She transmits the message/signature pair (M', σ) to Bob, who verifies it by checking the equation $M' = P_A(\sigma)$. If the equation holds, he accepts (M', σ) as a message that Alice has signed.

- Alice sends the message/signature pair (M', σ) to Bob.
- When Bob receives (M', σ), he can verify that it originated from Alice by using Alice's public key to verify the equation $M' = P_A(\sigma)$. (Presumably, M' contains Alice's name, so Bob knows whose public key to use.) If the equation holds, then Bob concludes that the message M' was actually signed by Alice. If the equation fails to hold, Bob concludes either that the message M' or the digital signature σ was corrupted by transmission errors or that the pair (M', σ) is an attempted forgery.

Because a digital signature provides both authentication of the signer's identity and authentication of the contents of the signed message, it is analogous to a handwritten signature at the end of a written document.

A digital signature must be verifiable by anyone who has access to the signer's public key. A signed message can be verified by one party and then passed on to other parties who can also verify the signature. For example, the message might be an electronic check from Alice to Bob. After Bob verifies Alice's signature on the check, he can give the check to his bank, who can then also verify the signature and effect the appropriate funds transfer.

A signed message is not necessarily encrypted; the message can be "in the clear" and not protected from disclosure. By composing the above protocols for encryption and for signatures, we can create messages that are both signed and encrypted. The signer first appends his or her digital signature to the message and then encrypts the resulting message/signature pair with the public key of the intended recipient. The recipient decrypts the received message with his or her secret key to obtain both the original message and its digital signature. The recipient can then verify the signature using the public key of the signer. The corresponding combined process using paper-based systems would be to sign the paper document and

then seal the document inside a paper envelope that is opened only by the intended recipient.

The RSA cryptosystem

In the ***RSA public-key cryptosystem***, a participant creates his or her public and secret keys with the following procedure:

1. Select at random two large prime numbers p and q such that $p \neq q$. The primes p and q might be, say, 1024 bits each.
2. Compute $n = pq$.
3. Select a small odd integer e that is relatively prime to $\phi(n)$, which, by equation (31.20), equals $(p-1)(q-1)$.
4. Compute d as the multiplicative inverse of e, modulo $\phi(n)$. (Corollary 31.26 guarantees that d exists and is uniquely defined. We can use the technique of Section 31.4 to compute d, given e and $\phi(n)$.)
5. Publish the pair $P = (e, n)$ as the participant's ***RSA public key***.
6. Keep secret the pair $S = (d, n)$ as the participant's ***RSA secret key***.

For this scheme, the domain $\mathcal{D}$ is the set $\mathbb{Z}_n$. To transform a message M associated with a public key $P = (e, n)$, compute

$$P(M) = M^e \bmod n \ . \tag{31.37}$$

To transform a ciphertext C associated with a secret key $S = (d, n)$, compute

$$S(C) = C^d \bmod n \ . \tag{31.38}$$

These equations apply to both encryption and signatures. To create a signature, the signer applies his or her secret key to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to it, rather than to a message to be encrypted.

We can implement the public-key and secret-key operations using the procedure MODULAR-EXPONENTIATION described in Section 31.6. To analyze the running time of these operations, assume that the public key (e, n) and secret key (d, n) satisfy $\lg e = O(1)$, $\lg d \leq \beta$, and $\lg n \leq \beta$. Then, applying a public key requires $O(1)$ modular multiplications and uses $O(\beta^2)$ bit operations. Applying a secret key requires $O(\beta)$ modular multiplications, using $O(\beta^3)$ bit operations.

Theorem 31.36 (Correctness of RSA)
The RSA equations (31.37) and (31.38) define inverse transformations of $\mathbb{Z}_n$ satisfying equations (31.35) and (31.36).

Proof From equations (31.37) and (31.38), we have that for any $M \in \mathbb{Z}_n$,

$$P(S(M)) = S(P(M)) = M^{ed} \pmod{n} .$$

Since e and d are multiplicative inverses modulo $\phi(n) = (p-1)(q-1)$,

$$ed = 1 + k(p-1)(q-1)$$

for some integer k. But then, if $M \not\equiv 0 \pmod{p}$, we have

$$\begin{aligned} M^{ed} &\equiv M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M((M \bmod p)^{p-1})^{k(q-1)} && \pmod{p} \\ &\equiv M(1)^{k(q-1)} && \pmod{p} \qquad \text{(by Theorem 31.31)} \\ &\equiv M && \pmod{p} . \end{aligned}$$

Also, $M^{ed} \equiv M \pmod{p}$ if $M \equiv 0 \pmod{p}$. Thus,

$$M^{ed} \equiv M \pmod{p}$$

for all M. Similarly,

$$M^{ed} \equiv M \pmod{q}$$

for all M. Thus, by Corollary 31.29 to the Chinese remainder theorem,

$$M^{ed} \equiv M \pmod{n}$$

for all M. ■

The security of the RSA cryptosystem rests in large part on the difficulty of factoring large integers. If an adversary can factor the modulus n in a public key, then the adversary can derive the secret key from the public key, using the knowledge of the factors p and q in the same way that the creator of the public key used them. Therefore, if factoring large integers is easy, then breaking the RSA cryptosystem is easy. The converse statement, that if factoring large integers is hard, then breaking RSA is hard, is unproven. After two decades of research, however, no easier method has been found to break the RSA public-key cryptosystem than to factor the modulus n. And as we shall see in Section 31.9, factoring large integers is surprisingly difficult. By randomly selecting and multiplying together two 1024-bit primes, we can create a public key that cannot be "broken" in any feasible amount of time with current technology. In the absence of a fundamental breakthrough in the design of number-theoretic algorithms, and when implemented with care following recommended standards, the RSA cryptosystem is capable of providing a high degree of security in applications.

In order to achieve security with the RSA cryptosystem, however, we should use integers that are quite long—hundreds or even more than one thousand bits

long—to resist possible advances in the art of factoring. At the time of this writing (2009), RSA moduli were commonly in the range of 768 to 2048 bits. To create moduli of such sizes, we must be able to find large primes efficiently. Section 31.8 addresses this problem.

For efficiency, RSA is often used in a "hybrid" or "key-management" mode with fast non-public-key cryptosystems. With such a system, the encryption and decryption keys are identical. If Alice wishes to send a long message M to Bob privately, she selects a random key K for the fast non-public-key cryptosystem and encrypts M using K, obtaining ciphertext C. Here, C is as long as M, but K is quite short. Then, she encrypts K using Bob's public RSA key. Since K is short, computing $P_B(K)$ is fast (much faster than computing $P_B(M)$). She then transmits $(C, P_B(K))$ to Bob, who decrypts $P_B(K)$ to obtain K and then uses K to decrypt C, obtaining M.

We can use a similar hybrid approach to make digital signatures efficiently. This approach combines RSA with a public ***collision-resistant hash function*** h—a function that is easy to compute but for which it is computationally infeasible to find two messages M and M' such that $h(M) = h(M')$. The value $h(M)$ is a short (say, 256-bit) "fingerprint" of the message M. If Alice wishes to sign a message M, she first applies h to M to obtain the fingerprint $h(M)$, which she then encrypts with her secret key. She sends $(M, S_A(h(M)))$ to Bob as her signed version of M. Bob can verify the signature by computing $h(M)$ and verifying that P_A applied to $S_A(h(M))$ as received equals $h(M)$. Because no one can create two messages with the same fingerprint, it is computationally infeasible to alter a signed message and preserve the validity of the signature.

Finally, we note that the use of ***certificates*** makes distributing public keys much easier. For example, assume there is a "trusted authority" T whose public key is known by everyone. Alice can obtain from T a signed message (her certificate) stating that "Alice's public key is P_A." This certificate is "self-authenticating" since everyone knows P_T. Alice can include her certificate with her signed messages, so that the recipient has Alice's public key immediately available in order to verify her signature. Because her key was signed by T, the recipient knows that Alice's key is really Alice's.

Exercises

31.7-1

Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of the message $M = 100$?

31.7-2
Prove that if Alice's public exponent e is 3 and an adversary obtains Alice's secret exponent d, where $0 < d < \phi(n)$, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n. (Although you are not asked to prove it, you may be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [255].)

31.7-3 ⋆
Prove that RSA is multiplicative in the sense that

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1M_2) \pmod{n} .$$

Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1 percent of messages from $\mathbb{Z}_n$ encrypted with P_A, then he could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

⋆ 31.8 Primality testing

In this section, we consider the problem of finding large primes. We begin with a discussion of the density of primes, proceed to examine a plausible, but incomplete, approach to primality testing, and then present an effective randomized primality test due to Miller and Rabin.

The density of prime numbers

For many applications, such as cryptography, we need to find large "random" primes. Fortunately, large primes are not too rare, so that it is feasible to test random integers of the appropriate size until we find a prime. The ***prime distribution function*** $\pi(n)$ specifies the number of primes that are less than or equal to n. For example, $\pi(10) = 4$, since there are 4 prime numbers less than or equal to 10, namely, 2, 3, 5, and 7. The prime number theorem gives a useful approximation to $\pi(n)$.

Theorem 31.37 (Prime number theorem)

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1 . \qquad \blacksquare$$

The approximation $n/\ln n$ gives reasonably accurate estimates of $\pi(n)$ even for small n. For example, it is off by less than 6% at $n = 10^9$, where $\pi(n) =$

50,847,534 and $n/\ln n \approx 48{,}254{,}942$. (To a number theorist, 10^9 is a small number.)

We can view the process of randomly selecting an integer n and determining whether it is prime as a Bernoulli trial (see Section C.4). By the prime number theorem, the probability of a success—that is, the probability that n is prime—is approximately $1/\ln n$. The geometric distribution tells us how many trials we need to obtain a success, and by equation (C.32), the expected number of trials is approximately $\ln n$. Thus, we would expect to examine approximately $\ln n$ integers chosen randomly near n in order to find a prime that is of the same length as n. For example, we expect that finding a 1024-bit prime would require testing approximately $\ln 2^{1024} \approx 710$ randomly chosen 1024-bit numbers for primality. (Of course, we can cut this figure in half by choosing only odd integers.)

In the remainder of this section, we consider the problem of determining whether or not a large odd integer n is prime. For notational convenience, we assume that n has the prime factorization

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} , \tag{31.39}$$

where $r \geq 1$, $p_1, p_2, \ldots, p_r$ are the prime factors of n, and $e_1, e_2, \ldots, e_r$ are positive integers. The integer n is prime if and only if $r = 1$ and $e_1 = 1$.

One simple approach to the problem of testing for primality is ***trial division***. We try dividing n by each integer $2, 3, \ldots, \lfloor \sqrt{n} \rfloor$. (Again, we may skip even integers greater than 2.) It is easy to see that n is prime if and only if none of the trial divisors divides n. Assuming that each trial division takes constant time, the worst-case running time is $\Theta(\sqrt{n})$, which is exponential in the length of n. (Recall that if n is encoded in binary using β bits, then $\beta = \lceil \lg(n+1) \rceil$, and so $\sqrt{n} = \Theta(2^{\beta/2})$.) Thus, trial division works well only if n is very small or happens to have a small prime factor. When it works, trial division has the advantage that it not only determines whether n is prime or composite, but also determines one of n's prime factors if n is composite.

In this section, we are interested only in finding out whether a given number n is prime; if n is composite, we are not concerned with finding its prime factorization. As we shall see in Section 31.9, computing the prime factorization of a number is computationally expensive. It is perhaps surprising that it is much easier to tell whether or not a given number is prime than it is to determine the prime factorization of the number if it is not prime.

Pseudoprimality testing

We now consider a method for primality testing that "almost works" and in fact is good enough for many practical applications. Later on, we shall present a re-

finement of this method that removes the small defect. Let $\mathbb{Z}_n^+$ denote the nonzero elements of $\mathbb{Z}_n$:

$$\mathbb{Z}_n^+ = \{1, 2, \ldots, n-1\} \ .$$

If n is prime, then $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$.

We say that n is a ***base-a pseudoprime*** if n is composite and

$$a^{n-1} \equiv 1 \pmod{n} \ . \tag{31.40}$$

Fermat's theorem (Theorem 31.31) implies that if n is prime, then n satisfies equation (31.40) for every a in $\mathbb{Z}_n^+$. Thus, if we can find any $a \in \mathbb{Z}_n^+$ such that n does *not* satisfy equation (31.40), then n is certainly composite. Surprisingly, the converse *almost* holds, so that this criterion forms an almost perfect test for primality. We test to see whether n satisfies equation (31.40) for $a = 2$. If not, we declare n to be composite by returning COMPOSITE. Otherwise, we return PRIME, guessing that n is prime (when, in fact, all we know is that n is either prime or a base-2 pseudoprime).

The following procedure pretends in this manner to be checking the primality of n. It uses the procedure MODULAR-EXPONENTIATION from Section 31.6. We assume that the input n is an odd integer greater than 2.

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2, n − 1, n) ≢ 1 (mod n)
2       return COMPOSITE        // definitely
3  else return PRIME            // we hope!
```

This procedure can make errors, but only of one type. That is, if it says that n is composite, then it is always correct. If it says that n is prime, however, then it makes an error only if n is a base-2 pseudoprime.

How often does this procedure err? Surprisingly rarely. There are only 22 values of n less than 10,000 for which it errs; the first four such values are 341, 561, 645, and 1105. We won't prove it, but the probability that this program makes an error on a randomly chosen β-bit number goes to zero as $\beta \to \infty$. Using more precise estimates due to Pomerance [279] of the number of base-2 pseudoprimes of a given size, we may estimate that a randomly chosen 512-bit number that is called prime by the above procedure has less than one chance in 10^{20} of being a base-2 pseudoprime, and a randomly chosen 1024-bit number that is called prime has less than one chance in 10^{41} of being a base-2 pseudoprime. So if you are merely trying to find a large prime for some application, for all practical purposes you almost never go wrong by choosing large numbers at random until one of them causes PSEUDOPRIME to return PRIME. But when the numbers being tested for primality are not randomly chosen, we need a better approach for testing primality.

As we shall see, a little more cleverness, and some randomization, will yield a primality-testing routine that works well on all inputs.

Unfortunately, we cannot entirely eliminate all the errors by simply checking equation (31.40) for a second base number, say $a = 3$, because there exist composite integers n, known as ***Carmichael numbers***, that satisfy equation (31.40) for *all* $a \in \mathbb{Z}_n^*$. (We note that equation (31.40) does fail when $\gcd(a, n) > 1$—that is, when $a \notin \mathbb{Z}_n^*$—but hoping to demonstrate that n is composite by finding such an a can be difficult if n has only large prime factors.) The first three Carmichael numbers are 561, 1105, and 1729. Carmichael numbers are extremely rare; there are, for example, only 255 of them less than 100,000,000. Exercise 31.8-2 helps explain why they are so rare.

We next show how to improve our primality test so that it won't be fooled by Carmichael numbers.

The Miller-Rabin randomized primality test

The Miller-Rabin primality test overcomes the problems of the simple test PSEUDOPRIME with two modifications:

- It tries several randomly chosen base values a instead of just one base value.
- While computing each modular exponentiation, it looks for a nontrivial square root of 1, modulo n, during the final set of squarings. If it finds one, it stops and returns COMPOSITE. Corollary 31.35 from Section 31.6 justifies detecting composites in this manner.

The pseudocode for the Miller-Rabin primality test follows. The input $n > 2$ is the odd number to be tested for primality, and s is the number of randomly chosen base values from $\mathbb{Z}_n^+$ to be tried. The code uses the random-number generator RANDOM described on page 117: RANDOM$(1, n-1)$ returns a randomly chosen integer a satisfying $1 \le a \le n-1$. The code uses an auxiliary procedure WITNESS such that WITNESS(a, n) is TRUE if and only if a is a "witness" to the compositeness of n—that is, if it is possible using a to prove (in a manner that we shall see) that n is composite. The test WITNESS(a, n) is an extension of, but more effective than, the test

$$a^{n-1} \not\equiv 1 \pmod{n}$$

that formed the basis (using $a = 2$) for PSEUDOPRIME. We first present and justify the construction of WITNESS, and then we shall show how we use it in the Miller-Rabin primality test. Let $n - 1 = 2^t u$ where $t \ge 1$ and u is odd; i.e., the binary representation of $n - 1$ is the binary representation of the odd integer u followed by exactly t zeros. Therefore, $a^{n-1} \equiv (a^u)^{2^t} \pmod{n}$, so that we can

compute $a^{n-1} \bmod n$ by first computing $a^u \bmod n$ and then squaring the result t times successively.

```
WITNESS(a, n)
1  let t and u be such that t ≥ 1, u is odd, and n − 1 = 2^t u
2  x_0 = MODULAR-EXPONENTIATION(a, u, n)
3  for i = 1 to t
4       x_i = x_{i−1}^2 mod n
5       if x_i == 1 and x_{i−1} ≠ 1 and x_{i−1} ≠ n − 1
6            return TRUE
7  if x_t ≠ 1
8       return TRUE
9  return FALSE
```

This pseudocode for WITNESS computes $a^{n-1} \bmod n$ by first computing the value $x_0 = a^u \bmod n$ in line 2 and then squaring the result t times in a row in the **for** loop of lines 3–6. By induction on i, the sequence $x_0, x_1, \ldots, x_t$ of values computed satisfies the equation $x_i \equiv a^{2^i u} \pmod{n}$ for $i = 0, 1, \ldots, t$, so that in particular $x_t \equiv a^{n-1} \pmod{n}$. After line 4 performs a squaring step, however, the loop may terminate early if lines 5–6 detect that a nontrivial square root of 1 has just been discovered. (We shall explain these tests shortly.) If so, the algorithm stops and returns TRUE. Lines 7–8 return TRUE if the value computed for $x_t \equiv a^{n-1} \pmod{n}$ is not equal to 1, just as the PSEUDOPRIME procedure returns COMPOSITE in this case. Line 9 returns FALSE if we haven't returned TRUE in lines 6 or 8.

We now argue that if WITNESS(a, n) returns TRUE, then we can construct a proof that n is composite using a as a witness.

If WITNESS returns TRUE from line 8, then it has discovered that $x_t = a^{n-1} \bmod n \neq 1$. If n is prime, however, we have by Fermat's theorem (Theorem 31.31) that $a^{n-1} \equiv 1 \pmod{n}$ for all $a \in \mathbb{Z}_n^+$. Therefore, n cannot be prime, and the equation $a^{n-1} \bmod n \neq 1$ proves this fact.

If WITNESS returns TRUE from line 6, then it has discovered that x_{i-1} is a nontrivial square root of 1, modulo n, since we have that $x_{i-1} \not\equiv \pm 1 \pmod{n}$ yet $x_i \equiv x_{i-1}^2 \equiv 1 \pmod{n}$. Corollary 31.35 states that only if n is composite can there exist a nontrivial square root of 1 modulo n, so that demonstrating that x_{i-1} is a nontrivial square root of 1 modulo n proves that n is composite.

This completes our proof of the correctness of WITNESS. If we find that the call WITNESS(a, n) returns TRUE, then n is surely composite, and the witness a, along with the reason that the procedure returns TRUE (did it return from line 6 or from line 8?), provides a proof that n is composite.

At this point, we briefly present an alternative description of the behavior of WITNESS as a function of the sequence $X = \langle x_0, x_1, \ldots, x_t \rangle$, which we shall find useful later on, when we analyze the efficiency of the Miller-Rabin primality test. Note that if $x_i = 1$ for some $0 \le i < t$, WITNESS might not compute the rest of the sequence. If it were to do so, however, each value $x_{i+1}, x_{i+2}, \ldots, x_t$ would be 1, and we consider these positions in the sequence X as being all 1s. We have four cases:

1. $X = \langle \ldots, d \rangle$, where $d \neq 1$: the sequence X does not end in 1. Return TRUE in line 8; a is a witness to the compositeness of n (by Fermat's Theorem).
2. $X = \langle 1, 1, \ldots, 1 \rangle$: the sequence X is all 1s. Return FALSE; a is not a witness to the compositeness of n.
3. $X = \langle \ldots, -1, 1, \ldots, 1 \rangle$: the sequence X ends in 1, and the last non-1 is equal to -1. Return FALSE; a is not a witness to the compositeness of n.
4. $X = \langle \ldots, d, 1, \ldots, 1 \rangle$, where $d \neq \pm 1$: the sequence X ends in 1, but the last non-1 is not -1. Return TRUE in line 6; a is a witness to the compositeness of n, since d is a nontrivial square root of 1.

We now examine the Miller-Rabin primality test based on the use of WITNESS. Again, we assume that n is an odd integer greater than 2.

```
MILLER-RABIN(n, s)
1  for j = 1 to s
2      a = RANDOM(1, n − 1)
3      if WITNESS(a, n)
4          return COMPOSITE          // definitely
5  return PRIME                      // almost surely
```

The procedure MILLER-RABIN is a probabilistic search for a proof that n is composite. The main loop (beginning on line 1) picks up to s random values of a from $\mathbb{Z}_n^+$ (line 2). If one of the a's picked is a witness to the compositeness of n, then MILLER-RABIN returns COMPOSITE on line 4. Such a result is always correct, by the correctness of WITNESS. If MILLER-RABIN finds no witness in s trials, then the procedure assumes that this is because no witnesses exist, and therefore it assumes that n is prime. We shall see that this result is likely to be correct if s is large enough, but that there is still a tiny chance that the procedure may be unlucky in its choice of a's and that witnesses do exist even though none has been found.

To illustrate the operation of MILLER-RABIN, let n be the Carmichael number 561, so that $n - 1 = 560 = 2^4 \cdot 35$, $t = 4$, and $u = 35$. If the procedure chooses $a = 7$ as a base, Figure 31.4 in Section 31.6 shows that WITNESS computes $x_0 \equiv a^{35} \equiv 241 \pmod{561}$ and thus computes the sequence

$X = \langle 241, 298, 166, 67, 1\rangle$. Thus, WITNESS discovers a nontrivial square root of 1 in the last squaring step, since $a^{280} \equiv 67 \pmod{n}$ and $a^{560} \equiv 1 \pmod{n}$. Therefore, $a = 7$ is a witness to the compositeness of n, WITNESS$(7, n)$ returns TRUE, and MILLER-RABIN returns COMPOSITE.

If n is a β-bit number, MILLER-RABIN requires $O(s\beta)$ arithmetic operations and $O(s\beta^3)$ bit operations, since it requires asymptotically no more work than s modular exponentiations.

Error rate of the Miller-Rabin primality test

If MILLER-RABIN returns PRIME, then there is a very slim chance that it has made an error. Unlike PSEUDOPRIME, however, the chance of error does not depend on n; there are no bad inputs for this procedure. Rather, it depends on the size of s and the "luck of the draw" in choosing base values a. Moreover, since each test is more stringent than a simple check of equation (31.40), we can expect on general principles that the error rate should be small for randomly chosen integers n. The following theorem presents a more precise argument.

Theorem 31.38
If n is an odd composite number, then the number of witnesses to the compositeness of n is at least $(n-1)/2$.

Proof The proof shows that the number of nonwitnesses is at most $(n-1)/2$, which implies the theorem.

We start by claiming that any nonwitness must be a member of $\mathbb{Z}_n^*$. Why? Consider any nonwitness a. It must satisfy $a^{n-1} \equiv 1 \pmod{n}$ or, equivalently, $a \cdot a^{n-2} \equiv 1 \pmod{n}$. Thus, the equation $ax \equiv 1 \pmod{n}$ has a solution, namely a^{n-2}. By Corollary 31.21, $\gcd(a, n) \mid 1$, which in turn implies that $\gcd(a, n) = 1$. Therefore, a is a member of $\mathbb{Z}_n^*$; all nonwitnesses belong to $\mathbb{Z}_n^*$.

To complete the proof, we show that not only are all nonwitnesses contained in $\mathbb{Z}_n^*$, they are all contained in a proper subgroup B of $\mathbb{Z}_n^*$ (recall that we say B is a *proper* subgroup of $\mathbb{Z}_n^*$ when B is subgroup of $\mathbb{Z}_n^*$ but B is not equal to $\mathbb{Z}_n^*$). By Corollary 31.16, we then have $|B| \le |\mathbb{Z}_n^*|/2$. Since $|\mathbb{Z}_n^*| \le n-1$, we obtain $|B| \le (n-1)/2$. Therefore, the number of nonwitnesses is at most $(n-1)/2$, so that the number of witnesses must be at least $(n-1)/2$.

We now show how to find a proper subgroup B of $\mathbb{Z}_n^*$ containing all of the nonwitnesses. We break the proof into two cases.

Case 1: There exists an $x \in \mathbb{Z}_n^*$ such that

$$x^{n-1} \not\equiv 1 \pmod{n} .$$

In other words, n is not a Carmichael number. Because, as we noted earlier, Carmichael numbers are extremely rare, case 1 is the main case that arises "in practice" (e.g., when n has been chosen randomly and is being tested for primality).

Let $B = \{b \in \mathbb{Z}_n^* : b^{n-1} \equiv 1 \pmod{n}\}$. Clearly, B is nonempty, since $1 \in B$. Since B is closed under multiplication modulo n, we have that B is a subgroup of $\mathbb{Z}_n^*$ by Theorem 31.14. Note that every nonwitness belongs to B, since a nonwitness a satisfies $a^{n-1} \equiv 1 \pmod{n}$. Since $x \in \mathbb{Z}_n^* - B$, we have that B is a proper subgroup of $\mathbb{Z}_n^*$.

Case 2: For all $x \in \mathbb{Z}_n^*$,

$$x^{n-1} \equiv 1 \pmod{n} . \tag{31.41}$$

In other words, n is a Carmichael number. This case is extremely rare in practice. However, the Miller-Rabin test (unlike a pseudo-primality test) can efficiently determine that Carmichael numbers are composite, as we now show.

In this case, n cannot be a prime power. To see why, let us suppose to the contrary that $n = p^e$, where p is a prime and $e > 1$. We derive a contradiction as follows. Since we assume that n is odd, p must also be odd. Theorem 31.32 implies that $\mathbb{Z}_n^*$ is a cyclic group: it contains a generator g such that $\mathrm{ord}_n(g) = |\mathbb{Z}_n^*| = \phi(n) = p^e(1 - 1/p) = (p-1)p^{e-1}$. (The formula for $\phi(n)$ comes from equation (31.20).) By equation (31.41), we have $g^{n-1} \equiv 1 \pmod{n}$. Then the discrete logarithm theorem (Theorem 31.33, taking $y = 0$) implies that $n - 1 \equiv 0 \pmod{\phi(n)}$, or

$$(p-1)p^{e-1} \mid p^e - 1 .$$

This is a contradiction for $e > 1$, since $(p-1)p^{e-1}$ is divisible by the prime p but $p^e - 1$ is not. Thus, n is not a prime power.

Since the odd composite number n is not a prime power, we decompose it into a product $n_1 n_2$, where n_1 and n_2 are odd numbers greater than 1 that are relatively prime to each other. (There may be several ways to decompose n, and it does not matter which one we choose. For example, if $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we can choose $n_1 = p_1^{e_1}$ and $n_2 = p_2^{e_2} p_3^{e_3} \cdots p_r^{e_r}$.)

Recall that we define t and u so that $n - 1 = 2^t u$, where $t \geq 1$ and u is odd, and that for an input a, the procedure WITNESS computes the sequence

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

(all computations are performed modulo n).

Let us call a pair (v, j) of integers ***acceptable*** if $v \in \mathbb{Z}_n^*$, $j \in \{0, 1, \ldots, t\}$, and

$$v^{2^j u} \equiv -1 \pmod{n} .$$

Acceptable pairs certainly exist since u is odd; we can choose $v = n - 1$ and $j = 0$, so that $(n-1, 0)$ is an acceptable pair. Now pick the largest possible j such that there exists an acceptable pair (v, j), and fix v so that (v, j) is an acceptable pair. Let

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} \equiv \pm 1 \pmod{n}\} .$$

Since B is closed under multiplication modulo n, it is a subgroup of $\mathbb{Z}_n^*$. By Theorem 31.15, therefore, $|B|$ divides $|\mathbb{Z}_n^*|$. Every nonwitness must be a member of B, since the sequence X produced by a nonwitness must either be all 1s or else contain a -1 no later than the jth position, by the maximality of j. (If (a, j') is acceptable, where a is a nonwitness, we must have $j' \le j$ by how we chose j.)

We now use the existence of v to demonstrate that there exists a $w \in \mathbb{Z}_n^* - B$, and hence that B is a proper subgroup of $\mathbb{Z}_n^*$. Since $v^{2^j u} \equiv -1 \pmod{n}$, we have $v^{2^j u} \equiv -1 \pmod{n_1}$ by Corollary 31.29 to the Chinese remainder theorem. By Corollary 31.28, there exists a w simultaneously satisfying the equations

$$\begin{aligned} w &\equiv v \pmod{n_1} , \\ w &\equiv 1 \pmod{n_2} . \end{aligned}$$

Therefore,

$$\begin{aligned} w^{2^j u} &\equiv -1 \pmod{n_1} , \\ w^{2^j u} &\equiv 1 \pmod{n_2} . \end{aligned}$$

By Corollary 31.29, $w^{2^j u} \not\equiv 1 \pmod{n_1}$ implies $w^{2^j u} \not\equiv 1 \pmod{n}$, and $w^{2^j u} \not\equiv -1 \pmod{n_2}$ implies $w^{2^j u} \not\equiv -1 \pmod{n}$. Hence, we conclude that $w^{2^j u} \not\equiv \pm 1 \pmod{n}$, and so $w \notin B$.

It remains to show that $w \in \mathbb{Z}_n^*$, which we do by first working separately modulo n_1 and modulo n_2. Working modulo n_1, we observe that since $v \in \mathbb{Z}_n^*$, we have that $\gcd(v, n) = 1$, and so also $\gcd(v, n_1) = 1$; if v does not have any common divisors with n, then it certainly does not have any common divisors with n_1. Since $w \equiv v \pmod{n_1}$, we see that $\gcd(w, n_1) = 1$. Working modulo n_2, we observe that $w \equiv 1 \pmod{n_2}$ implies $\gcd(w, n_2) = 1$. To combine these results, we use Theorem 31.6, which implies that $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. That is, $w \in \mathbb{Z}_n^*$.

Therefore $w \in \mathbb{Z}_n^* - B$, and we finish case 2 with the conclusion that B is a proper subgroup of $\mathbb{Z}_n^*$.

In either case, we see that the number of witnesses to the compositeness of n is at least $(n-1)/2$. ■

Theorem 31.39

For any odd integer $n > 2$ and positive integer s, the probability that MILLER-RABIN(n, s) errs is at most 2^{-s}.

Proof Using Theorem 31.38, we see that if n is composite, then each execution of the **for** loop of lines 1–4 has a probability of at least $1/2$ of discovering a witness x to the compositeness of n. MILLER-RABIN makes an error only if it is so unlucky as to miss discovering a witness to the compositeness of n on each of the s iterations of the main loop. The probability of such a sequence of misses is at most 2^{-s}. ■

If n is prime, MILLER-RABIN always reports PRIME, and if n is composite, the chance that MILLER-RABIN reports PRIME is at most 2^{-s}.

When applying MILLER-RABIN to a large randomly chosen integer n, however, we need to consider as well the prior probability that n is prime, in order to correctly interpret MILLER-RABIN's result. Suppose that we fix a bit length β and choose at random an integer n of length β bits to be tested for primality. Let A denote the event that n is prime. By the prime number theorem (Theorem 31.37), the probability that n is prime is approximately

$$\begin{aligned} \Pr\{A\} &\approx 1/\ln n \\ &\approx 1.443/\beta \ . \end{aligned}$$

Now let B denote the event that MILLER-RABIN returns PRIME. We have that $\Pr\left\{\overline{B} \mid A\right\} = 0$ (or equivalently, that $\Pr\{B \mid A\} = 1$) and $\Pr\left\{B \mid \overline{A}\right\} \leq 2^{-s}$ (or equivalently, that $\Pr\left\{\overline{B} \mid \overline{A}\right\} > 1 - 2^{-s}$).

But what is $\Pr\{A \mid B\}$, the probability that n is prime, given that MILLER-RABIN has returned PRIME? By the alternate form of Bayes's theorem (equation (C.18)) we have

$$\begin{aligned} \Pr\{A \mid B\} &= \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B \mid \overline{A}\right\}} \\ &\approx \frac{1}{1 + 2^{-s}(\ln n - 1)} \ . \end{aligned}$$

This probability does not exceed $1/2$ until s exceeds $\lg(\ln n - 1)$. Intuitively, that many initial trials are needed just for the confidence derived from failing to find a witness to the compositeness of n to overcome the prior bias in favor of n being composite. For a number with $\beta = 1024$ bits, this initial testing requires about

$$\begin{aligned} \lg(\ln n - 1) &\approx \lg(\beta/1.443) \\ &\approx 9 \end{aligned}$$

trials. In any case, choosing $s = 50$ should suffice for almost any imaginable application.

In fact, the situation is much better. If we are trying to find large primes by applying MILLER-RABIN to large randomly chosen odd integers, then choosing a small value of s (say 3) is very unlikely to lead to erroneous results, though

we won't prove it here. The reason is that for a randomly chosen odd composite integer n, the expected number of nonwitnesses to the compositeness of n is likely to be very much smaller than $(n-1)/2$.

If the integer n is not chosen randomly, however, the best that can be proven is that the number of nonwitnesses is at most $(n-1)/4$, using an improved version of Theorem 31.38. Furthermore, there do exist integers n for which the number of nonwitnesses is $(n-1)/4$.

Exercises

31.8-1
Prove that if an odd integer $n > 1$ is not a prime or a prime power, then there exists a nontrivial square root of 1 modulo n.

31.8-2 ⋆
It is possible to strengthen Euler's theorem slightly to the form

$$a^{\lambda(n)} \equiv 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^* ,$$

where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by

$$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) . \tag{31.42}$$

Prove that $\lambda(n) \mid \phi(n)$. A composite number n is a Carmichael number if $\lambda(n) \mid n-1$. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$; here, $\lambda(n) = \text{lcm}(2, 10, 16) = 80$, which divides 560. Prove that Carmichael numbers must be both "square-free" (not divisible by the square of any prime) and the product of at least three primes. (For this reason, they are not very common.)

31.8-3
Prove that if x is a nontrivial square root of 1, modulo n, then $\gcd(x-1, n)$ and $\gcd(x+1, n)$ are both nontrivial divisors of n.

⋆ 31.9 Integer factorization

Suppose we have an integer n that we wish to ***factor***, that is, to decompose into a product of primes. The primality test of the preceding section may tell us that n is composite, but it does not tell us the prime factors of n. Factoring a large integer n seems to be much more difficult than simply determining whether n is prime or composite. Even with today's supercomputers and the best algorithms to date, we cannot feasibly factor an arbitrary 1024-bit number.

Pollard's rho heuristic

Trial division by all integers up to R is guaranteed to factor completely any number up to R^2. For the same amount of work, the following procedure, POLLARD-RHO, factors any number up to R^4 (unless we are unlucky). Since the procedure is only a heuristic, neither its running time nor its success is guaranteed, although the procedure is highly effective in practice. Another advantage of the POLLARD-RHO procedure is that it uses only a constant number of memory locations. (If you wanted to, you could easily implement POLLARD-RHO on a programmable pocket calculator to find factors of small numbers.)

```
POLLARD-RHO(n)
 1  i = 1
 2  x_1 = RANDOM(0, n − 1)
 3  y = x_1
 4  k = 2
 5  while TRUE
 6      i = i + 1
 7      x_i = (x_{i−1}^2 − 1) mod n
 8      d = gcd(y − x_i, n)
 9      if d ≠ 1 and d ≠ n
10          print d
11      if i == k
12          y = x_i
13          k = 2k
```

The procedure works as follows. Lines 1–2 initialize i to 1 and x_1 to a randomly chosen value in $\mathbb{Z}_n$. The **while** loop beginning on line 5 iterates forever, searching for factors of n. During each iteration of the **while** loop, line 7 uses the recurrence

$$x_i = (x_{i-1}^2 - 1) \bmod n \tag{31.43}$$

to produce the next value of x_i in the infinite sequence

$$x_1, x_2, x_3, x_4, \ldots , \tag{31.44}$$

with line 6 correspondingly incrementing i. The pseudocode is written using subscripted variables x_i for clarity, but the program works the same if all of the subscripts are dropped, since only the most recent value of x_i needs to be maintained. With this modification, the procedure uses only a constant number of memory locations.

Every so often, the program saves the most recently generated x_i value in the variable y. Specifically, the values that are saved are the ones whose subscripts are powers of 2:

$x_1, x_2, x_4, x_8, x_{16}, \ldots$.

Line 3 saves the value x_1, and line 12 saves x_k whenever i is equal to k. The variable k is initialized to 2 in line 4, and line 13 doubles it whenever line 12 updates y. Therefore, k follows the sequence $1, 2, 4, 8, \ldots$ and always gives the subscript of the next value x_k to be saved in y.

Lines 8–10 try to find a factor of n, using the saved value of y and the current value of x_i. Specifically, line 8 computes the greatest common divisor $d = \gcd(y - x_i, n)$. If line 9 finds d to be a nontrivial divisor of n, then line 10 prints d.

This procedure for finding a factor may seem somewhat mysterious at first. Note, however, that POLLARD-RHO never prints an incorrect answer; any number it prints is a nontrivial divisor of n. POLLARD-RHO might not print anything at all, though; it comes with no guarantee that it will print any divisors. We shall see, however, that we have good reason to expect POLLARD-RHO to print a factor p of n after $\Theta(\sqrt{p})$ iterations of the **while** loop. Thus, if n is composite, we can expect this procedure to discover enough divisors to factor n completely after approximately $n^{1/4}$ updates, since every prime factor p of n except possibly the largest one is less than $\sqrt{n}$.

We begin our analysis of how this procedure behaves by studying how long it takes a random sequence modulo n to repeat a value. Since $\mathbb{Z}_n$ is finite, and since each value in the sequence (31.44) depends only on the previous value, the sequence (31.44) eventually repeats itself. Once we reach an x_i such that $x_i = x_j$ for some $j < i$, we are in a cycle, since $x_{i+1} = x_{j+1}$, $x_{i+2} = x_{j+2}$, and so on. The reason for the name "rho heuristic" is that, as Figure 31.7 shows, we can draw the sequence $x_1, x_2, \ldots, x_{j-1}$ as the "tail" of the rho and the cycle $x_j, x_{j+1}, \ldots, x_i$ as the "body" of the rho.

Let us consider the question of how long it takes for the sequence of x_i to repeat. This information is not exactly what we need, but we shall see later how to modify the argument. For the purpose of this estimation, let us assume that the function

$$f_n(x) = (x^2 - 1) \bmod n$$

behaves like a "random" function. Of course, it is not really random, but this assumption yields results consistent with the observed behavior of POLLARD-RHO. We can then consider each x_i to have been independently drawn from $\mathbb{Z}_n$ according to a uniform distribution on $\mathbb{Z}_n$. By the birthday-paradox analysis of Section 5.4.1, we expect $\Theta(\sqrt{n})$ steps to be taken before the sequence cycles.

Now for the required modification. Let p be a nontrivial factor of n such that $\gcd(p, n/p) = 1$. For example, if n has the factorization $n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$, then we may take p to be $p_1^{e_1}$. (If $e_1 = 1$, then p is just the smallest prime factor of n, a good example to keep in mind.)

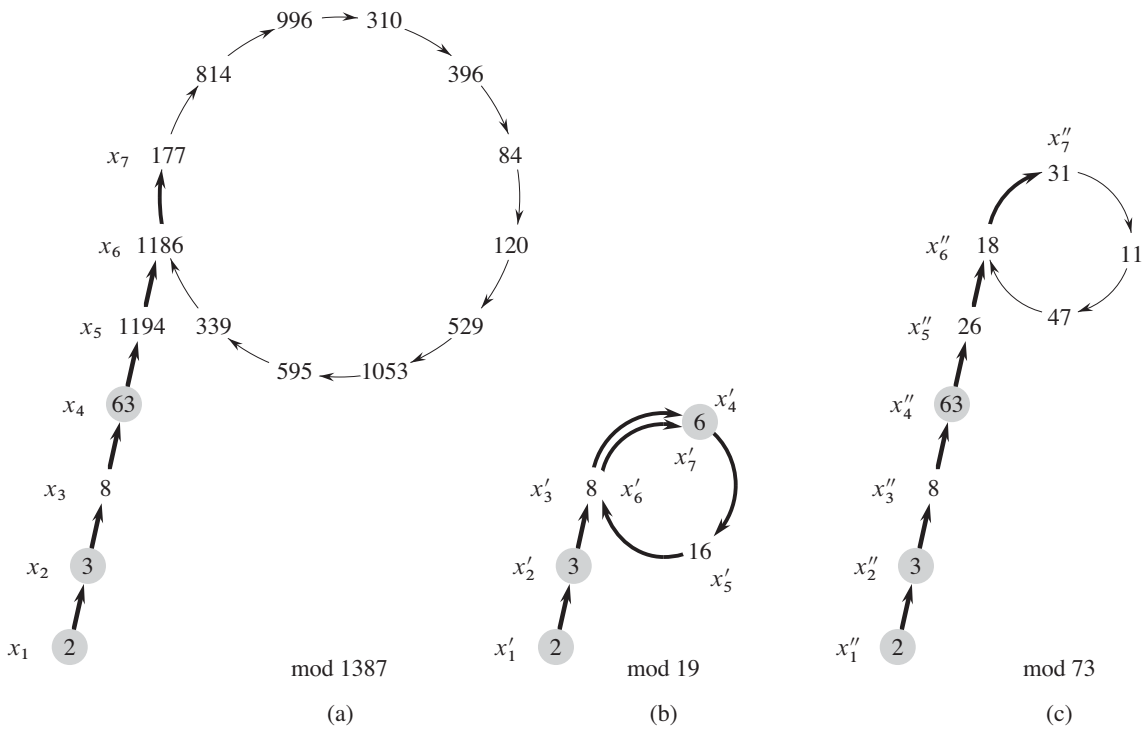


Figure 31.7 Pollard's rho heuristic. **(a)** The values produced by the recurrence $x_{i+1} = (x_i^2 - 1) \bmod 1387$, starting with $x_1 = 2$. The prime factorization of 1387 is $19 \cdot 73$. The heavy arrows indicate the iteration steps that are executed before the factor 19 is discovered. The light arrows point to unreached values in the iteration, to illustrate the "rho" shape. The shaded values are the y values stored by POLLARD-RHO. The factor 19 is discovered upon reaching $x_7 = 177$, when $\gcd(63 - 177, 1387) = 19$ is computed. The first x value that would be repeated is 1186, but the factor 19 is discovered before this value is repeated. **(b)** The values produced by the same recurrence, modulo 19. Every value x_i given in part (a) is equivalent, modulo 19, to the value x'_i shown here. For example, both $x_4 = 63$ and $x_7 = 177$ are equivalent to 6, modulo 19. **(c)** The values produced by the same recurrence, modulo 73. Every value x_i given in part (a) is equivalent, modulo 73, to the value x''_i shown here. By the Chinese remainder theorem, each node in part (a) corresponds to a pair of nodes, one from part (b) and one from part (c).

The sequence $\langle x_i \rangle$ induces a corresponding sequence $\langle x'_i \rangle$ modulo p, where

$$x'_i = x_i \bmod p$$

for all i.

Furthermore, because f_n is defined using only arithmetic operations (squaring and subtraction) modulo n, we can compute x'_{i+1} from x'_i; the "modulo p" view of

the sequence is a smaller version of what is happening modulo n:

$$\begin{aligned} x'_{i+1} &= x_{i+1} \bmod p \\ &= f_n(x_i) \bmod p \\ &= ((x_i^2 - 1) \bmod n) \bmod p \\ &= (x_i^2 - 1) \bmod p \qquad \text{(by Exercise 31.1-7)} \\ &= ((x_i \bmod p)^2 - 1) \bmod p \\ &= ((x'_i)^2 - 1) \bmod p \\ &= f_p(x'_i) \ . \end{aligned}$$

Thus, although we are not explicitly computing the sequence $\langle x'_i \rangle$, this sequence is well defined and obeys the same recurrence as the sequence $\langle x_i \rangle$.

Reasoning as before, we find that the expected number of steps before the sequence $\langle x'_i \rangle$ repeats is $\Theta(\sqrt{p})$. If p is small compared to n, the sequence $\langle x'_i \rangle$ might repeat much more quickly than the sequence $\langle x_i \rangle$. Indeed, as parts (b) and (c) of Figure 31.7 show, the $\langle x'_i \rangle$ sequence repeats as soon as two elements of the sequence $\langle x_i \rangle$ are merely equivalent modulo p, rather than equivalent modulo n.

Let t denote the index of the first repeated value in the $\langle x'_i \rangle$ sequence, and let $u > 0$ denote the length of the cycle that has been thereby produced. That is, t and $u > 0$ are the smallest values such that $x'_{t+i} = x'_{t+u+i}$ for all $i \geq 0$. By the above arguments, the expected values of t and u are both $\Theta(\sqrt{p})$. Note that if $x'_{t+i} = x'_{t+u+i}$, then $p \mid (x_{t+u+i} - x_{t+i})$. Thus, $\gcd(x_{t+u+i} - x_{t+i}, n) > 1$.

Therefore, once POLLARD-RHO has saved as y any value x_k such that $k \geq t$, then $y \bmod p$ is always on the cycle modulo p. (If a new value is saved as y, that value is also on the cycle modulo p.) Eventually, k is set to a value that is greater than u, and the procedure then makes an entire loop around the cycle modulo p without changing the value of y. The procedure then discovers a factor of n when x_i "runs into" the previously stored value of y, modulo p, that is, when $x_i \equiv y \pmod{p}$.

Presumably, the factor found is the factor p, although it may occasionally happen that a multiple of p is discovered. Since the expected values of both t and u are $\Theta(\sqrt{p})$, the expected number of steps required to produce the factor p is $\Theta(\sqrt{p})$.

This algorithm might not perform quite as expected, for two reasons. First, the heuristic analysis of the running time is not rigorous, and it is possible that the cycle of values, modulo p, could be much larger than $\sqrt{p}$. In this case, the algorithm performs correctly but much more slowly than desired. In practice, this issue seems to be moot. Second, the divisors of n produced by this algorithm might always be one of the trivial factors 1 or n. For example, suppose that $n = pq$, where p and q are prime. It can happen that the values of t and u for p are identical with the values of t and u for q, and thus the factor p is always revealed in the same gcd operation that reveals the factor q. Since both factors are revealed at the same

time, the trivial factor $pq = n$ is revealed, which is useless. Again, this problem seems to be insignificant in practice. If necessary, we can restart the heuristic with a different recurrence of the form $x_{i+1} = (x_i^2 - c) \bmod n$. (We should avoid the values $c = 0$ and $c = 2$ for reasons we will not go into here, but other values are fine.)

Of course, this analysis is heuristic and not rigorous, since the recurrence is not really "random." Nonetheless, the procedure performs well in practice, and it seems to be as efficient as this heuristic analysis indicates. It is the method of choice for finding small prime factors of a large number. To factor a β-bit composite number n completely, we only need to find all prime factors less than $\lfloor n^{1/2} \rfloor$, and so we expect POLLARD-RHO to require at most $n^{1/4} = 2^{\beta/4}$ arithmetic operations and at most $n^{1/4}\beta^2 = 2^{\beta/4}\beta^2$ bit operations. POLLARD-RHO's ability to find a small factor p of n with an expected number $\Theta(\sqrt{p})$ of arithmetic operations is often its most appealing feature.

Exercises

31.9-1

Referring to the execution history shown in Figure 31.7(a), when does POLLARD-RHO print the factor 73 of 1387?

31.9-2

Suppose that we are given a function $f : \mathbb{Z}_n \to \mathbb{Z}_n$ and an initial value $x_0 \in \mathbb{Z}_n$. Define $x_i = f(x_{i-1})$ for $i = 1, 2, \ldots$. Let t and $u > 0$ be the smallest values such that $x_{t+i} = x_{t+u+i}$ for $i = 0, 1, \ldots$. In the terminology of Pollard's rho algorithm, t is the length of the tail and u is the length of the cycle of the rho. Give an efficient algorithm to determine t and u exactly, and analyze its running time.

31.9-3

How many steps would you expect POLLARD-RHO to require to discover a factor of the form p^e, where p is prime and $e > 1$?

31.9-4 ★

One disadvantage of POLLARD-RHO as written is that it requires one gcd computation for each step of the recurrence. Instead, we could batch the gcd computations by accumulating the product of several x_i values in a row and then using this product instead of x_i in the gcd computation. Describe carefully how you would implement this idea, why it works, and what batch size you would pick as the most effective when working on a β-bit number n.

Problems

31-1 Binary gcd algorithm

Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the ***binary gcd algorithm***, which avoids the remainder computations used in Euclid's algorithm.

a. Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.

b. Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.

c. Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.

d. Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\lg a)$ time. Assume that each subtraction, parity test, and halving takes unit time.

31-2 Analysis of bit operations in Euclid's algorithm

a. Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b, which yields a quotient q and remainder r. Show that this method requires $O((1 + \lg q) \lg b)$ bit operations.

b. Define $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Show that the number of bit operations performed by EUCLID in reducing the problem of computing $\gcd(a, b)$ to that of computing $\gcd(b, a \bmod b)$ is at most $c(\mu(a, b) - \mu(b, a \bmod b))$ for some sufficiently large constant $c > 0$.

c. Show that EUCLID(a, b) requires $O(\mu(a, b))$ bit operations in general and $O(\beta^2)$ bit operations when applied to two β-bit inputs.

31-3 Three algorithms for Fibonacci numbers

This problem compares the efficiency of three methods for computing the nth Fibonacci number F_n, given n. Assume that the cost of adding, subtracting, or multiplying two numbers is $O(1)$, independent of the size of the numbers.

a. Show that the running time of the straightforward recursive method for computing F_n based on recurrence (3.22) is exponential in n. (See, for example, the FIB procedure on page 775.)

b. Show how to compute F_n in $O(n)$ time using memoization.

c. Show how to compute F_n in $O(\lg n)$ time using only integer addition and multiplication. (*Hint:* Consider the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

and its powers.)

d. Assume now that adding two β-bit numbers takes $\Theta(\beta)$ time and that multiplying two β-bit numbers takes $\Theta(\beta^2)$ time. What is the running time of these three methods under this more reasonable cost measure for the elementary arithmetic operations?

31-4 Quadratic residues

Let p be an odd prime. A number $a \in Z_p^*$ is a ***quadratic residue*** if the equation $x^2 = a \pmod{p}$ has a solution for the unknown x.

a. Show that there are exactly $(p-1)/2$ quadratic residues, modulo p.

b. If p is prime, we define the ***Legendre symbol*** $\left(\frac{a}{p}\right)$, for $a \in \mathbb{Z}_p^*$, to be 1 if a is a quadratic residue modulo p and -1 otherwise. Prove that if $a \in \mathbb{Z}_p^*$, then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p} .$$

Give an efficient algorithm that determines whether a given number a is a quadratic residue modulo p. Analyze the efficiency of your algorithm.

c. Prove that if p is a prime of the form $4k+3$ and a is a quadratic residue in $\mathbb{Z}_p^*$, then $a^{k+1} \bmod p$ is a square root of a, modulo p. How much time is required to find the square root of a quadratic residue a modulo p?

d. Describe an efficient randomized algorithm for finding a nonquadratic residue, modulo an arbitrary prime p, that is, a member of $\mathbb{Z}_p^*$ that is not a quadratic residue. How many arithmetic operations does your algorithm require on average?

Chapter notes

Niven and Zuckerman [265] provide an excellent introduction to elementary number theory. Knuth [210] contains a good discussion of algorithms for finding the

greatest common divisor, as well as other basic number-theoretic algorithms. Bach [30] and Riesel [295] provide more recent surveys of computational number theory. Dixon [91] gives an overview of factorization and primality testing. The conference proceedings edited by Pomerance [280] contains several excellent survey articles. More recently, Bach and Shallit [31] have provided an exceptional overview of the basics of computational number theory.

Knuth [210] discusses the origin of Euclid's algorithm. It appears in Book 7, Propositions 1 and 2, of the Greek mathematician Euclid's *Elements*, which was written around 300 B.C. Euclid's description may have been derived from an algorithm due to Eudoxus around 375 B.C. Euclid's algorithm may hold the honor of being the oldest nontrivial algorithm; it is rivaled only by an algorithm for multiplication known to the ancient Egyptians. Shallit [312] chronicles the history of the analysis of Euclid's algorithm.

Knuth attributes a special case of the Chinese remainder theorem (Theorem 31.27) to the Chinese mathematician Sun-Tsŭ, who lived sometime between 200 B.C. and A.D. 200—the date is quite uncertain. The same special case was given by the Greek mathematician Nichomachus around A.D. 100. It was generalized by Chhin Chiu-Shao in 1247. The Chinese remainder theorem was finally stated and proved in its full generality by L. Euler in 1734.

The randomized primality-testing algorithm presented here is due to Miller [255] and Rabin [289]; it is the fastest randomized primality-testing algorithm known, to within constant factors. The proof of Theorem 31.39 is a slight adaptation of one suggested by Bach [29]. A proof of a stronger result for MILLER-RABIN was given by Monier [258, 259]. For many years primality-testing was the classic example of a problem where randomization appeared to be necessary to obtain an efficient (polynomial-time) algorithm. In 2002, however, Agrawal, Kayal, and Saxema [4] surprised everyone with their deterministic polynomial-time primality-testing algorithm. Until then, the fastest deterministic primality testing algorithm known, due to Cohen and Lenstra [73], ran in time $(\lg n)^{O(\lg\lg\lg n)}$ on input n, which is just slightly superpolynomial. Nonetheless, for practical purposes randomized primality-testing algorithms remain more efficient and are preferred.

The problem of finding large "random" primes is nicely discussed in an article by Beauchemin, Brassard, Crépeau, Goutier, and Pomerance [36].

The concept of a public-key cryptosystem is due to Diffie and Hellman [87]. The RSA cryptosystem was proposed in 1977 by Rivest, Shamir, and Adleman [296]. Since then, the field of cryptography has blossomed. Our understanding of the RSA cryptosystem has deepened, and modern implementations use significant refinements of the basic techniques presented here. In addition, many new techniques have been developed for proving cryptosystems to be secure. For example, Goldwasser and Micali [142] show that randomization can be an effective tool in the design of secure public-key encryption schemes. For signature schemes,

Goldwasser, Micali, and Rivest [143] present a digital-signature scheme for which every conceivable type of forgery is provably as difficult as factoring. Menezes, van Oorschot, and Vanstone [254] provide an overview of applied cryptography.

The rho heuristic for integer factorization was invented by Pollard [277]. The version presented here is a variant proposed by Brent [56].

The best algorithms for factoring large numbers have a running time that grows roughly exponentially with the cube root of the length of the number n to be factored. The general number-field sieve factoring algorithm (as developed by Buhler, Lenstra, and Pomerance [57] as an extension of the ideas in the number-field sieve factoring algorithm by Pollard [278] and Lenstra et al. [232] and refined by Coppersmith [77] and others) is perhaps the most efficient such algorithm in general for large inputs. Although it is difficult to give a rigorous analysis of this algorithm, under reasonable assumptions we can derive a running-time estimate of $L(1/3, n)^{1.902+o(1)}$, where $L(\alpha, n) = e^{(\ln n)^{\alpha}(\ln\ln n)^{1-\alpha}}$.

The elliptic-curve method due to Lenstra [233] may be more effective for some inputs than the number-field sieve method, since, like Pollard's rho method, it can find a small prime factor p quite quickly. With this method, the time to find p is estimated to be $L(1/2, p)^{\sqrt{2}+o(1)}$.

32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem—called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

We formalize the string-matching problem as follows. We assume that the text is an array $T[1..n]$ of length n and that the pattern is an array $P[1..m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often called ***strings*** of characters.

Referring to Figure 32.1, we say that pattern P ***occurs with shift s*** in text T (or, equivalently, that pattern P ***occurs beginning at position s + 1*** in text T) if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$ (that is, if $T[s + j] = P[j]$, for $1 \leq j \leq m$). If P occurs with shift s in T, then we call s a ***valid shift***; otherwise, we call s an ***invalid shift***. The ***string-matching problem*** is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

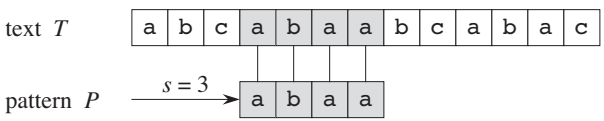


Figure 32.1 An example of the string-matching problem, where we want to find all occurrences of the pattern $P = \texttt{abaa}$ in the text $T = \texttt{abcabaabcabac}$. The pattern occurs only once in the text, at shift $s = 3$, which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 32.2 The string-matching algorithms in this chapter and their preprocessing and matching times.

Except for the naive brute-force algorithm, which we review in Section 32.1, each string-matching algorithm in this chapter performs some preprocessing based on the pattern and then finds all valid shifts; we call this latter phase "matching." Figure 32.2 shows the preprocessing and matching times for each of the algorithms in this chapter. The total running time of each algorithm is the sum of the preprocessing and matching times. Section 32.2 presents an interesting string-matching algorithm, due to Rabin and Karp. Although the $\Theta((n-m+1)m)$ worst-case running time of this algorithm is no better than that of the naive method, it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems. Section 32.3 then describes a string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern P in a text. This algorithm takes $O(m|\Sigma|)$ preprocessing time, but only $\Theta(n)$ matching time. Section 32.4 presents the similar, but much cleverer, Knuth-Morris-Pratt (or KMP) algorithm; it has the same $\Theta(n)$ matching time, and it reduces the preprocessing time to only $\Theta(m)$.

Notation and terminology

We denote by Σ^* (read "sigma-star") the set of all finite-length strings formed using characters from the alphabet Σ. In this chapter, we consider only finite-length strings. The zero-length ***empty string***, denoted ε, also belongs to Σ^*. The length of a string x is denoted $|x|$. The ***concatenation*** of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.

We say that a string w is a ***prefix*** of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubset x$, then $|w| \leq |x|$. Similarly, we say that a string w is a ***suffix*** of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. As with a prefix, $w \sqsupset x$ implies $|w| \leq |x|$. For example, we have $\texttt{ab} \sqsubset \texttt{abcca}$ and $\texttt{cca} \sqsupset \texttt{abcca}$. The empty string ε is both a suffix and a prefix of every string. For any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$.

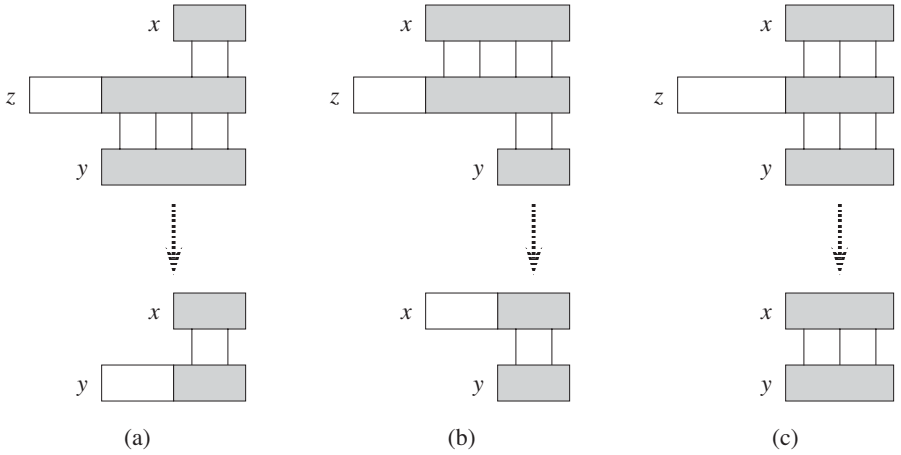


Figure 32.3 A graphical proof of Lemma 32.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. **(a)** If $|x| \leq |y|$, then $x \sqsupset y$. **(b)** If $|x| \geq |y|$, then $y \sqsupset x$. **(c)** If $|x| = |y|$, then $x = y$.

Also note that $\sqsubset$ and $\sqsupset$ are transitive relations. The following lemma will be useful later.

Lemma 32.1 (Overlapping-suffix lemma)
Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof See Figure 32.3 for a graphical proof. ■

For brevity of notation, we denote the k-character prefix $P[1\,..\,k]$ of the pattern $P[1\,..\,m]$ by P_k. Thus, $P_0 = \varepsilon$ and $P_m = P = P[1\,..\,m]$. Similarly, we denote the k-character prefix of the text T by T_k. Using this notation, we can state the string-matching problem as that of finding all shifts s in the range $0 \leq s \leq n - m$ such that $P \sqsupset T_{s+m}$.

In our pseudocode, we allow two equal-length strings to be compared for equality as a primitive operation. If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered. To be precise, the test "x == y" is assumed to take time $\Theta(t + 1)$, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$. (We write $\Theta(t + 1)$ rather than $\Theta(t)$ to handle the case in which $t = 0$; the first characters compared do not match, but it takes a positive amount of time to perform this comparison.)

32.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1\,..\,m] = T[s+1\,..\,s+m]$ for each of the $n-m+1$ possible values of s.

```
NAIVE-STRING-MATCHER(T, P)
1  n = T.length
2  m = P.length
3  for s = 0 to n − m
4      if P[1..m] == T[s + 1..s + m]
5          print "Pattern occurs with shift" s
```

Figure 32.4 portrays the naive string-matching procedure as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop of lines 3–5 considers each possible shift explicitly. The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

Procedure NAIVE-STRING-MATCHER takes time $O((n-m+1)m)$, and this bound is tight in the worst case. For example, consider the text string $\mathtt{a}^n$ (a string of n a's) and the pattern $\mathtt{a}^m$. For each of the $n-m+1$ possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus $\Theta((n-m+1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. Because it requires no preprocessing, NAIVE-STRING-MATCHER's running time equals its matching time.

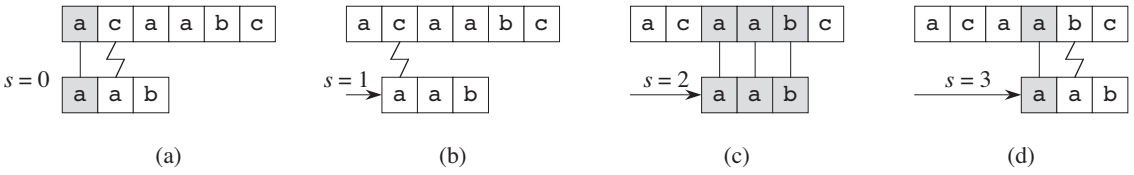


Figure 32.4 The operation of the naive string matcher for the pattern $P = \mathtt{aab}$ and the text $T = \mathtt{acaabc}$. We can imagine the pattern P as a template that we slide next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c).

As we shall see, NAIVE-STRING-MATCHER is not an optimal procedure for this problem. Indeed, in this chapter we shall see that the Knuth-Morris-Pratt algorithm is much better in the worst case. The naive string-matcher is inefficient because it entirely ignores information gained about the text for one value of s when it considers other values of s. Such information can be quite valuable, however. For example, if $P = \texttt{aaab}$ and we find that $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $T[4] = \texttt{b}$. In the following sections, we examine several ways to make effective use of this sort of information.

Exercises

32.1-1

Show the comparisons the naive string matcher makes for the pattern $P = \texttt{0001}$ in the text $T = \texttt{000010001010001}$.

32.1-2

Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n-character text T.

32.1-3

Suppose that pattern P and text T are *randomly* chosen strings of length m and n, respectively, from the d-ary alphabet $\Sigma_d = \{0, 1, \ldots, d-1\}$, where $d \geq 2$. Show that the *expected* number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$$

over all executions of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once it finds a mismatch or matches the entire pattern.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

32.1-4

Suppose we allow the pattern P to contain occurrences of a ***gap character*** $\diamond$ that can match an *arbitrary* string of characters (even one of zero length). For example, the pattern $\texttt{ab}\diamond\texttt{ba}\diamond\texttt{c}$ occurs in the text $\texttt{cabccbacbacab}$ as

$$\begin{array}{l} \texttt{c}\,\underbrace{\texttt{ab}}\,\underbrace{\texttt{cc}}\,\underbrace{\texttt{ba}}\,\underbrace{\texttt{cba}}\,\underbrace{\texttt{c}}\,\texttt{ab} \\ \quad\texttt{ab}\ \diamond\ \texttt{ba}\ \diamond\ \texttt{c} \end{array}$$

and as

$$\begin{array}{l} \texttt{c}\,\underbrace{\texttt{ab}}\,\underbrace{\texttt{ccbac}}\,\underbrace{\texttt{ba}}\,\underbrace{}\,\underbrace{\texttt{c}}\,\texttt{ab}\,. \\ \quad\texttt{ab}\ \ \diamond\ \ \texttt{ba}\ \diamond\ \texttt{c} \end{array}$$

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

32.2 The Rabin-Karp algorithm

Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$. Based on certain assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. You might want to refer to Section 31.1 for the relevant definitions.

For expository purposes, let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-d notation, where $d = |\Sigma|$.) We can then view a string of k consecutive characters as representing a length-k decimal number. The character string 31415 thus corresponds to the decimal number 31,415. Because we interpret the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

Given a pattern $P[1..m]$, let p denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, let t_s denote the decimal value of the length-m substring $T[s+1..s+m]$, for $s = 0, 1, \ldots, n-m$. Certainly, $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$; thus, s is a valid shift if and only if $t_s = p$. If we could compute p in time $\Theta(m)$ and all the t_s values in a total of $\Theta(n-m+1)$ time,[1] then we could determine all valid shifts s in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing p with each of the t_s values. (For the moment, let's not worry about the possibility that p and the t_s values might be very large numbers.)

We can compute p in time $\Theta(m)$ using Horner's rule (see Section 30.1):

$$p = P[m] + 10\,(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))\,.$$

Similarly, we can compute t_0 from $T[1..m]$ in time $\Theta(m)$.

[1] We write $\Theta(n-m+1)$ instead of $\Theta(n-m)$ because s takes on $n-m+1$ different values. The "+1" is significant in an asymptotic sense because when $m = n$, computing the lone t_s value takes $\Theta(1)$ time, not $\Theta(0)$ time.

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time $\Theta(n-m)$, we observe that we can compute t_{s+1} from t_s in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \,. \tag{32.1}$$

Subtracting $10^{m-1}T[s+1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left by one digit position, and adding $T[s+m+1]$ brings in the appropriate low-order digit. For example, if $m = 5$ and $t_s = 31415$, then we wish to remove the high-order digit $T[s+1] = 3$ and bring in the new low-order digit (suppose it is $T[s+5+1] = 2$) to obtain

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \,. \end{aligned}$$

If we precompute the constant 10^{m-1} (which we can do in time $O(\lg m)$ using the techniques of Section 31.6, although for this application a straightforward $O(m)$-time method suffices), then each execution of equation (32.1) takes a constant number of arithmetic operations. Thus, we can compute p in time $\Theta(m)$, and we can compute all of $t_0, t_1, \ldots, t_{n-m}$ in time $\Theta(n-m+1)$. Therefore, we can find all occurrences of the pattern $P[1\,..\,m]$ in the text $T[1\,..\,n]$ with $\Theta(m)$ preprocessing time and $\Theta(n-m+1)$ matching time.

Until now, we have intentionally overlooked one problem: p and t_s may be too large to work with conveniently. If P contains m characters, then we cannot reasonably assume that each arithmetic operation on p (which is m digits long) takes "constant time." Fortunately, we can solve this problem easily, as Figure 32.5 shows: compute p and the t_s values modulo a suitable modulus q. We can compute p modulo q in $\Theta(m)$ time and all the t_s values modulo q in $\Theta(n-m+1)$ time. If we choose the modulus q as a prime such that $10q$ just fits within one computer word, then we can perform all the necessary computations with single-precision arithmetic. In general, with a d-ary alphabet $\{0, 1, \ldots, d-1\}$, we choose q so that dq fits within a computer word and adjust the recurrence equation (32.1) to work modulo q, so that it becomes

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \,, \tag{32.2}$$

where $h \equiv d^{m-1} \pmod q$ is the value of the digit "1" in the high-order position of an m-digit text window.

The solution of working modulo q is not perfect, however: $t_s \equiv p \pmod q$ does not imply that $t_s = p$. On the other hand, if $t_s \not\equiv p \pmod q$, then we definitely have that $t_s \neq p$, so that shift s is invalid. We can thus use the test $t_s \equiv p \pmod q$ as a fast heuristic test to rule out invalid shifts s. Any shift s for which $t_s \equiv p \pmod q$ must be tested further to see whether s is really valid or we just have a ***spurious hit***. This additional test explicitly checks the condition

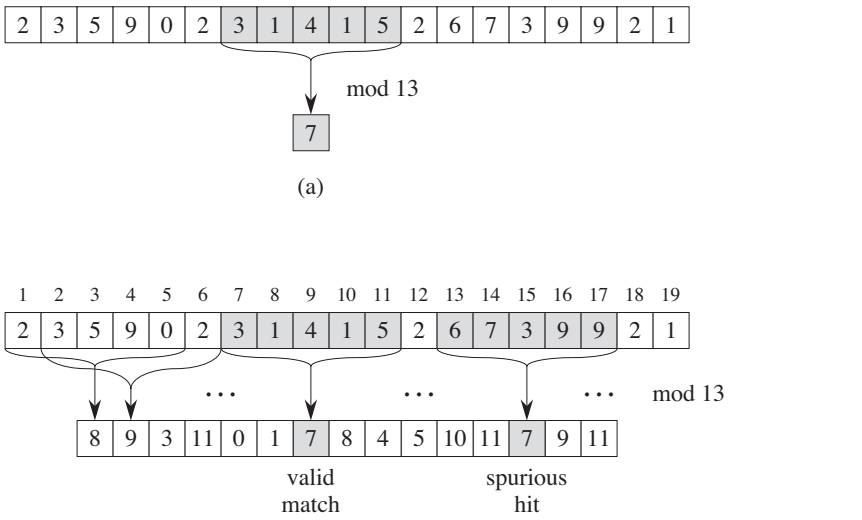


Figure 32.5 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. **(a)** A text string. A window of length 5 is shown shaded. The numerical value of the shaded number, computed modulo 13, yields the value 7. **(b)** The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, we look for windows whose value modulo 13 is 7, since $31415 \equiv 7 \pmod{13}$. The algorithm finds two such windows, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. **(c)** How to compute the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. Because all computations are performed modulo 13, the value for the first window is 7, and the value for the new window is 8.

$P[1..m] = T[s+1..s+m]$. If q is large enough, then we hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure are the text T, the pattern P, the radix d to use (which is typically taken to be $|\Sigma|$), and the prime q to use.

```
RABIN-KARP-MATCHER(T, P, d, q)
 1  n = T.length
 2  m = P.length
 3  h = d^(m-1) mod q
 4  p = 0
 5  t_0 = 0
 6  for i = 1 to m                    // preprocessing
 7      p = (dp + P[i]) mod q
 8      t_0 = (dt_0 + T[i]) mod q
 9  for s = 0 to n - m                // matching
10      if p == t_s
11          if P[1..m] == T[s+1..s+m]
12              print "Pattern occurs with shift" s
13      if s < n - m
14          t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) mod q
```

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4–8 compute p as the value of $P[1..m] \bmod q$ and t_0 as the value of $T[1..m] \bmod q$. The **for** loop of lines 9–14 iterates through all possible shifts s, maintaining the following invariant:

Whenever line 10 is executed, $t_s = T[s+1..s+m] \bmod q$.

If $p = t_s$ in line 10 (a "hit"), then line 11 checks to see whether $P[1..m] = T[s+1..s+m]$ in order to rule out the possibility of a spurious hit. Line 12 prints out any valid shifts that are found. If $s < n - m$ (checked in line 13), then the **for** loop will execute at least one more time, and so line 14 first executes to ensure that the loop invariant holds when we get back to line 10. Line 14 computes the value of $t_{s+1} \bmod q$ from the value of $t_s \bmod q$ in constant time using equation (32.2) directly.

RABIN-KARP-MATCHER takes $\Theta(m)$ preprocessing time, and its matching time is $\Theta((n-m+1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = \mathtt{a}^m$

and $T = \mathtt{a}^n$, then verifying takes time $\Theta((n-m+1)m)$, since each of the $n-m+1$ possible shifts is valid.

In many applications, we expect few valid shifts—perhaps some constant c of them. In such applications, the expected matching time of the algorithm is only $O((n-m+1)+cm) = O(n+m)$, plus the time required to process spurious hits. We can base a heuristic analysis on the assumption that reducing values modulo q acts like a random mapping from Σ^* to $\mathbb{Z}_q$. (See the discussion on the use of division for hashing in Section 11.3.1. It is difficult to formalize and prove such an assumption, although one viable approach is to assume that q is chosen randomly from integers of the appropriate size. We shall not pursue this formalization here.) We can then expect that the number of spurious hits is $O(n/q)$, since we can estimate the chance that an arbitrary t_s will be equivalent to p, modulo q, as $1/q$. Since there are $O(n)$ positions at which the test of line 10 fails and we spend $O(m)$ time for each hit, the expected matching time taken by the Rabin-Karp algorithm is

$$O(n) + O(m(v + n/q)) ,$$

where v is the number of valid shifts. This running time is $O(n)$ if $v = O(1)$ and we choose $q \geq m$. That is, if the expected number of valid shifts is small ($O(1)$) and we choose the prime q to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to use only $O(n+m)$ matching time. Since $m \leq n$, this expected matching time is $O(n)$.

Exercises

32.2-1

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

32.2-2

How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

32.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

32.2-4

Alice has a copy of a long n-bit file $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, and Bob similarly has an n-bit file $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Alice and Bob wish to know if their files are identical. To avoid transmitting all of A or B, they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer x from $\{0, 1, \ldots, q-1\}$. Then, Alice evaluates

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q$$

and Bob similarly evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (*Hint:* See Exercise 31.4-4.)

32.3 String matching with finite automata

Many string-matching algorithms build a finite automaton—a simple machine for processing information—that scans the text string T for all occurrences of the pattern P. This section presents a method for building such an automaton. These string-matching automata are very efficient: they examine each text character *exactly once*, taking constant time per text character. The matching time used—after preprocessing the pattern to build the automaton—is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large. Section 32.4 describes a clever way around this problem.

We begin this section with the definition of a finite automaton. We then examine a special string-matching automaton and show how to use it to find occurrences of a pattern in a text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

Finite automata

A ***finite automaton*** M, illustrated in Figure 32.6, is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.

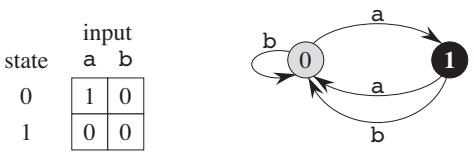


Figure 32.6 A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. **(a)** A tabular representation of the transition function δ. **(b)** An equivalent state-transition diagram. State 1, shown blackend, is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that $\delta(1, \mathtt{b}) = 0$. This automaton accepts those strings that end in an odd number of a's. More precisely, it accepts a string x if and only if $x = yz$, where $y = \varepsilon$ or y ends with a b, and $z = \mathtt{a}^k$, where k is odd. For example, on input abaaa, including the start state, this automaton enters the sequence of states $\langle 0, 1, 0, 1, 0, 1\rangle$, and so it accepts this input. For input abbaa, it enters the sequence of states $\langle 0, 1, 0, 0, 1, 0\rangle$, and so it rejects this input.

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves ("makes a transition") from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M has ***accepted*** the string read so far. An input that is not accepted is ***rejected***.

A finite automaton M induces a function ϕ, called the ***final-state function***, from Σ^* to Q such that $\phi(w)$ is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if $\phi(w) \in A$. We define the function ϕ recursively, using the transition function:

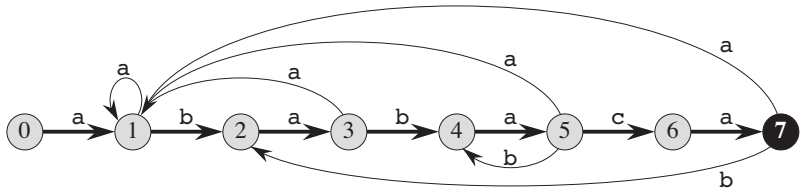
$$\begin{aligned} \phi(\varepsilon) &= q_0 \ , \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma \ . \end{aligned}$$

String-matching automata

For a given pattern P, we construct a string-matching automaton in a preprocessing step before using it to search the text string. Figure 32.7 illustrates how we construct the automaton for the pattern $P = \mathtt{ababaca}$. From now on, we shall assume that P is a given fixed pattern string; for brevity, we shall not indicate the dependence upon P in our notation.

In order to specify the string-matching automaton corresponding to a given pattern $P[1\,..\,m]$, we first define an auxiliary function σ, called the ***suffix function*** corresponding to P. The function σ maps Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} \ . \tag{32.3}$$



(a)

state	input a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	**7**	2	3

(c)

Figure 32.7 **(a)** A state-transition diagram for the string-matching automaton that accepts all strings ending in the string `ababaca`. State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents $\delta(i, a) = j$. The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are omitted; by convention, if a state i has no outgoing edge labeled a for some $a \in \Sigma$, then $\delta(i, a) = 0$. **(b)** The corresponding transition function δ, and the pattern string $P =$ `ababaca`. The entries corresponding to successful matches between pattern and input characters are shown shaded. **(c)** The operation of the automaton on the text $T =$ `abababacaba`. Under each text character $T[i]$ appears the state $\phi(T_i)$ that the automaton is in after processing the prefix T_i. The automaton finds one occurrence of the pattern, ending in position 9.

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string. As examples, for the pattern $P =$ `ab`, we have $\sigma(\varepsilon) = 0$, $\sigma($`ccaca`$) = 1$, and $\sigma($`ccab`$) = 2$. For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \sqsupset x$. From the definition of the suffix function, $x \sqsupset y$ implies $\sigma(x) \leq \sigma(y)$.

We define the string-matching automaton that corresponds to a given pattern $P[1\,..\,m]$ as follows:

- The state set Q is $\{0, 1, \ldots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a:

$$\delta(q, a) = \sigma(P_q a) \; . \tag{32.4}$$

We define $\delta(q, a) = \sigma(P_q a)$ because we want to keep track of the longest prefix of the pattern P that has matched the text string T so far. We consider the most recently read characters of T. In order for a substring of T—let's say the substring ending at $T[i]$—to match some prefix P_j of P, this prefix P_j must be a suffix of T_i. Suppose that $q = \phi(T_i)$, so that after reading T_i, the automaton is in state q. We design the transition function δ so that this state number, q, tells us the length of the longest prefix of P that matches a suffix of T_i. That is, in state q, $P_q \sqsupset T_i$ and $q = \sigma(T_i)$. (Whenever $q = m$, all m characters of P match a suffix of T_i, and so we have found a match.) Thus, since $\phi(T_i)$ and $\sigma(T_i)$ both equal q, we shall see (in Theorem 32.4, below) that the automaton maintains the following invariant:

$$\phi(T_i) = \sigma(T_i) \; . \tag{32.5}$$

If the automaton is in state q and reads the next character $T[i+1] = a$, then we want the transition to lead to the state corresponding to the longest prefix of P that is a suffix of $T_i a$, and that state is $\sigma(T_i a)$. Because P_q is the longest prefix of P that is a suffix of T_i, the longest prefix of P that is a suffix of $T_i a$ is not only $\sigma(T_i a)$, but also $\sigma(P_q a)$. (Lemma 32.3, on page 1000, proves that $\sigma(T_i a) = \sigma(P_q a)$.) Thus, when the automaton is in state q, we want the transition function on character a to take the automaton to state $\sigma(P_q a)$.

There are two cases to consider. In the first case, $a = P[q+1]$, so that the character a continues to match the pattern; in this case, because $\delta(q, a) = q+1$, the transition continues to go along the "spine" of the automaton (the heavy edges in Figure 32.7). In the second case, $a \neq P[q+1]$, so that a does not continue to match the pattern. Here, we must find a smaller prefix of P that is also a suffix of T_i. Because the preprocessing step matches the pattern against itself when creating the string-matching automaton, the transition function quickly identifies the longest such smaller prefix of P.

Let's look at an example. The string-matching automaton of Figure 32.7 has $\delta(5, \mathtt{c}) = 6$, illustrating the first case, in which the match continues. To illustrate the second case, observe that the automaton of Figure 32.7 has $\delta(5, \mathtt{b}) = 4$. We make this transition because if the automaton reads a `b` in state $q = 5$, then $P_q\mathtt{b} = \mathtt{ababab}$, and the longest prefix of P that is also a suffix of `ababab` is $P_4 = \mathtt{abab}$.

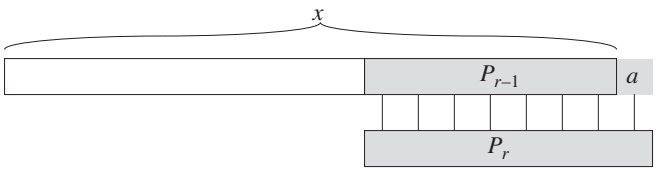


Figure 32.8 An illustration for the proof of Lemma 32.2. The figure shows that $r \leq \sigma(x) + 1$, where $r = \sigma(xa)$.

To clarify the operation of a string-matching automaton, we now give a simple, efficient program for simulating the behavior of such an automaton (represented by its transition function δ) in finding occurrences of a pattern P of length m in an input text $T[1..n]$. As for any string-matching automaton for a pattern of length m, the state set Q is $\{0, 1, \ldots, m\}$, the start state is 0, and the only accepting state is state m.

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n = T.length
2  q = 0
3  for i = 1 to n
4       q = δ(q, T[i])
5       if q == m
6            print "Pattern occurs with shift" i − m
```

From the simple loop structure of FINITE-AUTOMATON-MATCHER, we can easily see that its matching time on a text string of length n is $\Theta(n)$. This matching time, however, does not include the preprocessing time required to compute the transition function δ. We address this problem later, after first proving that the procedure FINITE-AUTOMATON-MATCHER operates correctly.

Consider how the automaton operates on an input text $T[1..n]$. We shall prove that the automaton is in state $\sigma(T_i)$ after scanning character $T[i]$. Since $\sigma(T_i) = m$ if and only if $P \sqsupset T_i$, the machine is in the accepting state m if and only if it has just scanned the pattern P. To prove this result, we make use of the following two lemmas about the suffix function σ.

Lemma 32.2 (Suffix-function inequality)
For any string x and character a, we have $\sigma(xa) \leq \sigma(x) + 1$.

Proof Referring to Figure 32.8, let $r = \sigma(xa)$. If $r = 0$, then the conclusion $\sigma(xa) = r \leq \sigma(x) + 1$ is trivially satisfied, by the nonnegativity of $\sigma(x)$. Now assume that $r > 0$. Then, $P_r \sqsupset xa$, by the definition of σ. Thus, $P_{r-1} \sqsupset x$, by

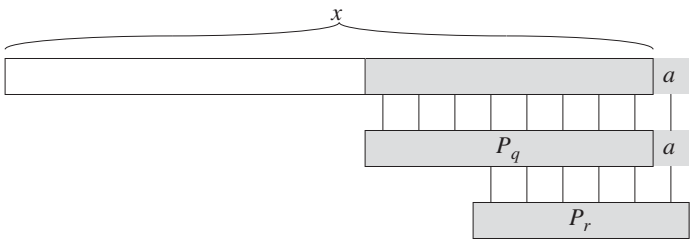


Figure 32.9 An illustration for the proof of Lemma 32.3. The figure shows that $r = \sigma(P_q a)$, where $q = \sigma(x)$ and $r = \sigma(xa)$.

dropping the a from the end of P_r and from the end of xa. Therefore, $r-1 \le \sigma(x)$, since $\sigma(x)$ is the largest k such that $P_k \sqsupset x$, and thus $\sigma(xa) = r \le \sigma(x) + 1$. ■

Lemma 32.3 (Suffix-function recursion lemma)
For any string x and character a, if $q = \sigma(x)$, then $\sigma(xa) = \sigma(P_q a)$.

Proof From the definition of σ, we have $P_q \sqsupset x$. As Figure 32.9 shows, we also have $P_q a \sqsupset xa$. If we let $r = \sigma(xa)$, then $P_r \sqsupset xa$ and, by Lemma 32.2, $r \le q + 1$. Thus, we have $|P_r| = r \le q + 1 = |P_q a|$. Since $P_q a \sqsupset xa$, $P_r \sqsupset xa$, and $|P_r| \le |P_q a|$, Lemma 32.1 implies that $P_r \sqsupset P_q a$. Therefore, $r \le \sigma(P_q a)$, that is, $\sigma(xa) \le \sigma(P_q a)$. But we also have $\sigma(P_q a) \le \sigma(xa)$, since $P_q a \sqsupset xa$. Thus, $\sigma(xa) = \sigma(P_q a)$. ■

We are now ready to prove our main theorem characterizing the behavior of a string-matching automaton on a given input text. As noted above, this theorem shows that the automaton is merely keeping track, at each step, of the longest prefix of the pattern that is a suffix of what has been read so far. In other words, the automaton maintains the invariant (32.5).

Theorem 32.4
If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1\,..\,n]$ is an input text for the automaton, then

$$\phi(T_i) = \sigma(T_i)$$

for $i = 0, 1, \ldots, n$.

Proof The proof is by induction on i. For $i = 0$, the theorem is trivially true, since $T_0 = \varepsilon$. Thus, $\phi(T_0) = 0 = \sigma(T_0)$.

Now, we assume that $\phi(T_i) = \sigma(T_i)$ and prove that $\phi(T_{i+1}) = \sigma(T_{i+1})$. Let q denote $\phi(T_i)$, and let a denote $T[i+1]$. Then,

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(by the definitions of } T_{i+1} \text{ and } a\text{)} \\
&= \delta(\phi(T_i), a) && \text{(by the definition of } \phi\text{)} \\
&= \delta(q, a) && \text{(by the definition of } q\text{)} \\
&= \sigma(P_q a) && \text{(by the definition (32.4) of } \delta\text{)} \\
&= \sigma(T_i a) && \text{(by Lemma 32.3 and induction)} \\
&= \sigma(T_{i+1}) && \text{(by the definition of } T_{i+1}\text{)} \ . \qquad \blacksquare
\end{aligned}$$

By Theorem 32.4, if the machine enters state q on line 4, then q is the largest value such that $P_q \sqsupset T_i$. Thus, we have $q = m$ on line 5 if and only if the machine has just scanned an occurrence of the pattern P. We conclude that FINITE-AUTOMATON-MATCHER operates correctly.

Computing the transition function

The following procedure computes the transition function δ from a given pattern $P[1..m]$.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
1  m = P.length
2  for q = 0 to m
3      for each character a ∈ Σ
4          k = min(m + 1, q + 2)
5          repeat
6              k = k − 1
7          until P_k ⊐ P_q a
8          δ(q, a) = k
9  return δ
```

This procedure computes $\delta(q, a)$ in a straightforward manner according to its definition in equation (32.4). The nested loops beginning on lines 2 and 3 consider all states q and all characters a, and lines 4–8 set $\delta(q, a)$ to be the largest k such that $P_k \sqsupset P_q a$. The code starts with the largest conceivable value of k, which is $\min(m, q+1)$. It then decreases k until $P_k \sqsupset P_q a$, which must eventually occur, since $P_0 = \varepsilon$ is a suffix of every string.

The running time of COMPUTE-TRANSITION-FUNCTION is $O(m^3 |\Sigma|)$, because the outer loops contribute a factor of $m|\Sigma|$, the inner **repeat** loop can run at most $m+1$ times, and the test $P_k \sqsupset P_q a$ on line 7 can require comparing up

to m characters. Much faster procedures exist; by utilizing some cleverly computed information about the pattern P (see Exercise 32.4-8), we can improve the time required to compute δ from P to $O(m\,|\Sigma|)$. With this improved procedure for computing δ, we can find all occurrences of a length-m pattern in a length-n text over an alphabet Σ with $O(m\,|\Sigma|)$ preprocessing time and $\Theta(n)$ matching time.

Exercises

32.3-1
Construct the string-matching automaton for the pattern $P = \texttt{aabab}$ and illustrate its operation on the text string $T = \texttt{aaababaabaababaab}$.

32.3-2
Draw a state-transition diagram for a string-matching automaton for the pattern $\texttt{ababbabbababbababbabb}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$.

32.3-3
We call a pattern P ***nonoverlappable*** if $P_k \sqsupset P_q$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

32.3-4 ⋆
Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of *either* pattern. Try to minimize the number of states in your automaton.

32.3-5
Given a pattern P containing gap characters (see Exercise 32.1-4), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ matching time, where $n = |T|$.

⋆ 32.4 The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. This algorithm avoids computing the transition function δ altogether, and its matching time is $\Theta(n)$ using just an auxiliary function π, which we precompute from the pattern in time $\Theta(m)$ and store in an array $\pi[1\,..\,m]$. The array π allows us to compute the transition function δ efficiently (in an amortized sense) "on the fly" as needed. Loosely speaking, for any state $q = 0, 1, \ldots, m$ and any character

$a \in \Sigma$, the value $\pi[q]$ contains the information we need to compute $\delta(q, a)$ but that does not depend on a. Since the array π has only m entries, whereas δ has $\Theta(m|\Sigma|)$ entries, we save a factor of $|\Sigma|$ in the preprocessing time by computing π rather than δ.

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. We can take advantage of this information to avoid testing useless shifts in the naive pattern-matching algorithm and to avoid precomputing the full transition function δ for a string-matching automaton.

Consider the operation of the naive string matcher. Figure 32.10(a) shows a particular shift s of a template containing the pattern $P = \texttt{ababaca}$ against a text T. For this example, $q = 5$ of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing these q text characters allows us to determine immediately that certain shifts are invalid. In the example of the figure, the shift $s + 1$ is necessarily invalid, since the first pattern character (`a`) would be aligned with a text character that we know does not match the first pattern character, but does match the second pattern character (`b`). The shift $s' = s + 2$ shown in part (b) of the figure, however, aligns the first three pattern characters with three text characters that must necessarily match. In general, it is useful to know the answer to the following question:

> Given that pattern characters $P[1..q]$ match text characters $T[s+1..s+q]$, what is the least shift $s' > s$ such that for some $k < q$,
>
> $$P[1..k] = T[s' + 1..s' + k]\,, \tag{32.6}$$
>
> where $s' + k = s + q$?

In other words, knowing that $P_q \sqsupset T_{s+q}$, we want the longest proper prefix P_k of P_q that is also a suffix of T_{s+q}. (Since $s' + k = s + q$, if we are given s and q, then finding the smallest shift s' is tantamount to finding the longest prefix length k.) We add the difference $q - k$ in the lengths of these prefixes of P to the shift s to arrive at our new shift s', so that $s' = s + (q - k)$. In the best case, $k = 0$, so that $s' = s + q$, and we immediately rule out shifts $s + 1, s + 2, \ldots, s + q - 1$. In any case, at the new shift s' we don't need to compare the first k characters of P with the corresponding characters of T, since equation (32.6) guarantees that they match.

We can precompute the necessary information by comparing the pattern against itself, as Figure 32.10(c) demonstrates. Since $T[s' + 1..s' + k]$ is part of the

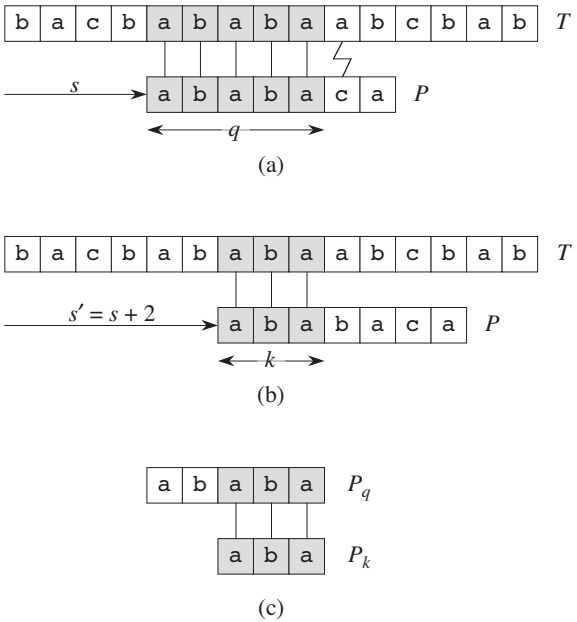


Figure 32.10 The prefix function π. **(a)** The pattern $P = \texttt{ababaca}$ aligns with a text T so that the first $q = 5$ characters match. Matching characters, shown shaded, are connected by vertical lines. **(b)** Using only our knowledge of the 5 matched characters, we can deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s+2$ is consistent with everything we know about the text and therefore is potentially valid. **(c)** We can precompute useful information for such deductions by comparing the pattern with itself. Here, we see that the longest prefix of P that is also a proper suffix of P_5 is P_3. We represent this precomputed information in the array π, so that $\pi[5] = 3$. Given that q characters have matched successfully at shift s, the next potentially valid shift is at $s' = s + (q - \pi[q])$ as shown in part (b).

known portion of the text, it is a suffix of the string P_q. Therefore, we can interpret equation (32.6) as asking for the greatest $k < q$ such that $P_k \sqsupset P_q$. Then, the new shift $s' = s + (q - k)$ is the next potentially valid shift. We will find it convenient to store, for each value of q, the number k of matching characters at the new shift s', rather than storing, say, $s' - s$.

We formalize the information that we precompute as follows. Given a pattern $P[1\,..\,m]$, the ***prefix function*** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ such that

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\} \ .$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q. Figure 32.11(a) gives the complete prefix function π for the pattern `ababaca`.

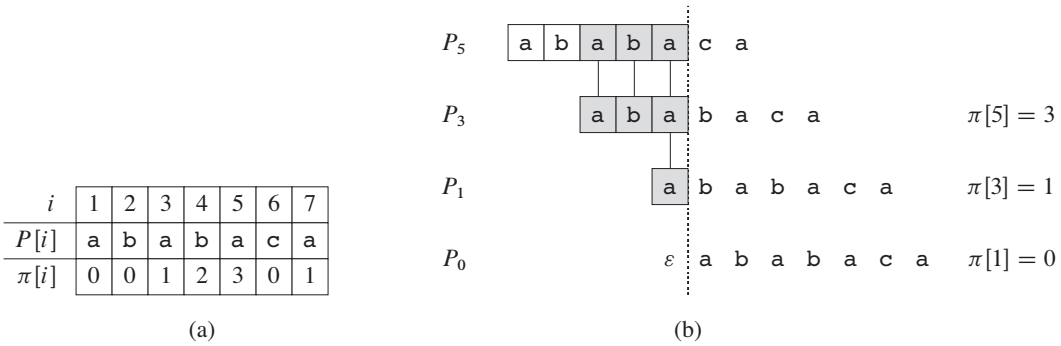


Figure 32.11 An illustration of Lemma 32.5 for the pattern $P = \texttt{ababaca}$ and $q = 5$. **(a)** The π function for the given pattern. Since $\pi[5] = 3$, $\pi[3] = 1$, and $\pi[1] = 0$, by iterating π we obtain $\pi^*[5] = \{3, 1, 0\}$. **(b)** We slide the template containing the pattern P to the right and note when some prefix P_k of P matches up with some proper suffix of P_5; we get matches when $k = 3$, 1, and 0. In the figure, the first row gives P, and the dotted vertical line is drawn just after P_5. Successive rows show all the shifts of P that cause some prefix P_k of P to match some suffix of P_5. Successfully matched characters are shown shaded. Vertical lines connect aligned matching characters. Thus, $\{k : k < 5 \text{ and } P_k \sqsupset P_5\} = \{3, 1, 0\}$. Lemma 32.5 claims that $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$ for all q.

The pseudocode below gives the Knuth-Morris-Pratt matching algorithm as the procedure KMP-MATCHER. For the most part, the procedure follows from FINITE-AUTOMATON-MATCHER, as we shall see. KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION to compute π.

```
KMP-MATCHER(T, P)
 1  n = T.length
 2  m = P.length
 3  π = COMPUTE-PREFIX-FUNCTION(P)
 4  q = 0                                  // number of characters matched
 5  for i = 1 to n                         // scan the text from left to right
 6      while q > 0 and P[q + 1] ≠ T[i]
 7          q = π[q]                       // next character does not match
 8      if P[q + 1] == T[i]
 9          q = q + 1                      // next character matches
10      if q == m                          // is all of P matched?
11          print "Pattern occurs with shift" i − m
12          q = π[q]                       // look for the next match
```

```
COMPUTE-PREFIX-FUNCTION(P)
 1  m = P.length
 2  let π[1..m] be a new array
 3  π[1] = 0
 4  k = 0
 5  for q = 2 to m
 6      while k > 0 and P[k + 1] ≠ P[q]
 7          k = π[k]
 8      if P[k + 1] == P[q]
 9          k = k + 1
10      π[q] = k
11  return π
```

These two procedures have much in common, because both match a string against the pattern P: KMP-MATCHER matches the text T against P, and COMPUTE-PREFIX-FUNCTION matches P against itself.

We begin with an analysis of the running times of these procedures. Proving these procedures correct will be more complicated.

Running-time analysis

The running time of COMPUTE-PREFIX-FUNCTION is $\Theta(m)$, which we show by using the aggregate method of amortized analysis (see Section 17.1). The only tricky part is showing that the **while** loop of lines 6–7 executes $O(m)$ times altogether. We shall show that it makes at most $m - 1$ iterations. We start by making some observations about k. First, line 4 starts k at 0, and the only way that k increases is by the increment operation in line 9, which executes at most once per iteration of the **for** loop of lines 5–10. Thus, the total increase in k is at most $m-1$. Second, since $k < q$ upon entering the **for** loop and each iteration of the loop increments q, we always have $k < q$. Therefore, the assignments in lines 3 and 10 ensure that $\pi[q] < q$ for all $q = 1, 2, \ldots, m$, which means that each iteration of the **while** loop decreases k. Third, k never becomes negative. Putting these facts together, we see that the total decrease in k from the **while** loop is bounded from above by the total increase in k over all iterations of the **for** loop, which is $m - 1$. Thus, the **while** loop iterates at most $m - 1$ times in all, and COMPUTE-PREFIX-FUNCTION runs in time $\Theta(m)$.

Exercise 32.4-4 asks you to show, by a similar aggregate analysis, that the matching time of KMP-MATCHER is $\Theta(n)$.

Compared with FINITE-AUTOMATON-MATCHER, by using π rather than δ, we have reduced the time for preprocessing the pattern from $O(m\,|\Sigma|)$ to $\Theta(m)$, while keeping the actual matching time bounded by $\Theta(n)$.

Correctness of the prefix-function computation

We shall see a little later that the prefix function π helps us simulate the transition function δ in a string-matching automaton. But first, we need to prove that the procedure COMPUTE-PREFIX-FUNCTION does indeed compute the prefix function correctly. In order to do so, we will need to find all prefixes P_k that are proper suffixes of a given prefix P_q. The value of $\pi[q]$ gives us the longest such prefix, but the following lemma, illustrated in Figure 32.11, shows that by iterating the prefix function π, we can indeed enumerate all the prefixes P_k that are proper suffixes of P_q. Let

$$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\} ,$$

where $\pi^{(i)}[q]$ is defined in terms of functional iteration, so that $\pi^{(0)}[q] = q$ and $\pi^{(i)}[q] = \pi[\pi^{(i-1)}[q]]$ for $i \geq 1$, and where the sequence in $\pi^*[q]$ stops upon reaching $\pi^{(t)}[q] = 0$.

Lemma 32.5 (Prefix-function iteration lemma)
Let P be a pattern of length m with prefix function π. Then, for $q = 1, 2, \ldots, m$, we have $\pi^*[q] = \{k : k < q \text{ and } P_k \sqsupset P_q\}$.

Proof We first prove that $\pi^*[q] \subseteq \{k : k < q \text{ and } P_k \sqsupset P_q\}$ or, equivalently,

$$i \in \pi^*[q] \text{ implies } P_i \sqsupset P_q . \tag{32.7}$$

If $i \in \pi^*[q]$, then $i = \pi^{(u)}[q]$ for some $u > 0$. We prove equation (32.7) by induction on u. For $u = 1$, we have $i = \pi[q]$, and the claim follows since $i < q$ and $P_{\pi[q]} \sqsupset P_q$ by the definition of π. Using the relations $\pi[i] < i$ and $P_{\pi[i]} \sqsupset P_i$ and the transitivity of $<$ and $\sqsupset$ establishes the claim for all i in $\pi^*[q]$. Therefore, $\pi^*[q] \subseteq \{k : k < q \text{ and } P_k \sqsupset P_q\}$.

We now prove that $\{k : k < q \text{ and } P_k \sqsupset P_q\} \subseteq \pi^*[q]$ by contradiction. Suppose to the contrary that the set $\{k : k < q \text{ and } P_k \sqsupset P_q\} - \pi^*[q]$ is nonempty, and let j be the largest number in the set. Because $\pi[q]$ is the largest value in $\{k : k < q \text{ and } P_k \sqsupset P_q\}$ and $\pi[q] \in \pi^*[q]$, we must have $j < \pi[q]$, and so we let j' denote the smallest integer in $\pi^*[q]$ that is greater than j. (We can choose $j' = \pi[q]$ if no other number in $\pi^*[q]$ is greater than j.) We have $P_j \sqsupset P_q$ because $j \in \{k : k < q \text{ and } P_k \sqsupset P_q\}$, and from $j' \in \pi^*[q]$ and equation (32.7), we have $P_{j'} \sqsupset P_q$. Thus, $P_j \sqsupset P_{j'}$ by Lemma 32.1, and j is the largest value less than j' with this property. Therefore, we must have $\pi[j'] = j$ and, since $j' \in \pi^*[q]$, we must have $j \in \pi^*[q]$ as well. This contradiction proves the lemma. ■

The algorithm COMPUTE-PREFIX-FUNCTION computes $\pi[q]$, in order, for $q = 1, 2, \ldots, m$. Setting $\pi[1]$ to 0 in line 3 of COMPUTE-PREFIX-FUNCTION is certainly correct, since $\pi[q] < q$ for all q. We shall use the following lemma and

its corollary to prove that COMPUTE-PREFIX-FUNCTION computes $\pi[q]$ correctly for $q > 1$.

Lemma 32.6
Let P be a pattern of length m, and let π be the prefix function for P. For $q = 1, 2, \ldots, m$, if $\pi[q] > 0$, then $\pi[q] - 1 \in \pi^*[q-1]$.

Proof Let $r = \pi[q] > 0$, so that $r < q$ and $P_r \sqsupset P_q$; thus, $r - 1 < q - 1$ and $P_{r-1} \sqsupset P_{q-1}$ (by dropping the last character from P_r and P_q, which we can do because $r > 0$). By Lemma 32.5, therefore, $r - 1 \in \pi^*[q-1]$. Thus, we have $\pi[q] - 1 = r - 1 \in \pi^*[q-1]$. ∎

For $q = 2, 3, \ldots, m$, define the subset $E_{q-1} \subseteq \pi^*[q-1]$ by

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ and } P_k \sqsupset P_{q-1} \text{ and } P[k+1] = P[q]\} \text{ (by Lemma 32.5)} \\ &= \{k : k < q-1 \text{ and } P_{k+1} \sqsupset P_q\} \ . \end{aligned}$$

The set E_{q-1} consists of the values $k < q-1$ for which $P_k \sqsupset P_{q-1}$ and for which, because $P[k+1] = P[q]$, we have $P_{k+1} \sqsupset P_q$. Thus, E_{q-1} consists of those values $k \in \pi^*[q-1]$ such that we can extend P_k to P_{k+1} and get a proper suffix of P_q.

Corollary 32.7
Let P be a pattern of length m, and let π be the prefix function for P. For $q = 2, 3, \ldots, m$,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \ , \\ 1 + \max\{k \in E_{q-1}\} & \text{if } E_{q-1} \neq \emptyset \ . \end{cases}$$

Proof If E_{q-1} is empty, there is no $k \in \pi^*[q-1]$ (including $k = 0$) for which we can extend P_k to P_{k+1} and get a proper suffix of P_q. Therefore $\pi[q] = 0$.

If E_{q-1} is nonempty, then for each $k \in E_{q-1}$ we have $k+1 < q$ and $P_{k+1} \sqsupset P_q$. Therefore, from the definition of $\pi[q]$, we have

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\} \ . \tag{32.8}$$

Note that $\pi[q] > 0$. Let $r = \pi[q] - 1$, so that $r + 1 = \pi[q]$ and therefore $P_{r+1} \sqsupset P_q$. Since $r + 1 > 0$, we have $P[r+1] = P[q]$. Furthermore, by Lemma 32.6, we have $r \in \pi^*[q-1]$. Therefore, $r \in E_{q-1}$, and so $r \leq \max\{k \in E_{q-1}\}$ or, equivalently,

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\} \ . \tag{32.9}$$

Combining equations (32.8) and (32.9) completes the proof. ∎

We now finish the proof that COMPUTE-PREFIX-FUNCTION computes π correctly. In the procedure COMPUTE-PREFIX-FUNCTION, at the start of each iteration of the **for** loop of lines 5–10, we have that $k = \pi[q-1]$. This condition is enforced by lines 3 and 4 when the loop is first entered, and it remains true in each successive iteration because of line 10. Lines 6–9 adjust k so that it becomes the correct value of $\pi[q]$. The **while** loop of lines 6–7 searches through all values $k \in \pi^*[q-1]$ until it finds a value of k for which $P[k+1] = P[q]$; at that point, k is the largest value in the set E_{q-1}, so that, by Corollary 32.7, we can set $\pi[q]$ to $k+1$. If the **while** loop cannot find a $k \in \pi^*[q-1]$ such that $P[k+1] = P[q]$, then k equals 0 at line 8. If $P[1] = P[q]$, then we should set both k and $\pi[q]$ to 1; otherwise we should leave k alone and set $\pi[q]$ to 0. Lines 8–10 set k and $\pi[q]$ correctly in either case. This completes our proof of the correctness of COMPUTE-PREFIX-FUNCTION.

Correctness of the Knuth-Morris-Pratt algorithm

We can think of the procedure KMP-MATCHER as a reimplemented version of the procedure FINITE-AUTOMATON-MATCHER, but using the prefix function π to compute state transitions. Specifically, we shall prove that in the ith iteration of the **for** loops of both KMP-MATCHER and FINITE-AUTOMATON-MATCHER, the state q has the same value when we test for equality with m (at line 10 in KMP-MATCHER and at line 5 in FINITE-AUTOMATON-MATCHER). Once we have argued that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER, the correctness of KMP-MATCHER follows from the correctness of FINITE-AUTOMATON-MATCHER (though we shall see a little later why line 12 in KMP-MATCHER is necessary).

Before we formally prove that KMP-MATCHER correctly simulates FINITE-AUTOMATON-MATCHER, let's take a moment to understand how the prefix function π replaces the δ transition function. Recall that when a string-matching automaton is in state q and it scans a character $a = T[i]$, it moves to a new state $\delta(q, a)$. If $a = P[q+1]$, so that a continues to match the pattern, then $\delta(q, a) = q + 1$. Otherwise, $a \neq P[q+1]$, so that a does not continue to match the pattern, and $0 \leq \delta(q, a) \leq q$. In the first case, when a continues to match, KMP-MATCHER moves to state $q+1$ without referring to the π function: the **while** loop test in line 6 comes up false the first time, the test in line 8 comes up true, and line 9 increments q.

The π function comes into play when the character a does not continue to match the pattern, so that the new state $\delta(q, a)$ is either q or to the left of q along the spine of the automaton. The **while** loop of lines 6–7 in KMP-MATCHER iterates through the states in $\pi^*[q]$, stopping either when it arrives in a state, say q', such that a matches $P[q'+1]$ or q' has gone all the way down to 0. If a matches $P[q'+1]$,

then line 9 sets the new state to $q'+1$, which should equal $\delta(q, a)$ for the simulation to work correctly. In other words, the new state $\delta(q, a)$ should be either state 0 or one greater than some state in $\pi^*[q]$.

Let's look at the example in Figures 32.7 and 32.11, which are for the pattern $P = \texttt{ababaca}$. Suppose that the automaton is in state $q = 5$; the states in $\pi^*[5]$ are, in descending order, 3, 1, and 0. If the next character scanned is `c`, then we can easily see that the automaton moves to state $\delta(5, \texttt{c}) = 6$ in both FINITE-AUTOMATON-MATCHER and KMP-MATCHER. Now suppose that the next character scanned is instead `b`, so that the automaton should move to state $\delta(5, \texttt{b}) = 4$. The **while** loop in KMP-MATCHER exits having executed line 7 once, and it arrives in state $q' = \pi[5] = 3$. Since $P[q' + 1] = P[4] = \texttt{b}$, the test in line 8 comes up true, and KMP-MATCHER moves to the new state $q' + 1 = 4 = \delta(5, \texttt{b})$. Finally, suppose that the next character scanned is instead `a`, so that the automaton should move to state $\delta(5, \texttt{a}) = 1$. The first three times that the test in line 6 executes, the test comes up true. The first time, we find that $P[6] = \texttt{c} \neq \texttt{a}$, and KMP-MATCHER moves to state $\pi[5] = 3$ (the first state in $\pi^*[5]$). The second time, we find that $P[4] = \texttt{b} \neq \texttt{a}$ and move to state $\pi[3] = 1$ (the second state in $\pi^*[5]$). The third time, we find that $P[2] = \texttt{b} \neq \texttt{a}$ and move to state $\pi[1] = 0$ (the last state in $\pi^*[5]$). The **while** loop exits once it arrives in state $q' = 0$. Now, line 8 finds that $P[q' + 1] = P[1] = \texttt{a}$, and line 9 moves the automaton to the new state $q' + 1 = 1 = \delta(5, \texttt{a})$.

Thus, our intuition is that KMP-MATCHER iterates through the states in $\pi^*[q]$ in decreasing order, stopping at some state q' and then possibly moving to state $q'+1$. Although that might seem like a lot of work just to simulate computing $\delta(q, a)$, bear in mind that asymptotically, KMP-MATCHER is no slower than FINITE-AUTOMATON-MATCHER.

We are now ready to formally prove the correctness of the Knuth-Morris-Pratt algorithm. By Theorem 32.4, we have that $q = \sigma(T_i)$ after each time we execute line 4 of FINITE-AUTOMATON-MATCHER. Therefore, it suffices to show that the same property holds with regard to the **for** loop in KMP-MATCHER. The proof proceeds by induction on the number of loop iterations. Initially, both procedures set q to 0 as they enter their respective **for** loops for the first time. Consider iteration i of the **for** loop in KMP-MATCHER, and let q' be state at the start of this loop iteration. By the inductive hypothesis, we have $q' = \sigma(T_{i-1})$. We need to show that $q = \sigma(T_i)$ at line 10. (Again, we shall handle line 12 separately.)

When we consider the character $T[i]$, the longest prefix of P that is a suffix of T_i is either $P_{q'+1}$ (if $P[q' + 1] = T[i]$) or some prefix (not necessarily proper, and possibly empty) of $P_{q'}$. We consider separately the three cases in which $\sigma(T_i) = 0$, $\sigma(T_i) = q' + 1$, and $0 < \sigma(T_i) \leq q'$.

- If $\sigma(T_i) = 0$, then $P_0 = \varepsilon$ is the only prefix of P that is a suffix of T_i. The **while** loop of lines 6–7 iterates through the values in $\pi^*[q']$, but although $P_q \sqsupset T_i$ for every $q \in \pi^*[q']$, the loop never finds a q such that $P[q+1] = T[i]$. The loop terminates when q reaches 0, and of course line 9 does not execute. Therefore, $q = 0$ at line 10, so that $q = \sigma(T_i)$.
- If $\sigma(T_i) = q' + 1$, then $P[q'+1] = T[i]$, and the **while** loop test in line 6 fails the first time through. Line 9 executes, incrementing q so that afterward we have $q = q' + 1 = \sigma(T_i)$.
- If $0 < \sigma(T_i) \le q'$, then the **while** loop of lines 6–7 iterates at least once, checking in decreasing order each value $q \in \pi^*[q']$ until it stops at some $q < q'$. Thus, P_q is the longest prefix of $P_{q'}$ for which $P[q+1] = T[i]$, so that when the **while** loop terminates, $q + 1 = \sigma(P_{q'}T[i])$. Since $q' = \sigma(T_{i-1})$, Lemma 32.3 implies that $\sigma(T_{i-1}T[i]) = \sigma(P_{q'}T[i])$. Thus, we have

$$\begin{aligned} q + 1 &= \sigma(P_{q'}T[i]) \\ &= \sigma(T_{i-1}T[i]) \\ &= \sigma(T_i) \end{aligned}$$

  when the **while** loop terminates. After line 9 increments q, we have $q = \sigma(T_i)$.

Line 12 is necessary in KMP-MATCHER, because otherwise, we might reference $P[m+1]$ on line 6 after finding an occurrence of P. (The argument that $q = \sigma(T_{i-1})$ upon the next execution of line 6 remains valid by the hint given in Exercise 32.4-8: $\delta(m, a) = \delta(\pi[m], a)$ or, equivalently, $\sigma(Pa) = \sigma(P_{\pi[m]}a)$ for any $a \in \Sigma$.) The remaining argument for the correctness of the Knuth-Morris-Pratt algorithm follows from the correctness of FINITE-AUTOMATON-MATCHER, since we have shown that KMP-MATCHER simulates the behavior of FINITE-AUTOMATON-MATCHER.

Exercises

32.4-1

Compute the prefix function π for the pattern `ababbabbabbababbabb`.

32.4-2

Give an upper bound on the size of $\pi^*[q]$ as a function of q. Give an example to show that your bound is tight.

32.4-3

Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT (the string of length $m+n$ that is the concatenation of P and T).

32.4-4
Use an aggregate analysis to show that the running time of KMP-MATCHER is $\Theta(n)$.

32.4-5
Use a potential function to show that the running time of KMP-MATCHER is $\Theta(n)$.

32.4-6
Show how to improve KMP-MATCHER by replacing the occurrence of π in line 7 (but not line 12) by π', where π' is defined recursively for $q = 1, 2, \ldots, m-1$ by the equation

$$\pi'[q] = \begin{cases} 0 & \text{if } \pi[q] = 0\,, \\ \pi'[\pi[q]] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] = P[q+1]\,, \\ \pi[q] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q]+1] \neq P[q+1]\,. \end{cases}$$

Explain why the modified algorithm is correct, and explain in what sense this change constitutes an improvement.

32.4-7
Give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T'. For example, `arc` and `car` are cyclic rotations of each other.

32.4-8 ⋆
Give an $O(m\,|\Sigma|)$-time algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern P. (*Hint:* Prove that $\delta(q, a) = \delta(\pi[q], a)$ if $q = m$ or $P[q+1] \neq a$.)

Problems

32-1 String matching based on repetition factors
Let y^i denote the concatenation of string y with itself i times. For example, $(\texttt{ab})^3 = \texttt{ababab}$. We say that a string $x \in \Sigma^*$ has ***repetition factor*** r if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let $\rho(x)$ denote the largest r such that x has repetition factor r.

a. Give an efficient algorithm that takes as input a pattern $P[1\,..\,m]$ and computes the value $\rho(P_i)$ for $i = 1, 2, \ldots, m$. What is the running time of your algorithm?

b. For any pattern $P[1..m]$, let $\rho^*(P)$ be defined as $\max_{1 \le i \le m} \rho(P_i)$. Prove that if the pattern P is chosen randomly from the set of all binary strings of length m, then the expected value of $\rho^*(P)$ is $O(1)$.

c. Argue that the following string-matching algorithm correctly finds all occurrences of pattern P in a text $T[1..n]$ in time $O(\rho^*(P)n + m)$:

```
REPETITION-MATCHER(P, T)
 1  m = P.length
 2  n = T.length
 3  k = 1 + ρ*(P)
 4  q = 0
 5  s = 0
 6  while s ≤ n − m
 7      if T[s + q + 1] == P[q + 1]
 8          q = q + 1
 9          if q == m
10              print "Pattern occurs with shift" s
11      if q == m or T[s + q + 1] ≠ P[q + 1]
12          s = s + max(1, ⌈q/k⌉)
13          q = 0
```

This algorithm is due to Galil and Seiferas. By extending these ideas greatly, they obtained a linear-time string-matching algorithm that uses only $O(1)$ storage beyond what is required for P and T.

Chapter notes

The relation of string matching to the theory of finite automata is discussed by Aho, Hopcroft, and Ullman [5]. The Knuth-Morris-Pratt algorithm [214] was invented independently by Knuth and Pratt and by Morris; they published their work jointly. Reingold, Urban, and Gries [294] give an alternative treatment of the Knuth-Morris-Pratt algorithm. The Rabin-Karp algorithm was proposed by Karp and Rabin [201]. Galil and Seiferas [126] give an interesting deterministic linear-time string-matching algorithm that uses only $O(1)$ space beyond that required to store the pattern and text.

33 Computational Geometry

Computational geometry is the branch of computer science that studies algorithms for solving geometric problems. In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VLSI design, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics. The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

In this chapter, we look at a few computational-geometry algorithms in two dimensions, that is, in the plane. We represent each input object by a set of points $\{p_1, p_2, p_3, \ldots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$. For example, we represent an n-vertex polygon P by a sequence $\langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$ of its vertices in order of their appearance on the boundary of P. Computational geometry can also apply to three dimensions, and even higher-dimensional spaces, but such problems and their solutions can be very difficult to visualize. Even in two dimensions, however, we can see a good sample of computational-geometry techniques.

Section 33.1 shows how to answer basic questions about line segments efficiently and accurately: whether one segment is clockwise or counterclockwise from another that shares an endpoint, which way we turn when traversing two adjoining line segments, and whether two line segments intersect. Section 33.2 presents a technique called "sweeping" that we use to develop an $O(n \lg n)$-time algorithm for determining whether a set of n line segments contains any intersections. Section 33.3 gives two "rotational-sweep" algorithms that compute the convex hull (smallest enclosing convex polygon) of a set of n points: Graham's scan, which runs in time $O(n \lg n)$, and Jarvis's march, which takes $O(nh)$ time, where h is the number of vertices of the convex hull. Finally, Section 33.4 gives

an $O(n \lg n)$-time divide-and-conquer algorithm for finding the closest pair of points in a set of n points in the plane.

33.1 Line-segment properties

Several of the computational-geometry algorithms in this chapter require answers to questions about the properties of line segments. A ***convex combination*** of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \le \alpha \le 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$. We also write that $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitively, p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line. Given two distinct points p_1 and p_2, the ***line segment*** $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2. We call p_1 and p_2 the ***endpoints*** of segment $\overline{p_1 p_2}$. Sometimes the ordering of p_1 and p_2 matters, and we speak of the ***directed segment*** $\overrightarrow{p_1 p_2}$. If p_1 is the ***origin*** $(0, 0)$, then we can treat the directed segment $\overrightarrow{p_1 p_2}$ as the ***vector*** p_2.

In this section, we shall explore the following questions:

1. Given two directed segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_0 p_2}$, is $\overrightarrow{p_0 p_1}$ clockwise from $\overrightarrow{p_0 p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$, if we traverse $\overline{p_0 p_1}$ and then $\overline{p_1 p_2}$, do we make a left turn at point p_1?
3. Do line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect?

There are no restrictions on the given points.

We can answer each question in $O(1)$ time, which should come as no surprise since the input size of each question is $O(1)$. Moreover, our methods use only additions, subtractions, multiplications, and comparisons. We need neither division nor trigonometric functions, both of which can be computationally expensive and prone to problems with round-off error. For example, the "straightforward" method of determining whether two segments intersect—compute the line equation of the form $y = mx + b$ for each segment (m is the slope and b is the y-intercept), find the point of intersection of the lines, and check whether this point is on both segments—uses division to find the point of intersection. When the segments are nearly parallel, this method is very sensitive to the precision of the division operation on real computers. The method in this section, which avoids division, is much more accurate.

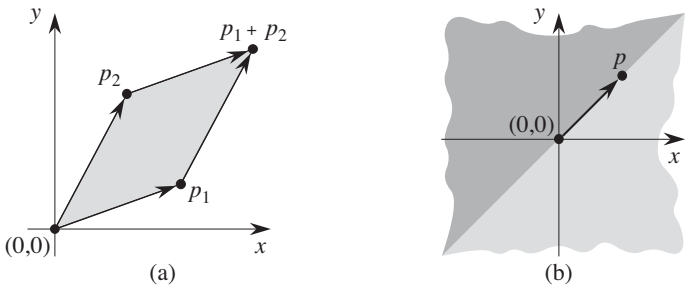


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

Cross products

Computing cross products lies at the heart of our line-segment methods. Consider vectors p_1 and p_2, shown in Figure 33.1(a). We can interpret the ***cross product*** $p_1 \times p_2$ as the signed area of the parallelogram formed by the points $(0, 0)$, p_1, p_2, and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. An equivalent, but more useful, definition gives the cross product as the determinant of a matrix:[1]

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \ . \end{aligned}$$

If $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2 with respect to the origin $(0, 0)$; if this cross product is negative, then p_1 is counterclockwise from p_2. (See Exercise 33.1-1.) Figure 33.1(b) shows the clockwise and counterclockwise regions relative to a vector p. A boundary condition arises if the cross product is 0; in this case, the vectors are ***colinear***, pointing in either the same or opposite directions.

To determine whether a directed segment $\overrightarrow{p_0 p_1}$ is closer to a directed segment $\overrightarrow{p_0 p_2}$ in a clockwise direction or in a counterclockwise direction with respect to their common endpoint p_0, we simply translate to use p_0 as the origin. That is, we let $p_1 - p_0$ denote the vector $p_1' = (x_1', y_1')$, where $x_1' = x_1 - x_0$ and $y_1' = y_1 - y_0$, and we define $p_2 - p_0$ similarly. We then compute the cross product

[1] Actually, the cross product is a three-dimensional concept. It is a vector that is perpendicular to both p_1 and p_2 according to the "right-hand rule" and whose magnitude is $|x_1 y_2 - x_2 y_1|$. In this chapter, however, we find it convenient to treat the cross product simply as the value $x_1 y_2 - x_2 y_1$.

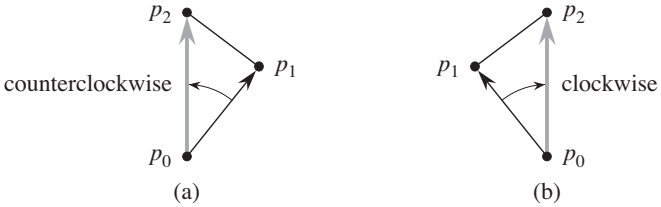


Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) \ .$$

If this cross product is positive, then $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$; if negative, it is counterclockwise.

Determining whether consecutive segments turn left or right

Our next question is whether two consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn left or right at point p_1. Equivalently, we want a method to determine which way a given angle $\angle p_0p_1p_2$ turns. Cross products allow us to answer this question without computing the angle. As Figure 33.2 shows, we simply check whether directed segment $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to directed segment $\overrightarrow{p_0p_1}$. To do so, we compute the cross product $(p_2 - p_0) \times (p_1 - p_0)$. If the sign of this cross product is negative, then $\overrightarrow{p_0p_2}$ is counterclockwise with respect to $\overrightarrow{p_0p_1}$, and thus we make a left turn at p_1. A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 means that points p_0, p_1, and p_2 are colinear.

Determining whether two line segments intersect

To determine whether two line segments intersect, we check whether each segment straddles the line containing the other. A segment $\overline{p_1p_2}$ ***straddles*** a line if point p_1 lies on one side of the line and point p_2 lies on the other side. A boundary case arises if p_1 or p_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions holds:

1. Each segment straddles the line containing the other.
2. An endpoint of one segment lies on the other segment. (This condition comes from the boundary case.)

The following procedures implement this idea. SEGMENTS-INTERSECT returns TRUE if segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect and FALSE if they do not. It calls the subroutines DIRECTION, which computes relative orientations using the cross-product method above, and ON-SEGMENT, which determines whether a point known to be colinear with a segment lies on that segment.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
 1  d1 = DIRECTION(p3, p4, p1)
 2  d2 = DIRECTION(p3, p4, p2)
 3  d3 = DIRECTION(p1, p2, p3)
 4  d4 = DIRECTION(p1, p2, p4)
 5  if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
 6      return TRUE
 7  elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
 8      return TRUE
 9  elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
10      return TRUE
11  elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
12      return TRUE
13  elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
14      return TRUE
15  else return FALSE

DIRECTION(pi, pj, pk)
1  return (pk − pi) × (pj − pi)

ON-SEGMENT(pi, pj, pk)
1  if min(xi, xj) ≤ xk ≤ max(xi, xj) and min(yi, yj) ≤ yk ≤ max(yi, yj)
2      return TRUE
3  else return FALSE
```

SEGMENTS-INTERSECT works as follows. Lines 1–4 compute the relative orientation d_i of each endpoint p_i with respect to the other segment. If all the relative orientations are nonzero, then we can easily determine whether segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect, as follows. Segment $\overline{p_1p_2}$ straddles the line containing segment $\overline{p_3p_4}$ if directed segments $\overrightarrow{p_3p_1}$ and $\overrightarrow{p_3p_2}$ have opposite orientations relative to $\overrightarrow{p_3p_4}$. In this case, the signs of d_1 and d_2 differ. Similarly, $\overline{p_3p_4}$ straddles the line containing $\overline{p_1p_2}$ if the signs of d_3 and d_4 differ. If the test of line 5 is true, then the segments straddle each other, and SEGMENTS-INTERSECT returns TRUE. Figure 33.3(a) shows this case. Otherwise, the segments do not straddle

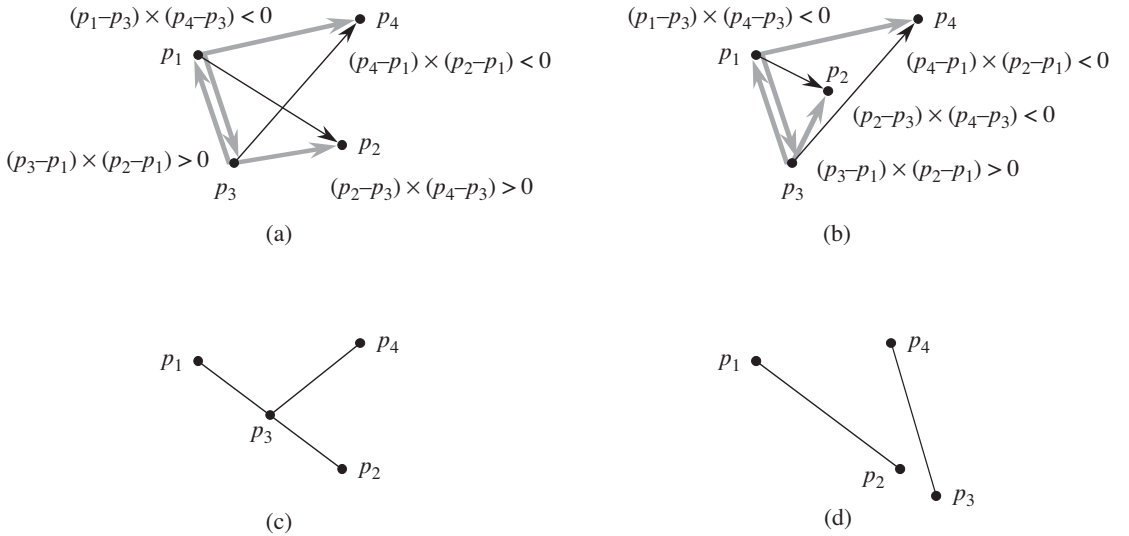


Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point p_3 is colinear with $\overline{p_1 p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is colinear with $\overline{p_1 p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

each other's lines, although a boundary case may apply. If all the relative orientations are nonzero, no boundary case applies. All the tests against 0 in lines 7–13 then fail, and SEGMENTS-INTERSECT returns FALSE in line 15. Figure 33.3(b) shows this case.

A boundary case occurs if any relative orientation d_k is 0. Here, we know that p_k is colinear with the other segment. It is directly on the other segment if and only if it is between the endpoints of the other segment. The procedure ON-SEGMENT returns whether p_k is between the endpoints of segment $\overline{p_i p_j}$, which will be the other segment when called in lines 7–13; the procedure assumes that p_k is colinear with segment $\overline{p_i p_j}$. Figures 33.3(c) and (d) show cases with colinear points. In Figure 33.3(c), p_3 is on $\overline{p_1 p_2}$, and so SEGMENTS-INTERSECT returns TRUE in line 12. No endpoints are on other segments in Figure 33.3(d), and so SEGMENTS-INTERSECT returns FALSE in line 15.

Other applications of cross products

Later sections of this chapter introduce additional uses for cross products. In Section 33.3, we shall need to sort a set of points according to their polar angles with respect to a given origin. As Exercise 33.1-3 asks you to show, we can use cross products to perform the comparisons in the sorting procedure. In Section 33.2, we shall use red-black trees to maintain the vertical ordering of a set of line segments. Rather than keeping explicit key values which we compare to each other in the red-black tree code, we shall compute a cross-product to determine which of two segments that intersect a given vertical line is above the other.

Exercises

33.1-1
Prove that if $p_1 \times p_2$ is positive, then vector p_1 is clockwise from vector p_2 with respect to the origin $(0, 0)$ and that if this cross product is negative, then p_1 is counterclockwise from p_2.

33.1-2
Professor van Pelt proposes that only the x-dimension needs to be tested in line 1 of ON-SEGMENT. Show why the professor is wrong.

33.1-3
The ***polar angle*** of a point p_1 with respect to an origin point p_0 is the angle of the vector $p_1 - p_0$ in the usual polar coordinate system. For example, the polar angle of $(3, 5)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 45 degrees or $\pi/4$ radians. The polar angle of $(3, 3)$ with respect to $(2, 4)$ is the angle of the vector $(1, -1)$, which is 315 degrees or $7\pi/4$ radians. Write pseudocode to sort a sequence $\langle p_1, p_2, \ldots, p_n \rangle$ of n points according to their polar angles with respect to a given origin point p_0. Your procedure should take $O(n \lg n)$ time and use cross products to compare angles.

33.1-4
Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are colinear.

33.1-5
A ***polygon*** is a piecewise-linear, closed curve in the plane. That is, it is a curve ending on itself that is formed by a sequence of straight-line segments, called the ***sides*** of the polygon. A point joining two consecutive sides is a ***vertex*** of the polygon. If the polygon is ***simple***, as we shall generally assume, it does not cross itself. The set of points in the plane enclosed by a simple polygon forms the ***interior*** of

the polygon, the set of points on the polygon itself forms its ***boundary***, and the set of points surrounding the polygon forms its ***exterior***. A simple polygon is ***convex*** if, given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior. A vertex of a convex polygon cannot be expressed as a convex combination of any two distinct points on the boundary or in the interior of the polygon.

Professor Amundsen proposes the following method to determine whether a sequence $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ of n points forms the consecutive vertices of a convex polygon. Output "yes" if the set $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1\}$, where subscript addition is performed modulo n, does not contain both left turns and right turns; otherwise, output "no." Show that although this method runs in linear time, it does not always produce the correct answer. Modify the professor's method so that it always produces the correct answer in linear time.

33.1-6

Given a point $p_0 = (x_0, y_0)$, the ***right horizontal ray*** from p_0 is the set of points $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ and } y_i = y_0\}$, that is, it is the set of points due right of p_0 along with p_0 itself. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{p_1 p_2}$ in $O(1)$ time by reducing the problem to that of determining whether two line segments intersect.

33.1-7

One way to determine whether a point p_0 is in the interior of a simple, but not necessarily convex, polygon P is to look at any ray from p_0 and check that the ray intersects the boundary of P an odd number of times but that p_0 itself is not on the boundary of P. Show how to compute in $\Theta(n)$ time whether a point p_0 is in the interior of an n-vertex polygon P. (*Hint:* Use Exercise 33.1-6. Make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.)

33.1-8

Show how to compute the area of an n-vertex simple, but not necessarily convex, polygon in $\Theta(n)$ time. (See Exercise 33.1-5 for definitions pertaining to polygons.)

33.2 Determining whether any pair of segments intersects

This section presents an algorithm for determining whether any two line segments in a set of segments intersect. The algorithm uses a technique known as "sweeping," which is common to many computational-geometry algorithms. Moreover, as

the exercises at the end of this section show, this algorithm, or simple variations of it, can help solve other computational-geometry problems.

The algorithm runs in $O(n \lg n)$ time, where n is the number of segments we are given. It determines only whether or not any intersection exists; it does not print all the intersections. (By Exercise 33.2-1, it takes $\Omega(n^2)$ time in the worst case to find *all* the intersections in a set of n line segments.)

In ***sweeping***, an imaginary vertical ***sweep line*** passes through the given set of geometric objects, usually from left to right. We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time. Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them. The line-segment-intersection algorithm in this section considers all the line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

To describe and prove correct our algorithm for determining whether any two of n line segments intersect, we shall make two simplifying assumptions. First, we assume that no input segment is vertical. Second, we assume that no three input segments intersect at a single point. Exercises 33.2-8 and 33.2-9 ask you to show that the algorithm is robust enough that it needs only a slight modification to work even when these assumptions do not hold. Indeed, removing such simplifying assumptions and dealing with boundary conditions often present the most difficult challenges when programming computational-geometry algorithms and proving their correctness.

Ordering segments

Because we assume that there are no vertical segments, we know that any input segment intersecting a given vertical sweep line intersects it at a single point. Thus, we can order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection.

To be more precise, consider two segments s_1 and s_2. We say that these segments are ***comparable*** at x if the vertical sweep line with x-coordinate x intersects both of them. We say that s_1 is ***above*** s_2 at x, written $s_1 \succcurlyeq_x s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line, or if s_1 and s_2 intersect at the sweep line. In Figure 33.4(a), for example, we have the relationships $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$, and $b \succcurlyeq_u c$. Segment d is not comparable with any other segment.

For any given x, the relation "$\succcurlyeq_x$" is a total preorder (see Section B.2) for all segments that intersect the sweep line at x. That is, the relation is transitive, and if segments s_1 and s_2 each intersect the sweep line at x, then either $s_1 \succcurlyeq_x s_2$ or $s_2 \succcurlyeq_x s_1$, or both (if s_1 and s_2 intersect at the sweep line). (The relation $\succcurlyeq_x$ is

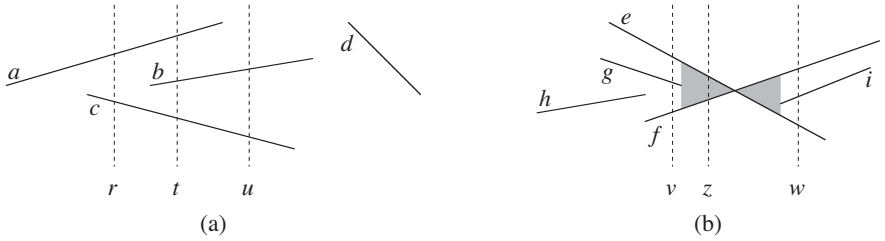


Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$, and $b \succcurlyeq_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, they reverse their orders: we have $e \succcurlyeq_v f$ but $f \succcurlyeq_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in the ordering given by the relation $\succcurlyeq_z$.

also reflexive, but neither symmetric nor antisymmetric.) The total preorder may differ for differing values of x, however, as segments enter and leave the ordering. A segment enters the ordering when its left endpoint is encountered by the sweep, and it leaves the ordering when its right endpoint is encountered.

What happens when the sweep line passes through the intersection of two segments? As Figure 33.4(b) shows, the segments reverse their positions in the total preorder. Sweep lines v and w are to the left and right, respectively, of the point of intersection of segments e and f, and we have $e \succcurlyeq_v f$ and $f \succcurlyeq_w e$. Note that because we assume that no three segments intersect at the same point, there must be some vertical sweep line x for which intersecting segments e and f are *consecutive* in the total preorder $\succcurlyeq_x$. Any sweep line that passes through the shaded region of Figure 33.4(b), such as z, has e and f consecutive in its total preorder.

Moving the sweep line

Sweeping algorithms typically manage two sets of data:

1. The ***sweep-line status*** gives the relationships among the objects that the sweep line intersects.
2. The ***event-point schedule*** is a sequence of points, called ***event points***, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point, and then resumes. Changes to the sweep-line status occur only at event points.

For some algorithms (the algorithm asked for in Exercise 33.2-7, for example), the event-point schedule develops dynamically as the algorithm progresses. The algorithm at hand, however, determines all the event points before the sweep, based

solely on simple properties of the input data. In particular, each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. (If two or more endpoints are ***covertical***, i.e., they have the same x-coordinate, we break the tie by putting all the covertical left endpoints before the covertical right endpoints. Within a set of covertical left endpoints, we put those with lower y-coordinates first, and we do the same within a set of covertical right endpoints.) When we encounter a segment's left endpoint, we insert the segment into the sweep-line status, and we delete the segment from the sweep-line status upon encountering its right endpoint. Whenever two segments first become consecutive in the total preorder, we check whether they intersect.

The sweep-line status is a total preorder T, for which we require the following operations:

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

It is possible for segments s_1 and s_2 to be mutually above each other in the total preorder T; this situation can occur if s_1 and s_2 intersect at the sweep line whose total preorder is given by T. In this case, the two segments may appear in either order in T.

If the input contains n segments, we can perform each of the operations INSERT, DELETE, ABOVE, and BELOW in $O(\lg n)$ time using red-black trees. Recall that the red-black-tree operations in Chapter 13 involve comparing keys. We can replace the key comparisons by comparisons that use cross products to determine the relative ordering of two segments (see Exercise 33.2-2).

Segment-intersection pseudocode

The following algorithm takes as input a set S of n line segments, returning the boolean value TRUE if any pair of segments in S intersects, and FALSE otherwise. A red-black tree maintains the total preorder T.

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
 3  for each point p in the sorted list of endpoints
 4      if p is the left endpoint of a segment s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
 7              return TRUE
 8      if p is the right endpoint of a segment s
 9          if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

Figure 33.5 illustrates how the algorithm works. Line 1 initializes the total preorder to be empty. Line 2 determines the event-point schedule by sorting the $2n$ segment endpoints from left to right, breaking ties as described above. One way to perform line 2 is by lexicographically sorting the endpoints on (x, e, y), where x and y are the usual coordinates, $e = 0$ for a left endpoint, and $e = 1$ for a right endpoint.

Each iteration of the **for** loop of lines 3–11 processes one event point p. If p is the left endpoint of a segment s, line 5 adds s to the total preorder, and lines 6–7 return TRUE if s intersects either of the segments it is consecutive with in the total preorder defined by the sweep line passing through p. (A boundary condition occurs if p lies on another segment s'. In this case, we require only that s and s' be placed consecutively into T.) If p is the right endpoint of a segment s, then we need to delete s from the total preorder. But first, lines 9–10 return TRUE if there is an intersection between the segments surrounding s in the total preorder defined by the sweep line passing through p. If these segments do not intersect, line 11 deletes segment s from the total preorder. If the segments surrounding segment s intersect, they would have become consecutive after deleting s had the **return** statement in line 10 not prevented line 11 from executing. The correctness argument, which follows, will make it clear why it suffices to check the segments surrounding s. Finally, if we never find any intersections after having processed all $2n$ event points, line 12 returns FALSE.

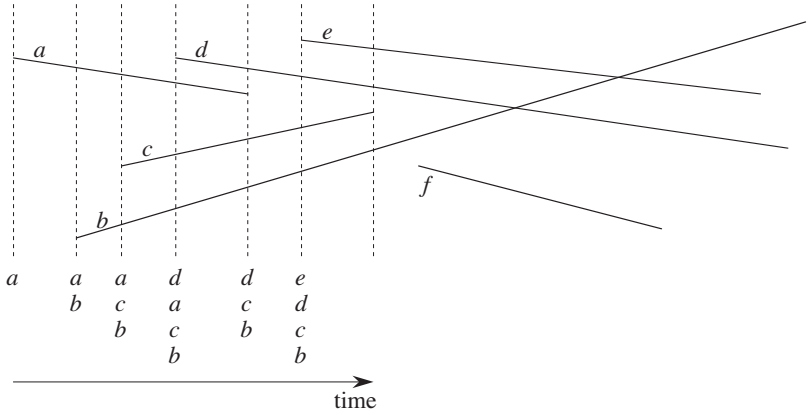


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep line corresponds to the total preorder T at the end of the **for** loop processing the corresponding event point. The rightmost sweep line occurs when processing the right endpoint of segment c; because segments d and b surround c and intersect each other, the procedure returns TRUE.

Correctness

To show that ANY-SEGMENTS-INTERSECT is correct, we will prove that the call ANY-SEGMENTS-INTERSECT(S) returns TRUE if and only if there is an intersection among the segments in S.

It is easy to see that ANY-SEGMENTS-INTERSECT returns TRUE (on lines 7 and 10) only if it finds an intersection between two of the input segments. Hence, if it returns TRUE, there is an intersection.

We also need to show the converse: that if there is an intersection, then ANY-SEGMENTS-INTERSECT returns TRUE. Let us suppose that there is at least one intersection. Let p be the leftmost intersection point, breaking ties by choosing the point with the lowest y-coordinate, and let a and b be the segments that intersect at p. Since no intersections occur to the left of p, the order given by T is correct at all points to the left of p. Because no three segments intersect at the same point, a and b become consecutive in the total preorder at some sweep line z.[2] Moreover, z is to the left of p or goes through p. Some segment endpoint q on sweep line z

[2]If we allow three segments to intersect at the same point, there may be an intervening segment c that intersects both a and b at point p. That is, we may have $a \succcurlyeq_w c$ and $c \succcurlyeq_w b$ for all sweep lines w to the left of p for which $a \succcurlyeq_w b$. Exercise 33.2-8 asks you to show that ANY-SEGMENTS-INTERSECT is correct even if three segments do intersect at the same point.

is the event point at which a and b become consecutive in the total preorder. If p is on sweep line z, then $q = p$. If p is not on sweep line z, then q is to the left of p. In either case, the order given by T is correct just before encountering q. (Here is where we use the lexicographic order in which the algorithm processes event points. Because p is the lowest of the leftmost intersection points, even if p is on sweep line z and some other intersection point p' is on z, event point $q = p$ is processed before the other intersection p' can interfere with the total preorder T. Moreover, even if p is the left endpoint of one segment, say a, and the right endpoint of the other segment, say b, because left endpoint events occur before right endpoint events, segment b is in T upon first encountering segment a.) Either event point q is processed by ANY-SEGMENTS-INTERSECT or it is not processed.

If q is processed by ANY-SEGMENTS-INTERSECT, only two possible actions may occur:

1. Either a or b is inserted into T, and the other segment is above or below it in the total preorder. Lines 4–7 detect this case.
2. Segments a and b are already in T, and a segment between them in the total preorder is deleted, making a and b become consecutive. Lines 8–11 detect this case.

In either case, we find the intersection p and ANY-SEGMENTS-INTERSECT returns TRUE.

If event point q is not processed by ANY-SEGMENTS-INTERSECT, the procedure must have returned before processing all event points. This situation could have occurred only if ANY-SEGMENTS-INTERSECT had already found an intersection and returned TRUE.

Thus, if there is an intersection, ANY-SEGMENTS-INTERSECT returns TRUE. As we have already seen, if ANY-SEGMENTS-INTERSECT returns TRUE, there is an intersection. Therefore, ANY-SEGMENTS-INTERSECT always returns a correct answer.

Running time

If set S contains n segments, then ANY-SEGMENTS-INTERSECT runs in time $O(n \lg n)$. Line 1 takes $O(1)$ time. Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort. The **for** loop of lines 3–11 iterates at most once per event point, and so with $2n$ event points, the loop iterates at most $2n$ times. Each iteration takes $O(\lg n)$ time, since each red-black-tree operation takes $O(\lg n)$ time and, using the method of Section 33.1, each intersection test takes $O(1)$ time. The total time is thus $O(n \lg n)$.

Exercises

33.2-1
Show that a set of n line segments may contain $\Theta(n^2)$ intersections.

33.2-2
Given two segments a and b that are comparable at x, show how to determine in $O(1)$ time which of $a \succcurlyeq_x b$ or $b \succcurlyeq_x a$ holds. Assume that neither segment is vertical. (*Hint:* If a and b do not intersect, you can just use cross products. If a and b intersect—which you can of course determine using only cross products—you can still use only addition, subtraction, and multiplication, avoiding division. Of course, in the application of the $\succcurlyeq_x$ relation used here, if a and b intersect, we can just stop and declare that we have found an intersection.)

33.2-3
Professor Mason suggests that we modify ANY-SEGMENTS-INTERSECT so that instead of returning upon finding an intersection, it prints the segments that intersect and continues on to the next iteration of the **for** loop. The professor calls the resulting procedure PRINT-INTERSECTING-SEGMENTS and claims that it prints all intersections, from left to right, as they occur in the set of line segments. Professor Dixon disagrees, claiming that Professor Mason's idea is incorrect. Which professor is right? Will PRINT-INTERSECTING-SEGMENTS always find the leftmost intersection first? Will it always find all the intersections?

33.2-4
Give an $O(n \lg n)$-time algorithm to determine whether an n-vertex polygon is simple.

33.2-5
Give an $O(n \lg n)$-time algorithm to determine whether two simple polygons with a total of n vertices intersect.

33.2-6
A ***disk*** consists of a circle plus its interior and is represented by its center point and radius. Two disks intersect if they have any point in common. Give an $O(n \lg n)$-time algorithm to determine whether any two disks in a set of n intersect.

33.2-7
Given a set of n line segments containing a total of k intersections, show how to output all k intersections in $O((n + k) \lg n)$ time.

33.2-8
Argue that ANY-SEGMENTS-INTERSECT works correctly even if three or more segments intersect at the same point.

33.2-9
Show that ANY-SEGMENTS-INTERSECT works correctly in the presence of vertical segments if we treat the bottom endpoint of a vertical segment as if it were a left endpoint and the top endpoint as if it were a right endpoint. How does your answer to Exercise 33.2-2 change if we allow vertical segments?

33.3 Finding the convex hull

The ***convex hull*** of a set Q of points, denoted by $\mathrm{CH}(Q)$, is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. (See Exercise 33.1-5 for a precise definition of a convex polygon.) We implicitly assume that all points in the set Q are unique and that Q contains at least three points which are not colinear. Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails. Figure 33.6 shows a set of points and its convex hull.

In this section, we shall present two algorithms that compute the convex hull of a set of n points. Both algorithms output the vertices of the convex hull in counterclockwise order. The first, known as Graham's scan, runs in $O(n \lg n)$ time. The second, called Jarvis's march, runs in $O(nh)$ time, where h is the number of vertices of the convex hull. As Figure 33.6 illustrates, every vertex of $\mathrm{CH}(Q)$ is a

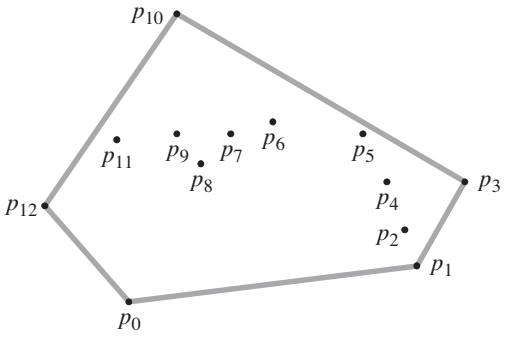


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull $\mathrm{CH}(Q)$ in gray.

point in Q. Both algorithms exploit this property, deciding which vertices in Q to keep as vertices of the convex hull and which vertices in Q to reject.

We can compute convex hulls in $O(n \lg n)$ time by any one of several methods. Both Graham's scan and Jarvis's march use a technique called "rotational sweep," processing vertices in the order of the polar angles they form with a reference vertex. Other methods include the following:

- In the ***incremental method***, we first sort the points from left to right, yielding a sequence $\langle p_1, p_2, \ldots, p_n \rangle$. At the ith stage, we update the convex hull of the $i - 1$ leftmost points, $\text{CH}(\{p_1, p_2, \ldots, p_{i-1}\})$, according to the ith point from the left, thus forming $\text{CH}(\{p_1, p_2, \ldots, p_i\})$. Exercise 33.3-6 asks you how to implement this method to take a total of $O(n \lg n)$ time.
- In the ***divide-and-conquer method***, we divide the set of n points in $\Theta(n)$ time into two subsets, one containing the leftmost $\lceil n/2 \rceil$ points and one containing the rightmost $\lfloor n/2 \rfloor$ points, recursively compute the convex hulls of the subsets, and then, by means of a clever method, combine the hulls in $O(n)$ time. The running time is described by the familiar recurrence $T(n) = 2T(n/2) + O(n)$, and so the divide-and-conquer method runs in $O(n \lg n)$ time.
- The ***prune-and-search method*** is similar to the worst-case linear-time median algorithm of Section 9.3. With this method, we find the upper portion (or "upper chain") of the convex hull by repeatedly throwing out a constant fraction of the remaining points until only the upper chain of the convex hull remains. We then do the same for the lower chain. This method is asymptotically the fastest: if the convex hull contains h vertices, it runs in only $O(n \lg h)$ time.

Computing the convex hull of a set of points is an interesting problem in its own right. Moreover, algorithms for some other computational-geometry problems start by computing a convex hull. Consider, for example, the two-dimensional ***farthest-pair problem***: we are given a set of n points in the plane and wish to find the two points whose distance from each other is maximum. As Exercise 33.3-3 asks you to prove, these two points must be vertices of the convex hull. Although we won't prove it here, we can find the farthest pair of vertices of an n-vertex convex polygon in $O(n)$ time. Thus, by computing the convex hull of the n input points in $O(n \lg n)$ time and then finding the farthest pair of the resulting convex-polygon vertices, we can find the farthest pair of points in any set of n points in $O(n \lg n)$ time.

Graham's scan

Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. It pushes each point of the input set Q onto the stack one time,

and it eventually pops from the stack each point that is not a vertex of $\mathrm{CH}(Q)$. When the algorithm terminates, stack S contains exactly the vertices of $\mathrm{CH}(Q)$, in counterclockwise order of their appearance on the boundary.

The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$. It calls the functions TOP(S), which returns the point on top of stack S without changing S, and NEXT-TO-TOP(S), which returns the point one entry below the top of stack S without changing S. As we shall prove in a moment, the stack S returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices of $\mathrm{CH}(Q)$ in counterclockwise order.

```
GRAHAM-SCAN(Q)
 1  let p_0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
 2  let ⟨p_1, p_2, ..., p_m⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p_0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p_0)
 3  let S be an empty stack
 4  PUSH(p_0, S)
 5  PUSH(p_1, S)
 6  PUSH(p_2, S)
 7  for i = 3 to m
 8      while the angle formed by points NEXT-TO-TOP(S), TOP(S),
              and p_i makes a nonleft turn
 9          POP(S)
10      PUSH(p_i, S)
11  return S
```

Figure 33.7 illustrates the progress of GRAHAM-SCAN. Line 1 chooses point p_0 as the point with the lowest y-coordinate, picking the leftmost such point in case of a tie. Since there is no point in Q that is below p_0 and any other points with the same y-coordinate are to its right, p_0 must be a vertex of $\mathrm{CH}(Q)$. Line 2 sorts the remaining points of Q by polar angle relative to p_0, using the same method—comparing cross products—as in Exercise 33.1-3. If two or more points have the same polar angle relative to p_0, all but the farthest such point are convex combinations of p_0 and the farthest point, and so we remove them entirely from consideration. We let m denote the number of points other than p_0 that remain. The polar angle, measured in radians, of each point in Q relative to p_0 is in the half-open interval $[0, \pi)$. Since the points are sorted according to polar angles, they are sorted in counterclockwise order relative to p_0. We designate this sorted sequence of points by $\langle p_1, p_2, \ldots, p_m \rangle$. Note that points p_1 and p_m are vertices

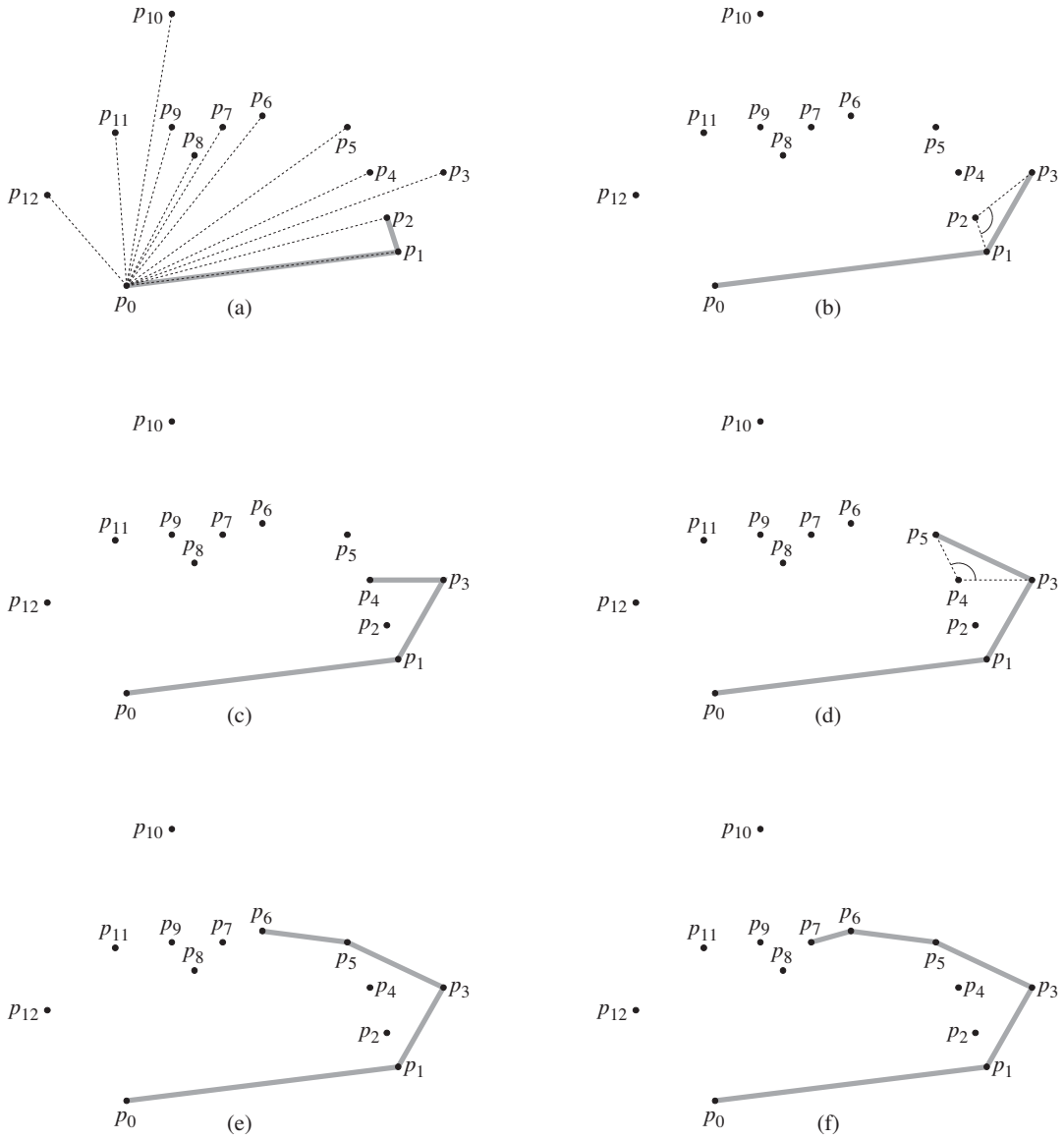


Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped.

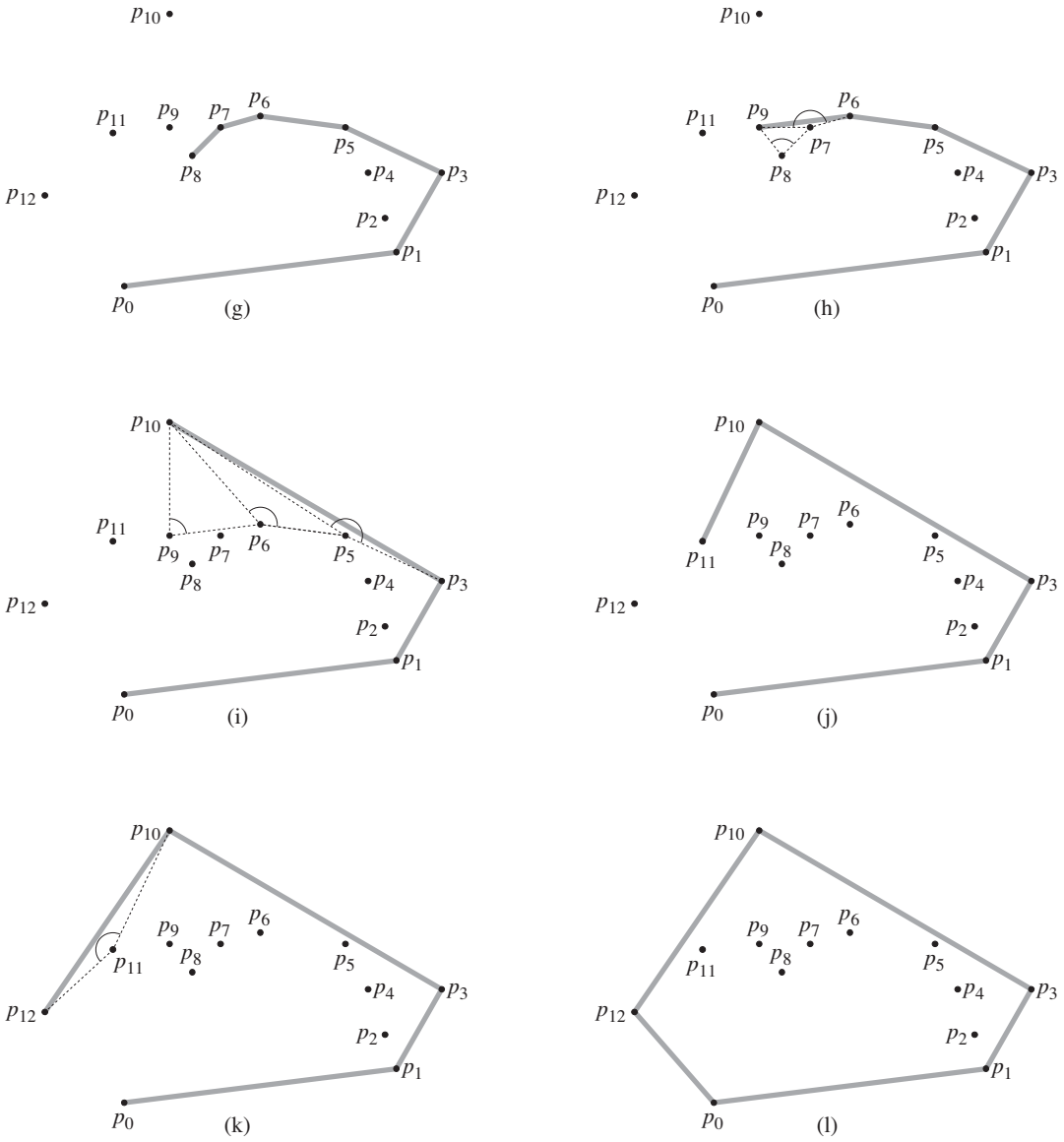


Figure 33.7, continued **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.

of CH(Q) (see Exercise 33.3-1). Figure 33.7(a) shows the points of Figure 33.6 sequentially numbered in order of increasing polar angle relative to p_0.

The remainder of the procedure uses the stack S. Lines 3–6 initialize the stack to contain, from bottom to top, the first three points p_0, p_1, and p_2. Figure 33.7(a) shows the initial stack S. The **for** loop of lines 7–10 iterates once for each point in the subsequence $\langle p_3, p_4, \ldots, p_m \rangle$. We shall see that after processing point p_i, stack S contains, from bottom to top, the vertices of CH($\{p_0, p_1, \ldots, p_i\}$) in counterclockwise order. The **while** loop of lines 8–9 removes points from the stack if we find them not to be vertices of the convex hull. When we traverse the convex hull counterclockwise, we should make a left turn at each vertex. Thus, each time the **while** loop finds a vertex at which we make a nonleft turn, we pop the vertex from the stack. (By checking for a nonleft turn, rather than just a right turn, this test precludes the possibility of a straight angle at a vertex of the resulting convex hull. We want no straight angles, since no vertex of a convex polygon may be a convex combination of other vertices of the polygon.) After we pop all vertices that have nonleft turns when heading toward point p_i, we push p_i onto the stack. Figures 33.7(b)–(k) show the state of the stack S after each iteration of the **for** loop. Finally, GRAHAM-SCAN returns the stack S in line 11. Figure 33.7(l) shows the corresponding convex hull.

The following theorem formally proves the correctness of GRAHAM-SCAN.

Theorem 33.1 (Correctness of Graham's scan)
If GRAHAM-SCAN executes on a set Q of points, where $|Q| \geq 3$, then at termination, the stack S consists of, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

Proof After line 2, we have the sequence of points $\langle p_1, p_2, \ldots, p_m \rangle$. Let us define, for $i = 2, 3, \ldots, m$, the subset of points $Q_i = \{p_0, p_1, \ldots, p_i\}$. The points in $Q - Q_m$ are those that were removed because they had the same polar angle relative to p_0 as some point in Q_m; these points are not in CH(Q), and so CH(Q_m) = CH(Q). Thus, it suffices to show that when GRAHAM-SCAN terminates, the stack S consists of the vertices of CH(Q_m) in counterclockwise order, when listed from bottom to top. Note that just as p_0, p_1, and p_m are vertices of CH(Q), the points p_0, p_1, and p_i are all vertices of CH(Q_i).

The proof uses the following loop invariant:

> At the start of each iteration of the **for** loop of lines 7–10, stack S consists of, from bottom to top, exactly the vertices of CH(Q_{i-1}) in counterclockwise order.

Initialization: The invariant holds the first time we execute line 7, since at that time, stack S consists of exactly the vertices of $Q_2 = Q_{i-1}$, and this set of three

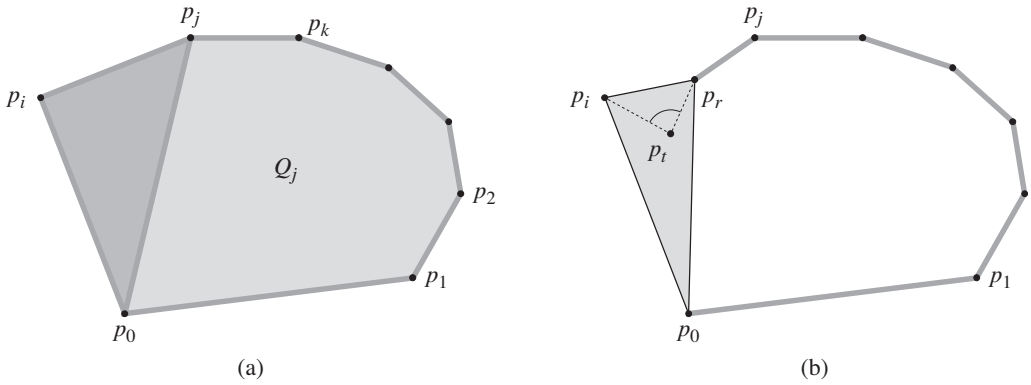


Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $\mathrm{CH}(Q_j)$ gives exactly the vertices of $\mathrm{CH}(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, which means that it cannot be a vertex of $\mathrm{CH}(Q_i)$.

vertices forms its own convex hull. Moreover, they appear in counterclockwise order from bottom to top.

Maintenance: Entering an iteration of the **for** loop, the top point on stack S is p_{i-1}, which was pushed at the end of the previous iteration (or before the first iteration, when $i = 3$). Let p_j be the top point on S after executing the while loop of lines 8–9 but before line 10 pushes p_i, and let p_k be the point just below p_j on S. At the moment that p_j is the top point on S and we have not yet pushed p_i, stack S contains exactly the same points it contained after iteration j of the **for** loop. By the loop invariant, therefore, S contains exactly the vertices of $\mathrm{CH}(Q_j)$ at that moment, and they appear in counterclockwise order from bottom to top.

Let us continue to focus on this moment just before pushing p_i. We know that p_i's polar angle relative to p_0 is greater than p_j's polar angle and that the angle $\angle p_k p_j p_i$ makes a left turn (otherwise we would have popped p_j). Therefore, because S contains exactly the vertices of $\mathrm{CH}(Q_j)$, we see from Figure 33.8(a) that once we push p_i, stack S will contain exactly the vertices of $\mathrm{CH}(Q_j \cup \{p_i\})$, still in counterclockwise order from bottom to top.

We now show that $\mathrm{CH}(Q_j \cup \{p_i\})$ is the same set of points as $\mathrm{CH}(Q_i)$. Consider any point p_t that was popped during iteration i of the **for** loop, and let p_r be the point just below p_t on stack S at the time p_t was popped (p_r might be p_j). The angle $\angle p_r p_t p_i$ makes a nonleft turn, and the polar angle of p_t relative to p_0 is greater than the polar angle of p_r. As Figure 33.8(b) shows, p_t must

be either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of this triangle (but it is not a vertex of the triangle). Clearly, since p_t is within a triangle formed by three other points of Q_i, it cannot be a vertex of $\mathrm{CH}(Q_i)$. Since p_t is not a vertex of $\mathrm{CH}(Q_i)$, we have that

$$\mathrm{CH}(Q_i - \{p_t\}) = \mathrm{CH}(Q_i) \; . \tag{33.1}$$

Let P_i be the set of points that were popped during iteration i of the **for** loop. Since the equality (33.1) applies for all points in P_i, we can apply it repeatedly to show that $\mathrm{CH}(Q_i - P_i) = \mathrm{CH}(Q_i)$. But $Q_i - P_i = Q_j \cup \{p_i\}$, and so we conclude that $\mathrm{CH}(Q_j \cup \{p_i\}) = \mathrm{CH}(Q_i - P_i) = \mathrm{CH}(Q_i)$.

We have shown that once we push p_i, stack S contains exactly the vertices of $\mathrm{CH}(Q_i)$ in counterclockwise order from bottom to top. Incrementing i will then cause the loop invariant to hold for the next iteration.

Termination: When the loop terminates, we have $i = m + 1$, and so the loop invariant implies that stack S consists of exactly the vertices of $\mathrm{CH}(Q_m)$, which is $\mathrm{CH}(Q)$, in counterclockwise order from bottom to top. This completes the proof. ■

We now show that the running time of GRAHAM-SCAN is $O(n \lg n)$, where $n = |Q|$. Line 1 takes $\Theta(n)$ time. Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort to sort the polar angles and the cross-product method of Section 33.1 to compare angles. (We can remove all but the farthest point with the same polar angle in total of $O(n)$ time over all n points.) Lines 3–6 take $O(1)$ time. Because $m \le n - 1$, the **for** loop of lines 7–10 executes at most $n - 3$ times. Since PUSH takes $O(1)$ time, each iteration takes $O(1)$ time exclusive of the time spent in the **while** loop of lines 8–9, and thus overall the **for** loop takes $O(n)$ time exclusive of the nested **while** loop.

We use aggregate analysis to show that the **while** loop takes $O(n)$ time overall. For $i = 0, 1, \ldots, m$, we push each point p_i onto stack S exactly once. As in the analysis of the MULTIPOP procedure of Section 17.1, we observe that we can pop at most the number of items that we push. At least three points — p_0, p_1, and p_m — are never popped from the stack, so that in fact at most $m - 2$ POP operations are performed in total. Each iteration of the **while** loop performs one POP, and so there are at most $m - 2$ iterations of the **while** loop altogether. Since the test in line 8 takes $O(1)$ time, each call of POP takes $O(1)$ time, and $m \le n - 1$, the total time taken by the **while** loop is $O(n)$. Thus, the running time of GRAHAM-SCAN is $O(n \lg n)$.

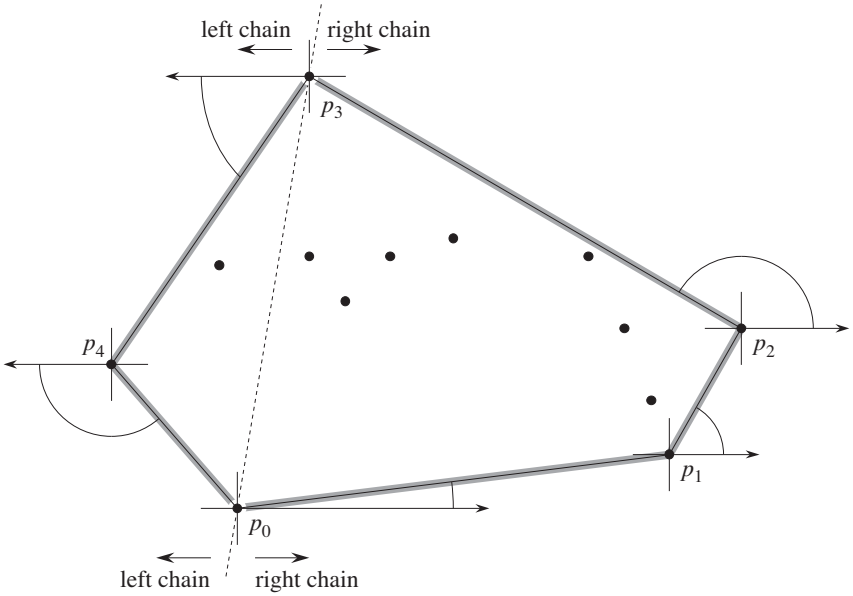


Figure 33.9 The operation of Jarvis's march. We choose the first vertex as the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, we construct the left chain by finding smallest polar angles with respect to the negative x-axis.

Jarvis's march

Jarvis's march computes the convex hull of a set Q of points by a technique known as ***package wrapping*** (or ***gift wrapping***). The algorithm runs in time $O(nh)$, where h is the number of vertices of $\mathrm{CH}(Q)$. When h is $o(\lg n)$, Jarvis's march is asymptotically faster than Graham's scan.

Intuitively, Jarvis's march simulates wrapping a taut piece of paper around the set Q. We start by taping the end of the paper to the lowest point in the set, that is, to the same point p_0 with which we start Graham's scan. We know that this point must be a vertex of the convex hull. We pull the paper to the right to make it taut, and then we pull it higher until it touches a point. This point must also be a vertex of the convex hull. Keeping the paper taut, we continue in this way around the set of vertices until we come back to our original point p_0.

More formally, Jarvis's march builds a sequence $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ of the vertices of $\mathrm{CH}(Q)$. We start with p_0. As Figure 33.9 shows, the next vertex p_1 in the convex hull has the smallest polar angle with respect to p_0. (In case of ties, we choose the point farthest from p_0.) Similarly, p_2 has the smallest polar angle

with respect to p_1, and so on. When we reach the highest vertex, say p_k (breaking ties by choosing the farthest such vertex), we have constructed, as Figure 33.9 shows, the ***right chain*** of $\text{CH}(Q)$. To construct the ***left chain***, we start at p_k and choose p_{k+1} as the point with the smallest polar angle with respect to p_k, but *from the negative x-axis*. We continue on, forming the left chain by taking polar angles from the negative x-axis, until we come back to our original vertex p_0.

We could implement Jarvis's march in one conceptual sweep around the convex hull, that is, without separately constructing the right and left chains. Such implementations typically keep track of the angle of the last convex-hull side chosen and require the sequence of angles of hull sides to be strictly increasing (in the range of 0 to 2π radians). The advantage of constructing separate chains is that we need not explicitly compute angles; the techniques of Section 33.1 suffice to compare angles.

If implemented properly, Jarvis's march has a running time of $O(nh)$. For each of the h vertices of $\text{CH}(Q)$, we find the vertex with the minimum polar angle. Each comparison between polar angles takes $O(1)$ time, using the techniques of Section 33.1. As Section 9.1 shows, we can compute the minimum of n values in $O(n)$ time if each comparison takes $O(1)$ time. Thus, Jarvis's march takes $O(nh)$ time.

Exercises

33.3-1

Prove that in the procedure GRAHAM-SCAN, points p_1 and p_m must be vertices of $\text{CH}(Q)$.

33.3-2

Consider a model of computation that supports addition, comparison, and multiplication and for which there is a lower bound of $\Omega(n \lg n)$ to sort n numbers. Prove that $\Omega(n \lg n)$ is a lower bound for computing, in order, the vertices of the convex hull of a set of n points in such a model.

33.3-3

Given a set of points Q, prove that the pair of points farthest from each other must be vertices of $\text{CH}(Q)$.

33.3-4

For a given polygon P and a point q on its boundary, the ***shadow*** of q is the set of points r such that the segment $\overline{qr}$ is entirely on the boundary or in the interior of P. As Figure 33.10 illustrates, a polygon P is ***star-shaped*** if there exists a point p in the interior of P that is in the shadow of every point on the boundary of P. The set of all such points p is called the ***kernel*** of P. Given an n-vertex,

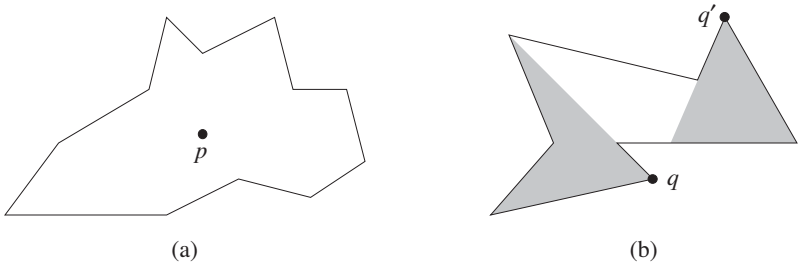


Figure 33.10 The definition of a star-shaped polygon, for use in Exercise 33.3-4. **(a)** A star-shaped polygon. The segment from point p to any point q on the boundary intersects the boundary only at q. **(b)** A non-star-shaped polygon. The shaded region on the left is the shadow of q, and the shaded region on the right is the shadow of q'. Since these regions are disjoint, the kernel is empty.

star-shaped polygon P specified by its vertices in counterclockwise order, show how to compute $\mathrm{CH}(P)$ in $O(n)$ time.

33.3-5

In the ***on-line convex-hull problem***, we are given the set Q of n points one point at a time. After receiving each point, we compute the convex hull of the points seen so far. Obviously, we could run Graham's scan once for each point, with a total running time of $O(n^2 \lg n)$. Show how to solve the on-line convex-hull problem in a total of $O(n^2)$ time.

33.3-6 ⋆

Show how to implement the incremental method for computing the convex hull of n points so that it runs in $O(n \lg n)$ time.

33.4 Finding the closest pair of points

We now consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points. "Closest" refers to the usual euclidean distance: the distance between points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Two points in set Q may be coincident, in which case the distance between them is zero. This problem has applications in, for example, traffic-control systems. A system for controlling air or sea traffic might need to identify the two closest vehicles in order to detect potential collisions.

A brute-force closest-pair algorithm simply looks at all the $\binom{n}{2} = \Theta(n^2)$ pairs of points. In this section, we shall describe a divide-and-conquer algorithm for

this problem, whose running time is described by the familiar recurrence $T(n) = 2T(n/2) + O(n)$. Thus, this algorithm uses only $O(n \lg n)$ time.

The divide-and-conquer algorithm

Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrays X and Y, each of which contains all the points of the input subset P. The points in array X are sorted so that their x-coordinates are monotonically increasing. Similarly, array Y is sorted by monotonically increasing y-coordinate. Note that in order to attain the $O(n \lg n)$ time bound, we cannot afford to sort in each recursive call; if we did, the recurrence for the running time would be $T(n) = 2T(n/2) + O(n \lg n)$, whose solution is $T(n) = O(n \lg^2 n)$. (Use the version of the master method given in Exercise 4.6-2.) We shall see a little later how to use "presorting" to maintain this sorted property without actually sorting in each recursive call.

A given recursive invocation with inputs P, X, and Y first checks whether $|P| \leq 3$. If so, the invocation simply performs the brute-force method described above: try all $\binom{|P|}{2}$ pairs of points and return the closest pair. If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows.

Divide: Find a vertical line l that bisects the point set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points in P_L are on or to the left of line l, and all points in P_R are on or to the right of l. Divide the array X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide the array Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.

Conquer: Having divided P into P_L and P_R, make two recursive calls, one to find the closest pair of points in P_L and the other to find the closest pair of points in P_R. The inputs to the first call are the subset P_L and arrays X_L and Y_L; the second call receives the inputs P_R, X_R, and Y_R. Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.

Combine: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R. The algorithm determines whether there is a pair with one point in P_L and the other point in P_R and whose distance is less than δ. Observe that if a pair of points has distance less than δ, both points of the pair must be within δ units of line l. Thus, as Figure 33.11(a) shows, they both must reside in the 2δ-wide vertical strip centered at line l. To find such a pair, if one exists, we do the following:

1. Create an array Y', which is the array Y with all points not in the 2δ-wide vertical strip removed. The array Y' is sorted by y-coordinate, just as Y is.
2. For each point p in the array Y', try to find points in Y' that are within δ units of p. As we shall see shortly, only the 7 points in Y' that follow p need be considered. Compute the distance from p to each of these 7 points, and keep track of the closest-pair distance δ' found over all pairs of points in Y'.
3. If $\delta' < \delta$, then the vertical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ found by the recursive calls.

The above description omits some implementation details that are necessary to achieve the $O(n \lg n)$ running time. After proving the correctness of the algorithm, we shall show how to implement the algorithm to achieve the desired time bound.

Correctness

The correctness of this closest-pair algorithm is obvious, except for two aspects. First, by bottoming out the recursion when $|P| \leq 3$, we ensure that we never try to solve a subproblem consisting of only one point. The second aspect is that we need only check the 7 points following each point p in array Y'; we shall now prove this property.

Suppose that at some level of the recursion, the closest pair of points is $p_L \in P_L$ and $p_R \in P_R$. Thus, the distance δ' between p_L and p_R is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away. Similarly, p_R is on or to the right of l and less than δ units away. Moreover, p_L and p_R are within δ units of each other vertically. Thus, as Figure 33.11(a) shows, p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at line l. (There may be other points within this rectangle as well.)

We next show that at most 8 points of P can reside within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square forming the left half of this rectangle. Since all points within P_L are at least δ units apart, at most 4 points can reside within this square; Figure 33.11(b) shows how. Similarly, at most 4 points in P_R can reside within the $\delta \times \delta$ square forming the right half of the rectangle. Thus, at most 8 points of P can reside within the $\delta \times 2\delta$ rectangle. (Note that since points on line l may be in either P_L or P_R, there may be up to 4 points on l. This limit is achieved if there are two pairs of coincident points such that each pair consists of one point from P_L and one point from P_R, one pair is at the intersection of l and the top of the rectangle, and the other pair is where l intersects the bottom of the rectangle.)

Having shown that at most 8 points of P can reside within the rectangle, we can easily see why we need to check only the 7 points following each point in the array Y'. Still assuming that the closest pair is p_L and p_R, let us assume without

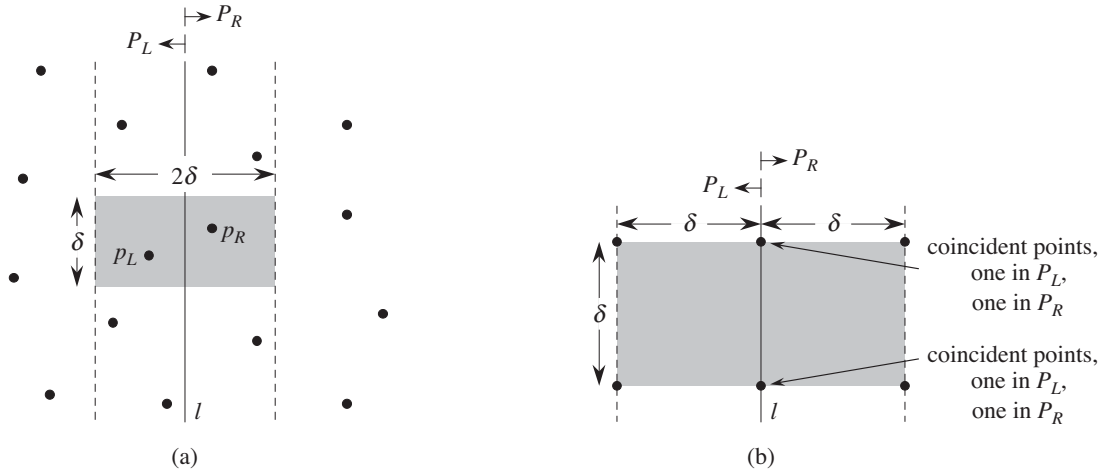


Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ units apart can all reside within a $\delta \times \delta$ square. On the left are 4 points in P_L, and on the right are 4 points in P_R. The $\delta \times 2\delta$ rectangle can contain 8 points if the points shown on line l are actually pairs of coincident points with one point in P_L and one in P_R.

loss of generality that p_L precedes p_R in array Y'. Then, even if p_L occurs as early as possible in Y' and p_R occurs as late as possible, p_R is in one of the 7 positions following p_L. Thus, we have shown the correctness of the closest-pair algorithm.

Implementation and running time

As we have noted, our goal is to have the recurrence for the running time be $T(n) = 2T(n/2) + O(n)$, where $T(n)$ is the running time for a set of n points. The main difficulty comes from ensuring that the arrays X_L, X_R, Y_L, and Y_R, which are passed to recursive calls, are sorted by the proper coordinate and also that the array Y' is sorted by y-coordinate. (Note that if the array X that is received by a recursive call is already sorted, then we can easily divide set P into P_L and P_R in linear time.)

The key observation is that in each call, we wish to form a sorted subset of a sorted array. For example, a particular invocation receives the subset P and the array Y, sorted by y-coordinate. Having partitioned P into P_L and P_R, it needs to form the arrays Y_L and Y_R, which are sorted by y-coordinate, in linear time. We can view the method as the opposite of the MERGE procedure from merge sort in

Section 2.3.1: we are splitting a sorted array into two sorted arrays. The following pseudocode gives the idea.

```
1  let Y_L[1..Y.length] and Y_R[1..Y.length] be new arrays
2  Y_L.length = Y_R.length = 0
3  for i = 1 to Y.length
4      if Y[i] ∈ P_L
5          Y_L.length = Y_L.length + 1
6          Y_L[Y_L.length] = Y[i]
7      else Y_R.length = Y_R.length + 1
8          Y_R[Y_R.length] = Y[i]
```

We simply examine the points in array Y in order. If a point $Y[i]$ is in P_L, we append it to the end of array Y_L; otherwise, we append it to the end of array Y_R. Similar pseudocode works for forming arrays X_L, X_R, and Y'.

The only remaining question is how to get the points sorted in the first place. We ***presort*** them; that is, we sort them once and for all *before* the first recursive call. We pass these sorted arrays into the first recursive call, and from there we whittle them down through the recursive calls as necessary. Presorting adds an additional $O(n \lg n)$ term to the running time, but now each step of the recursion takes linear time exclusive of the recursive calls. Thus, if we let $T(n)$ be the running time of each recursive step and $T'(n)$ be the running time of the entire algorithm, we get $T'(n) = T(n) + O(n \lg n)$ and

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3\,, \\ O(1) & \text{if } n \le 3\,. \end{cases}$$

Thus, $T(n) = O(n \lg n)$ and $T'(n) = O(n \lg n)$.

Exercises

33.4-1

Professor Williams comes up with a scheme that allows the closest-pair algorithm to check only 5 points following each point in array Y'. The idea is always to place points on line l into set P_L. Then, there cannot be pairs of coincident points on line l with one point in P_L and one in P_R. Thus, at most 6 points can reside in the $\delta \times 2\delta$ rectangle. What is the flaw in the professor's scheme?

33.4-2

Show that it actually suffices to check only the points in the 5 array positions following each point in the array Y'.

33.4-3

We can define the distance between two points in ways other than euclidean. In the plane, the ***L_m-distance*** between points p_1 and p_2 is given by the expression $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Euclidean distance, therefore, is L_2-distance. Modify the closest-pair algorithm to use the L_1-distance, which is also known as the ***Manhattan distance***.

33.4-4

Given two points p_1 and p_2 in the plane, the L_∞-distance between them is given by $\max(|x_1 - x_2|, |y_1 - y_2|)$. Modify the closest-pair algorithm to use the L_∞-distance.

33.4-5

Suppose that $\Omega(n)$ of the points given to the closest-pair algorithm are covertical. Show how to determine the sets P_L and P_R and how to determine whether each point of Y is in P_L or P_R so that the running time for the closest-pair algorithm remains $O(n \lg n)$.

33.4-6

Suggest a change to the closest-pair algorithm that avoids presorting the Y array but leaves the running time as $O(n \lg n)$. (*Hint:* Merge sorted arrays Y_L and Y_R to form the sorted array Y.)

Problems

33-1 Convex layers

Given a set Q of points in the plane, we define the ***convex layers*** of Q inductively. The first convex layer of Q consists of those points in Q that are vertices of $\mathrm{CH}(Q)$. For $i > 1$, define Q_i to consist of the points of Q with all points in convex layers $1, 2, \ldots, i-1$ removed. Then, the ith convex layer of Q is $\mathrm{CH}(Q_i)$ if $Q_i \neq \emptyset$ and is undefined otherwise.

a. Give an $O(n^2)$-time algorithm to find the convex layers of a set of n points.

b. Prove that $\Omega(n \lg n)$ time is required to compute the convex layers of a set of n points with any model of computation that requires $\Omega(n \lg n)$ time to sort n real numbers.

33-2 Maximal layers

Let Q be a set of n points in the plane. We say that point (x, y) ***dominates*** point (x', y') if $x \geq x'$ and $y \geq y'$. A point in Q that is dominated by no other points in Q is said to be ***maximal***. Note that Q may contain many maximal points, which can be organized into ***maximal layers*** as follows. The first maximal layer L_1 is the set of maximal points of Q. For $i > 1$, the ith maximal layer L_i is the set of maximal points in $Q - \bigcup_{j=1}^{i-1} L_j$.

Suppose that Q has k nonempty maximal layers, and let y_i be the y-coordinate of the leftmost point in L_i for $i = 1, 2, \ldots, k$. For now, assume that no two points in Q have the same x- or y-coordinate.

a. Show that $y_1 > y_2 > \cdots > y_k$.

Consider a point (x, y) that is to the left of any point in Q and for which y is distinct from the y-coordinate of any point in Q. Let $Q' = Q \cup \{(x, y)\}$.

b. Let j be the minimum index such that $y_j < y$, unless $y < y_k$, in which case we let $j = k + 1$. Show that the maximal layers of Q' are as follows:

- If $j \leq k$, then the maximal layers of Q' are the same as the maximal layers of Q, except that L_j also includes (x, y) as its new leftmost point.
- If $j = k + 1$, then the first k maximal layers of Q' are the same as for Q, but in addition, Q' has a nonempty $(k + 1)$st maximal layer: $L_{k+1} = \{(x, y)\}$.

c. Describe an $O(n \lg n)$-time algorithm to compute the maximal layers of a set Q of n points. (*Hint:* Move a sweep line from right to left.)

d. Do any difficulties arise if we now allow input points to have the same x- or y-coordinate? Suggest a way to resolve such problems.

33-3 Ghostbusters and ghosts

A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The Ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost pairs, and then simultaneously each Ghostbuster will shoot a stream at his chosen ghost. As we all know, it is *very* dangerous to let streams cross, and so the Ghostbusters must choose pairings for which no streams will cross.

Assume that the position of each Ghostbuster and each ghost is a fixed point in the plane and that no three positions are colinear.

a. Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \lg n)$ time.

b. Give an $O(n^2 \lg n)$-time algorithm to pair Ghostbusters with ghosts in such a way that no streams cross.

33-4 Picking up sticks

Professor Charon has a set of n sticks, which are piled up in some configuration. Each stick is specified by its endpoints, and each endpoint is an ordered triple giving its (x, y, z) coordinates. No stick is vertical. He wishes to pick up all the sticks, one at a time, subject to the condition that he may pick up a stick only if there is no other stick on top of it.

a. Give a procedure that takes two sticks a and b and reports whether a is above, below, or unrelated to b.

b. Describe an efficient algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a legal order in which to pick them up.

33-5 Sparse-hulled distributions

Consider the problem of computing the convex hull of a set of points in the plane that have been drawn according to some known random distribution. Sometimes, the number of points, or size, of the convex hull of n points drawn from such a distribution has expectation $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$. We call such a distribution ***sparse-hulled***. Sparse-hulled distributions include the following:

- Points drawn uniformly from a unit-radius disk. The convex hull has expected size $\Theta(n^{1/3})$.
- Points drawn uniformly from the interior of a convex polygon with k sides, for any constant k. The convex hull has expected size $\Theta(\lg n)$.
- Points drawn according to a two-dimensional normal distribution. The convex hull has expected size $\Theta(\sqrt{\lg n})$.

a. Given two convex polygons with n_1 and n_2 vertices respectively, show how to compute the convex hull of all $n_1 + n_2$ points in $O(n_1 + n_2)$ time. (The polygons may overlap.)

b. Show how to compute the convex hull of a set of n points drawn independently according to a sparse-hulled distribution in $O(n)$ average-case time. (*Hint:* Recursively find the convex hulls of the first $n/2$ points and the second $n/2$ points, and then combine the results.)

Chapter notes

This chapter barely scratches the surface of computational-geometry algorithms and techniques. Books on computational geometry include those by Preparata and Shamos [282], Edelsbrunner [99], and O'Rourke [269].

Although geometry has been studied since antiquity, the development of algorithms for geometric problems is relatively new. Preparata and Shamos note that the earliest notion of the complexity of a problem was given by E. Lemoine in 1902. He was studying euclidean constructions—those using a compass and a ruler—and devised a set of five primitives: placing one leg of the compass on a given point, placing one leg of the compass on a given line, drawing a circle, passing the ruler's edge through a given point, and drawing a line. Lemoine was interested in the number of primitives needed to effect a given construction; he called this amount the "simplicity" of the construction.

The algorithm of Section 33.2, which determines whether any segments intersect, is due to Shamos and Hoey [313].

The original version of Graham's scan is given by Graham [150]. The package-wrapping algorithm is due to Jarvis [189]. Using a decision-tree model of computation, Yao [359] proved a worst-case lower bound of $\Omega(n \lg n)$ for the running time of any convex-hull algorithm. When the number of vertices h of the convex hull is taken into account, the prune-and-search algorithm of Kirkpatrick and Seidel [206], which takes $O(n \lg h)$ time, is asymptotically optimal.

The $O(n \lg n)$-time divide-and-conquer algorithm for finding the closest pair of points is by Shamos and appears in Preparata and Shamos [282]. Preparata and Shamos also show that the algorithm is asymptotically optimal in a decision-tree model.

34 NP-Completeness

Almost all the algorithms we have studied thus far have been ***polynomial-time algorithms***: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k. You might wonder whether *all* problems can be solved in polynomial time. The answer is no. For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time we allow. There are also problems that can be solved, but not in time $O(n^k)$ for any constant k. Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

The subject of this chapter, however, is an interesting class of problems, called the "NP-complete" problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called P $\neq$ NP question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

Several NP-complete problems are particularly tantalizing because they seem on the surface to be similar to problems that we know how to solve in polynomial time. In each of the following pairs of problems, one is solvable in polynomial time and the other is NP-complete, but the difference between problems appears to be slight:

Shortest vs. longest simple paths: In Chapter 24, we saw that even with negative edge weights, we can find *shortest* paths from a single source in a directed graph $G = (V, E)$ in $O(VE)$ time. Finding a *longest* simple path between two vertices is difficult, however. Merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.

Euler tour vs. hamiltonian cycle: An ***Euler tour*** of a connected, directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it is allowed to visit each vertex more than once. By Problem 22-3, we can determine whether a graph has an Euler tour in only $O(E)$ time and, in fact,

we can find the edges of the Euler tour in $O(E)$ time. A ***hamiltonian cycle*** of a directed graph $G = (V, E)$ is a simple cycle that contains each *vertex* in V. Determining whether a directed graph has a hamiltonian cycle is NP-complete. (Later in this chapter, we shall prove that determining whether an *undirected* graph has a hamiltonian cycle is NP-complete.)

2-CNF satisfiability vs. 3-CNF satisfiability: A boolean formula contains variables whose values are 0 or 1; boolean connectives such as $\wedge$ (AND), $\vee$ (OR), and $\neg$ (NOT); and parentheses. A boolean formula is ***satisfiable*** if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1. We shall define terms more formally later in this chapter, but informally, a boolean formula is in ***k-conjunctive normal form***, or k-CNF, if it is the AND of clauses of ORs of exactly k variables or their negations. For example, the boolean formula $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is in 2-CNF. (It has the satisfying assignment $x_1 = 1, x_2 = 0, x_3 = 1$.) Although we can determine in polynomial time whether a 2-CNF formula is satisfiable, we shall see later in this chapter that determining whether a 3-CNF formula is satisfiable is NP-complete.

NP-completeness and the classes P and NP

Throughout this chapter, we shall refer to three classes of problems: P, NP, and NPC, the latter class being the NP-complete problems. We describe them informally here, and we shall define them more formally later on.

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous chapters are in P.

The class NP consists of those problems that are "verifiable" in polynomial time. What do we mean by a problem being verifiable? If we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. For example, in the hamiltonian-cycle problem, given a directed graph $G = (V, E)$, a certificate would be a sequence $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$ of $|V|$ vertices. We could easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, \ldots, |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well. As another example, for 3-CNF satisfiability, a certificate would be an assignment of values to variables. We could check in polynomial time that this assignment satisfies the boolean formula.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate. We shall formalize this notion later in this chapter, but for now we can believe that P $\subseteq$ NP. The open question is whether or not P is a proper subset of NP.

Informally, a problem is in the class NPC—and we refer to it as being ***NP-complete***—if it is in NP and is as "hard" as any problem in NP. We shall formally define what it means to be as hard as any problem in NP later in this chapter. In the meantime, we will state without proof that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time algorithm. Most theoretical computer scientists believe that the NP-complete problems are intractable, since given the wide range of NP-complete problems that have been studied to date—without anyone having discovered a polynomial-time solution to any of them—it would be truly astounding if all of them could be solved in polynomial time. Yet, given the effort devoted thus far to proving that NP-complete problems are intractable—without a conclusive outcome—we cannot rule out the possibility that the NP-complete problems are in fact solvable in polynomial time.

To become a good algorithm designer, you must understand the rudiments of the theory of NP-completeness. If you can establish a problem as NP-complete, you provide good evidence for its intractability. As an engineer, you would then do better to spend your time developing an approximation algorithm (see Chapter 35) or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly. Moreover, many natural and interesting problems that on the surface seem no harder than sorting, graph searching, or network flow are in fact NP-complete. Therefore, you should become familiar with this remarkable class of problems.

Overview of showing problems to be NP-complete

The techniques we use to show that a particular problem is NP-complete differ fundamentally from the techniques used throughout most of this book to design and analyze algorithms. When we demonstrate that a problem is NP-complete, we are making a statement about how hard it is (or at least how hard we think it is), rather than about how easy it is. We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist. In this way, NP-completeness proofs bear some similarity to the proof in Section 8.1 of an $\Omega(n \lg n)$-time lower bound for any comparison sort algorithm; the specific techniques used for showing NP-completeness differ from the decision-tree method used in Section 8.1, however.

We rely on three key concepts in showing a problem to be NP-complete:

Decision problems vs. optimization problems

Many problems of interest are ***optimization problems***, in which each feasible (i.e., "legal") solution has an associated value, and we wish to find a feasible solution with the best value. For example, in a problem that we call SHORTEST-PATH,

we are given an undirected graph G and vertices u and v, and we wish to find a path from u to v that uses the fewest edges. In other words, SHORTEST-PATH is the single-pair shortest-path problem in an unweighted, undirected graph. NP-completeness applies directly not to optimization problems, however, but to ***decision problems***, in which the answer is simply "yes" or "no" (or, more formally, "1" or "0").

Although NP-complete problems are confined to the realm of decision problems, we can take advantage of a convenient relationship between optimization problems and decision problems. We usually can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized. For example, a decision problem related to SHORTEST-PATH is PATH: given a directed graph G, vertices u and v, and an integer k, does a path exist from u to v consisting of at most k edges?

The relationship between an optimization problem and its related decision problem works in our favor when we try to show that the optimization problem is "hard." That is because the decision problem is in a sense "easier," or at least "no harder." As a specific example, we can solve PATH by solving SHORTEST-PATH and then comparing the number of edges in the shortest path found to the value of the decision-problem parameter k. In other words, if an optimization problem is easy, its related decision problem is easy as well. Stated in a way that has more relevance to NP-completeness, if we can provide evidence that a decision problem is hard, we also provide evidence that its related optimization problem is hard. Thus, even though it restricts attention to decision problems, the theory of NP-completeness often has implications for optimization problems as well.

Reductions

The above notion of showing that one problem is no harder or no easier than another applies even when both problems are decision problems. We take advantage of this idea in almost every NP-completeness proof, as follows. Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an ***instance*** of that problem; for example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.
- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes."

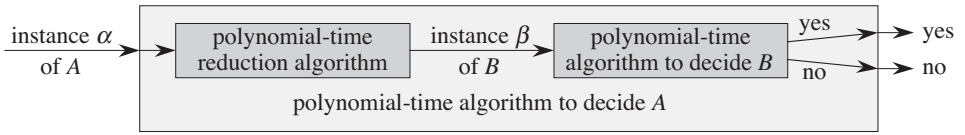


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, we transform an instance α of A into an instance β of B, we solve B in polynomial time, and we use the answer for β as the answer for α.

We call such a procedure a polynomial-time ***reduction algorithm*** and, as Figure 34.1 shows, it provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B, we use the "easiness" of B to prove the "easiness" of A.

Recalling that NP-completeness is about showing how hard a problem is rather than how easy it is, we use polynomial-time reductions in the opposite way to show that a problem is NP-complete. Let us take the idea a step further, and show how we could use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B. Suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist. (Let us not concern ourselves for now with how to find such a problem A.) Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B. Now we can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B. Suppose otherwise; i.e., suppose that B has a polynomial-time algorithm. Then, using the method shown in Figure 34.1, we would have a way to solve problem A in polynomial time, which contradicts our assumption that there is no polynomial-time algorithm for A.

For NP-completeness, we cannot assume that there is absolutely no polynomial-time algorithm for problem A. The proof methodology is similar, however, in that we prove that problem B is NP-complete on the assumption that problem A is also NP-complete.

A first NP-complete problem

Because the technique of reduction relies on having a problem already known to be NP-complete in order to prove a different problem NP-complete, we need a "first" NP-complete problem. The problem we shall use is the circuit-satisfiability problem, in which we are given a boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of boolean inputs to this circuit that causes its output to be 1. We shall prove that this first problem is NP-complete in Section 34.3.

Chapter outline

This chapter studies the aspects of NP-completeness that bear most directly on the analysis of algorithms. In Section 34.1, we formalize our notion of "problem" and define the complexity class P of polynomial-time solvable decision problems. We also see how these notions fit into the framework of formal-language theory. Section 34.2 defines the class NP of decision problems whose solutions are verifiable in polynomial time. It also formally poses the $P \neq NP$ question.

Section 34.3 shows we can relate problems via polynomial-time "reductions." It defines NP-completeness and sketches a proof that one problem, called "circuit satisfiability," is NP-complete. Having found one NP-complete problem, we show in Section 34.4 how to prove other problems to be NP-complete much more simply by the methodology of reductions. We illustrate this methodology by showing that two formula-satisfiability problems are NP-complete. With additional reductions, we show in Section 34.5 a variety of other problems to be NP-complete.

34.1 Polynomial time

We begin our study of NP-completeness by formalizing our notion of polynomial-time solvable problems. We generally regard these problems as tractable, but for philosophical, not mathematical, reasons. We can offer three supporting arguments.

First, although we may reasonably regard a problem that requires time $\Theta(n^{100})$ to be intractable, very few practical problems require time on the order of such a high-degree polynomial. The polynomial-time computable problems encountered in practice typically require much less time. Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow. Even if the current best algorithm for a problem has a running time of $\Theta(n^{100})$, an algorithm with a much better running time will likely soon be discovered.

Second, for many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another. For example, the class of problems solvable in polynomial time by the serial random-access machine used throughout most of this book is the same as the class of problems solvable in polynomial time on abstract Turing machines.[1] It is also the same as the class of problems solvable in polynomial time on a parallel computer when the number of processors grows polynomially with the input size.

Third, the class of polynomial-time solvable problems has nice closure properties, since polynomials are closed under addition, multiplication, and composition. For example, if the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial. Exercise 34.1-5 asks you to show that if an algorithm makes a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then the running time of the composite algorithm is polynomial.

Abstract problems

To understand the class of polynomial-time solvable problems, we must first have a formal notion of what a "problem" is. We define an ***abstract problem*** Q to be a binary relation on a set I of problem ***instances*** and a set S of problem ***solutions***. For example, an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices. A solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists. The problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices. Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

This formulation of an abstract problem is more general than we need for our purposes. As we saw above, the theory of NP-completeness restricts attention to ***decision problems***: those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$. For example, a decision problem related to SHORTEST-PATH is the problem PATH that we saw earlier. If $i = \langle G, u, v, k \rangle$ is an instance of the decision problem PATH, then $\text{PATH}(i) = 1$ (yes) if a shortest path from u to v has at most k edges, and $\text{PATH}(i) = 0$ (no) otherwise. Many abstract problems are not decision problems, but rather ***optimization problems***, which require some value to be minimized or maximized. As we saw above, however, we can usually recast an optimization problem as a decision problem that is no harder.

---

[1] See Hopcroft and Ullman [180] or Lewis and Papadimitriou [236] for a thorough treatment of the Turing-machine model.

Encodings

In order for a computer program to solve an abstract problem, we must represent problem instances in a way that the program understands. An ***encoding*** of a set S of abstract objects is a mapping e from S to the set of binary strings.[2] For example, we are all familiar with encoding the natural numbers $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$ as the strings $\{0, 1, 10, 11, 100, \ldots\}$. Using this encoding, $e(17) = 10001$. If you have looked at computer representations of keyboard characters, you probably have seen the ASCII code, where, for example, the encoding of `A` is 1000001. We can encode a compound object as a binary string by combining the representations of its constituent parts. Polygons, graphs, functions, ordered pairs, programs—all can be encoded as binary strings.

Thus, a computer algorithm that "solves" some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a ***concrete problem***. We say that an algorithm ***solves*** a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$ time.[3] A concrete problem is ***polynomial-time solvable***, therefore, if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.

We can now formally define the ***complexity class*** **P** as the set of concrete decision problems that are polynomial-time solvable.

We can use encodings to map abstract problems to concrete problems. Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$, an encoding $e : I \to \{0, 1\}^*$ can induce a related concrete decision problem, which we denote by $e(Q)$.[4] If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$, then the solution to the concrete-problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$. As a technicality, some binary strings might represent no meaningful abstract-problem instance. For convenience, we shall assume that any such string maps arbitrarily to 0. Thus, the concrete problem produces the same solutions as the abstract problem on binary-string instances that represent the encodings of abstract-problem instances.

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge, but we would

---

[2] The codomain of e need not be *binary* strings; any set of strings over a finite alphabet having at least 2 symbols will do.

[3] We assume that the algorithm's output is separate from its input. Because it takes at least one time step to produce each bit of the output and the algorithm takes $O(T(n))$ time steps, the size of the output is $O(T(n))$.

[4] We denote by $\{0, 1\}^*$ the set of all strings composed of symbols from the set $\{0, 1\}$.

like the definition to be independent of any particular encoding. That is, the efficiency of solving a problem should not depend on how the problem is encoded. Unfortunately, it depends quite heavily on the encoding. For example, suppose that an integer k is to be provided as the sole input to an algorithm, and suppose that the running time of the algorithm is $\Theta(k)$. If the integer k is provided in ***unary***—a string of k 1s—then the running time of the algorithm is $O(n)$ on length-n inputs, which is polynomial time. If we use the more natural binary representation of the integer k, however, then the input length is $n = \lfloor \lg k \rfloor + 1$. In this case, the running time of the algorithm is $\Theta(k) = \Theta(2^n)$, which is exponential in the size of the input. Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

How we encode an abstract problem matters quite a bit to how we understand polynomial time. We cannot really talk about solving an abstract problem without first specifying an encoding. Nevertheless, in practice, if we rule out "expensive" encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time. For example, representing integers in base 3 instead of binary has no effect on whether a problem is solvable in polynomial time, since we can convert an integer represented in base 3 to an integer represented in base 2 in polynomial time.

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is ***polynomial-time computable*** if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$. For some set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomially related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.[5] That is, a polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa. If two encodings e_1 and e_2 of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use, as the following lemma shows.

Lemma 34.1

Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in \mathrm{P}$ if and only if $e_2(Q) \in \mathrm{P}$.

---

[5]Technically, we also require the functions f_{12} and f_{21} to "map noninstances to noninstances." A ***noninstance*** of an encoding e is a string $x \in \{0,1\}^*$ such that there is no instance i for which $e(i) = x$. We require that $f_{12}(x) = y$ for every noninstance x of encoding e_1, where y is some noninstance of e_2, and that $f_{21}(x') = y'$ for every noninstance x' of e_2, where y' is some noninstance of e_1.

Proof We need only prove the forward direction, since the backward direction is symmetric. Suppose, therefore, that $e_1(Q)$ can be solved in time $O(n^k)$ for some constant k. Further, suppose that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in time $O(n^c)$ for some constant c, where $n = |e_2(i)|$. To solve problem $e_2(Q)$, on input $e_2(i)$, we first compute $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$. How long does this take? Converting encodings takes time $O(n^c)$, and therefore $|e_1(i)| = O(n^c)$, since the output of a serial computer cannot be longer than its running time. Solving the problem on $e_1(i)$ takes time $O(|e_1(i)|^k) = O(n^{ck})$, which is polynomial since both c and k are constants. ■

Thus, whether an abstract problem has its instances encoded in binary or base 3 does not affect its "complexity," that is, whether it is polynomial-time solvable or not; but if instances are encoded in unary, its complexity may change. In order to be able to converse in an encoding-independent fashion, we shall generally assume that problem instances are encoded in any reasonable, concise fashion, unless we specifically say otherwise. To be precise, we shall assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas. (ASCII is one such encoding scheme.) With such a "standard" encoding in hand, we can derive reasonable encodings of other mathematical objects, such as tuples, graphs, and formulas. To denote the standard encoding of an object, we shall enclose the object in angle braces. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

As long as we implicitly use an encoding that is polynomially related to this standard encoding, we can talk directly about abstract problems without reference to any particular encoding, knowing that the choice of encoding has no effect on whether the abstract problem is polynomial-time solvable. Henceforth, we shall generally assume that all problem instances are binary strings encoded using the standard encoding, unless we explicitly specify the contrary. We shall also typically neglect the distinction between abstract and concrete problems. You should watch out for problems that arise in practice, however, in which a standard encoding is not obvious and the encoding does make a difference.

A formal-language framework

By focusing on decision problems, we can take advantage of the machinery of formal-language theory. Let's review some definitions from that theory. An ***alphabet*** Σ is a finite set of symbols. A ***language*** L over Σ is any set of strings made up of symbols from Σ. For example, if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$ is the language of binary represen-

tations of prime numbers. We denote the ***empty string*** by ε, the ***empty language*** by $\emptyset$, and the language of all strings over Σ by Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$ is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

We can perform a variety of operations on languages. Set-theoretic operations, such as ***union*** and ***intersection***, follow directly from the set-theoretic definitions. We define the ***complement*** of L by $\overline{L} = \Sigma^* - L$. The ***concatenation*** $L_1 L_2$ of two languages L_1 and L_2 is the language

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\} .$$

The ***closure*** or ***Kleene star*** of a language L is the language

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots ,$$

where L^k is the language obtained by concatenating L to itself k times.

From the point of view of language theory, the set of instances for any decision problem Q is simply the set Σ^*, where $\Sigma = \{0, 1\}$. Since Q is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view Q as a language L over $\Sigma = \{0, 1\}$, where

$$L = \{x \in \Sigma^* : Q(x) = 1\} .$$

For example, the decision problem PATH has the corresponding language

$$\begin{aligned} \text{PATH} = \{\langle G, u, v, k\rangle : \ & G = (V, E) \text{ is an undirected graph,} \\ & u, v \in V, \\ & k \geq 0 \text{ is an integer, and} \\ & \text{there exists a path from } u \text{ to } v \text{ in } G \\ & \text{consisting of at most } k \text{ edges}\} . \end{aligned}$$

(Where convenient, we shall sometimes use the same name—PATH in this case—to refer to both a decision problem and its corresponding language.)

The formal-language framework allows us to express concisely the relation between decision problems and algorithms that solve them. We say that an algorithm A ***accepts*** a string $x \in \{0, 1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1. The language ***accepted*** by an algorithm A is the set of strings $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts. An algorithm A ***rejects*** a string x if $A(x) = 0$.

Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm may loop forever. A language L is ***decided*** by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. A language L is ***accepted in polynomial time*** by an algorithm A if it is accepted by A and if in addition there exists a constant k such that for any length-n string $x \in L$,

algorithm A accepts x in time $O(n^k)$. A language L is ***decided in polynomial time*** by an algorithm A if there exists a constant k such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$. Thus, to accept a language, an algorithm need only produce an answer when provided a string in L, but to decide a language, it must correctly accept or reject every string in $\{0,1\}^*$.

As an example, the language PATH can be accepted in polynomial time. One polynomial-time accepting algorithm verifies that G encodes an undirected graph, verifies that u and v are vertices in G, uses breadth-first search to compute a shortest path from u to v in G, and then compares the number of edges on the shortest path obtained with k. If G encodes an undirected graph and the path found from u to v has at most k edges, the algorithm outputs 1 and halts. Otherwise, the algorithm runs forever. This algorithm does not decide PATH, however, since it does not explicitly output 0 for instances in which a shortest path has more than k edges. A decision algorithm for PATH must explicitly reject binary strings that do not belong to PATH. For a decision problem such as PATH, such a decision algorithm is easy to design: instead of running forever when there is not a path from u to v with at most k edges, it outputs 0 and halts. (It must also output 0 and halt if the input encoding is faulty.) For other problems, such as Turing's Halting Problem, there exists an accepting algorithm, but no decision algorithm exists.

We can informally define a ***complexity class*** as a set of languages, membership in which is determined by a ***complexity measure***, such as running time, of an algorithm that determines whether a given string x belongs to language L. The actual definition of a complexity class is somewhat more technical.[6]

Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

$$\mathrm{P} = \{L \subseteq \{0,1\}^* : \text{there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\} .$$

In fact, P is also the class of languages that can be accepted in polynomial time.

Theorem 34.2
$\mathrm{P} = \{L : L \text{ is accepted by a polynomial-time algorithm}\}$.

Proof Because the class of languages decided by polynomial-time algorithms is a subset of the class of languages accepted by polynomial-time algorithms, we need only show that if L is accepted by a polynomial-time algorithm, it is decided by a polynomial-time algorithm. Let L be the language accepted by some

[6]For more on complexity classes, see the seminal paper by Hartmanis and Stearns [162].

polynomial-time algorithm A. We shall use a classic "simulation" argument to construct another polynomial-time algorithm A' that decides L. Because A accepts L in time $O(n^k)$ for some constant k, there also exists a constant c such that A accepts L in at most cn^k steps. For any input string x, the algorithm A' simulates cn^k steps of A. After simulating cn^k steps, algorithm A' inspects the behavior of A. If A has accepted x, then A' accepts x by outputting a 1. If A has not accepted x, then A' rejects x by outputting a 0. The overhead of A' simulating A does not increase the running time by more than a polynomial factor, and thus A' is a polynomial-time algorithm that decides L. ■

Note that the proof of Theorem 34.2 is nonconstructive. For a given language $L \in \text{P}$, we may not actually know a bound on the running time for the algorithm A that accepts L. Nevertheless, we know that such a bound exists, and therefore, that an algorithm A' exists that can check the bound, even though we may not be able to find the algorithm A' easily.

Exercises

34.1-1

Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices. Define the decision problem LONGEST-PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ is an undirected graph, $u, v \in V$, $k \geq 0$ is an integer, and there exists a simple path from u to v in G consisting of at least k edges$\}$. Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH $\in$ P.

34.1-2

Give a formal definition for the problem of finding the longest simple cycle in an undirected graph. Give a related decision problem. Give the language corresponding to the decision problem.

34.1-3

Give a formal encoding of directed graphs as binary strings using an adjacency-matrix representation. Do the same using an adjacency-list representation. Argue that the two representations are polynomially related.

34.1-4

Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

34.1-5
Show that if an algorithm makes at most a constant number of calls to polynomial-time subroutines and performs an additional amount of work that also takes polynomial time, then it runs in polynomial time. Also show that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

34.1-6
Show that the class P, viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if $L_1, L_2 \in \mathrm{P}$, then $L_1 \cup L_2 \in \mathrm{P}$, $L_1 \cap L_2 \in \mathrm{P}$, $L_1 L_2 \in \mathrm{P}$, $\overline{L}_1 \in \mathrm{P}$, and $L_1^* \in \mathrm{P}$.

34.2 Polynomial-time verification

We now look at algorithms that verify membership in languages. For example, suppose that for a given instance $\langle G, u, v, k \rangle$ of the decision problem PATH, we are also given a path p from u to v. We can easily check whether p is a path in G and whether the length of p is at most k, and if so, we can view p as a "certificate" that the instance indeed belongs to PATH. For the decision problem PATH, this certificate doesn't seem to buy us much. After all, PATH belongs to P—in fact, we can solve PATH in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch. We shall now examine a problem for which we know of no polynomial-time decision algorithm and yet, given a certificate, verification is easy.

Hamiltonian cycles

The problem of finding a hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a ***hamiltonian cycle*** of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a hamiltonian cycle is said to be ***hamiltonian***; otherwise, it is ***nonhamiltonian***. The name honors W. R. Hamilton, who described a mathematical game on the dodecahedron (Figure 34.2(a)) in which one player sticks five pins in any five consecutive vertices and the other player must complete the path to form a cycle

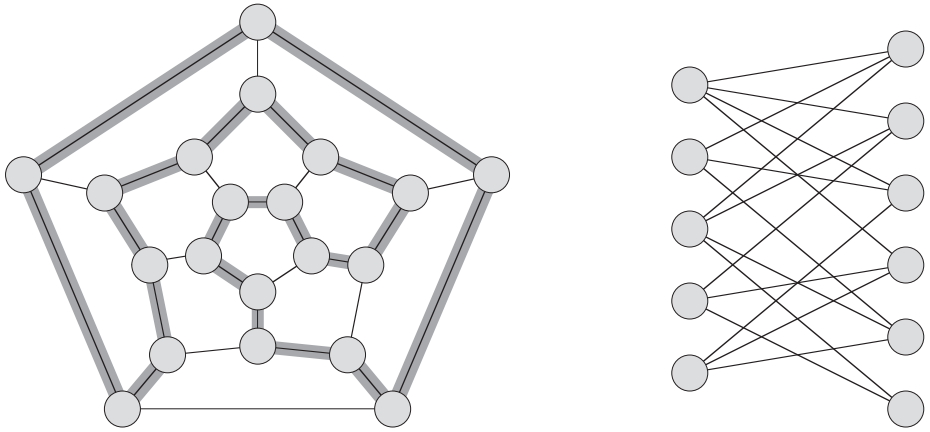


Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by shaded edges. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

containing all the vertices.[7] The dodecahedron is hamiltonian, and Figure 34.2(a) shows one hamiltonian cycle. Not all graphs are hamiltonian, however. For example, Figure 34.2(b) shows a bipartite graph with an odd number of vertices. Exercise 34.2-2 asks you to show that all such graphs are nonhamiltonian.

We can define the ***hamiltonian-cycle problem***, "Does a graph G have a hamiltonian cycle?" as a formal language:

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\} .$$

How might an algorithm decide the language HAM-CYCLE? Given a problem instance $\langle G \rangle$, one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a hamiltonian path. What is the running time of this algorithm? If we use the "reasonable" encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\sqrt{n})$, where $n = |\langle G \rangle|$ is the length of the encoding of G. There are $m!$ possible permutations

[7] In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [157, p. 624] wrote, "I have found that some young persons have been much amused by trying a new mathematical game which the Icosion furnishes, one person sticking five pins in any five consecutive points ... and the other player then aiming to insert, which by the theory in this letter can always be done, fifteen other pins, in cyclical succession, so as to cover all the other points, and to end in immediate proximity to the pin wherewith his antagonist had begun."

of the vertices, and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n}\,!) = \Omega(2^{\sqrt{n}})$, which is not $O(n^k)$ for any constant k. Thus, this naive algorithm does not run in polynomial time. In fact, the hamiltonian-cycle problem is NP-complete, as we shall prove in Section 34.5.

Verification algorithms

Consider a slightly easier problem. Suppose that a friend tells you that a given graph G is hamiltonian, and then offers to prove it by giving you the vertices in order along the hamiltonian cycle. It would certainly be easy enough to verify the proof: simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V and whether each of the consecutive edges along the cycle actually exists in the graph. You could certainly implement this verification algorithm to run in $O(n^2)$ time, where n is the length of the encoding of G. Thus, a proof that a hamiltonian cycle exists in a graph can be verified in polynomial time.

We define a ***verification algorithm*** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that $A(x, y) = 1$. The ***language verified*** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\} .$$

Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreover, for any string $x \notin L$, there must be no certificate proving that $x \in L$. For example, in the hamiltonian-cycle problem, the certificate is the list of vertices in some hamiltonian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact. Conversely, if a graph is not hamiltonian, there can be no list of vertices that fools the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the proposed "cycle" to be sure.

The complexity class NP

The ***complexity class*** **NP** is the class of languages that can be verified by a polynomial-time algorithm.[8] More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* : \text{ there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\} .$$

We say that algorithm A ***verifies*** language L ***in polynomial time***.

From our earlier discussion on the hamiltonian-cycle problem, we now see that HAM-CYCLE $\in$ NP. (It is always nice to know that an important set is nonempty.) Moreover, if $L \in$ P, then $L \in$ NP, since if there is a polynomial-time algorithm to decide L, the algorithm can be easily converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to be in L. Thus, P $\subseteq$ NP.

It is unknown whether P = NP, but most researchers believe that P and NP are not the same class. Intuitively, the class P consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly. You may have learned from experience that it is often more difficult to solve a problem from scratch than to verify a clearly presented solution, especially when working under time constraints. Theoretical computer scientists generally believe that this analogy extends to the classes P and NP, and thus that NP includes languages that are not in P.

There is more compelling, though not conclusive, evidence that P $\neq$ NP—the existence of languages that are "NP-complete." We shall study this class in Section 34.3.

Many other fundamental questions beyond the P $\neq$ NP question remain unresolved. Figure 34.3 shows some possible scenarios. Despite much work by many researchers, no one even knows whether the class NP is closed under complement. That is, does $L \in$ NP imply $\overline{L} \in$ NP? We can define the ***complexity class*** **co-NP** as the set of languages L such that $\overline{L} \in$ NP. We can restate the question of whether NP is closed under complement as whether NP = co-NP. Since P is closed under complement (Exercise 34.1-6), it follows from Exercise 34.2-9 that P $\subseteq$ NP $\cap$ co-NP. Once again, however, no one knows whether P = NP $\cap$ co-NP or whether there is some language in NP $\cap$ co-NP $-$ P.

---

[8] The name "NP" stands for "nondeterministic polynomial time." The class NP was originally studied in the context of nondeterminism, but this book uses the somewhat simpler yet equivalent notion of verification. Hopcroft and Ullman [180] give a good presentation of NP-completeness in terms of nondeterministic models of computation.

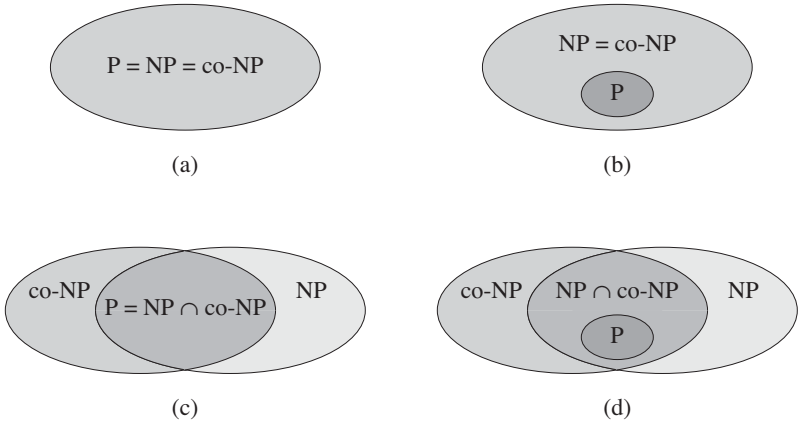


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** P = NP = co-NP. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. **(c)** P = NP∩co-NP, but NP is not closed under complement. **(d)** NP ≠ co-NP and P ≠ NP ∩ co-NP. Most researchers regard this possibility as the most likely.

Thus, our understanding of the precise relationship between P and NP is woefully incomplete. Nevertheless, even though we might not be able to prove that a particular problem is intractable, if we can prove that it is NP-complete, then we have gained valuable information about it.

Exercises

34.2-1

Consider the language GRAPH-ISOMORPHISM $= \{\langle G_1, G_2\rangle : G_1 \text{ and } G_2 \text{ are isomorphic graphs}\}$. Prove that GRAPH-ISOMORPHISM $\in$ NP by describing a polynomial-time algorithm to verify the language.

34.2-2

Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian.

34.2-3

Show that if HAM-CYCLE $\in$ P, then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

34.2-4
Prove that the class NP of languages is closed under union, intersection, concatenation, and Kleene star. Discuss the closure of NP under complement.

34.2-5
Show that any language in NP can be decided by an algorithm running in time $2^{O(n^k)}$ for some constant k.

34.2-6
A ***hamiltonian path*** in a graph is a simple path that visits every vertex exactly once. Show that the language HAM-PATH $= \{\langle G, u, v\rangle$: there is a hamiltonian path from u to v in graph $G\}$ belongs to NP.

34.2-7
Show that the hamiltonian-path problem from Exercise 34.2-6 can be solved in polynomial time on directed acyclic graphs. Give an efficient algorithm for the problem.

34.2-8
Let ϕ be a boolean formula constructed from the boolean input variables $x_1, x_2, \ldots, x_k$, negations ($\neg$), ANDs ($\wedge$), ORs ($\vee$), and parentheses. The formula ϕ is a ***tautology*** if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY $\in$ co-NP.

34.2-9
Prove that P $\subseteq$ co-NP.

34.2-10
Prove that if NP $\neq$ co-NP, then P $\neq$ NP.

34.2-11
Let G be a connected, undirected graph with at least 3 vertices, and let G^3 be the graph obtained by connecting all pairs of vertices that are connected by a path in G of length at most 3. Prove that G^3 is hamiltonian. (*Hint:* Construct a spanning tree for G, and use an inductive argument.)

34.3 NP-completeness and reducibility

Perhaps the most compelling reason why theoretical computer scientists believe that $\mathrm{P} \neq \mathrm{NP}$ comes from the existence of the class of "NP-complete" problems. This class has the intriguing property that if *any* NP-complete problem can be solved in polynomial time, then *every* problem in NP has a polynomial-time solution, that is, $\mathrm{P} = \mathrm{NP}$. Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem.

The language HAM-CYCLE is one NP-complete problem. If we could decide HAM-CYCLE in polynomial time, then we could solve every problem in NP in polynomial time. In fact, if $\mathrm{NP} - \mathrm{P}$ should turn out to be nonempty, we could say with certainty that HAM-CYCLE $\in \mathrm{NP} - \mathrm{P}$.

The NP-complete languages are, in a sense, the "hardest" languages in NP. In this section, we shall show how to compare the relative "hardness" of languages using a precise notion called "polynomial-time reducibility." Then we formally define the NP-complete languages, and we finish by sketching a proof that one such language, called CIRCUIT-SAT, is NP-complete. In Sections 34.4 and 34.5, we shall use the notion of reducibility to show that many other problems are NP-complete.

Reducibility

Intuitively, a problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q. For example, the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations. Given an instance $ax + b = 0$, we transform it to $0x^2 + ax + b = 0$, whose solution provides a solution to $ax + b = 0$. Thus, if a problem Q reduces to another problem Q', then Q is, in a sense, "no harder to solve" than Q'.

Returning to our formal-language framework for decision problems, we say that a language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_{\mathrm{P}} L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \text{ if and only if } f(x) \in L_2 \,. \tag{34.1}$$

We call the function f the ***reduction function***, and a polynomial-time algorithm F that computes f is a ***reduction algorithm***.

Figure 34.4 illustrates the idea of a polynomial-time reduction from a language L_1 to another language L_2. Each language is a subset of $\{0,1\}^*$. The reduction function f provides a polynomial-time mapping such that if $x \in L_1$,

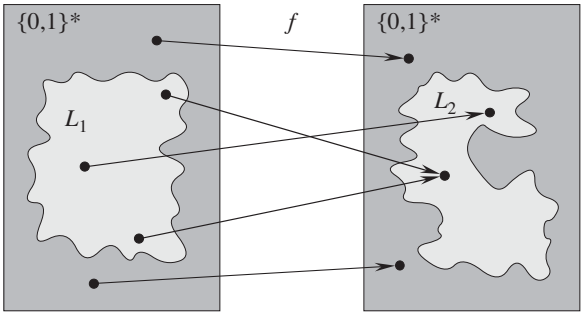


Figure 34.4 An illustration of a polynomial-time reduction from a language L_1 to a language L_2 via a reduction function f. For any input $x \in \{0,1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$.

then $f(x) \in L_2$. Moreover, if $x \notin L_1$, then $f(x) \notin L_2$. Thus, the reduction function maps any instance x of the decision problem represented by the language L_1 to an instance $f(x)$ of the problem represented by L_2. Providing an answer to whether $f(x) \in L_2$ directly provides the answer to whether $x \in L_1$.

Polynomial-time reductions give us a powerful tool for proving that various languages belong to P.

Lemma 34.3
If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in$ P implies $L_1 \in$ P.

Proof Let A_2 be a polynomial-time algorithm that decides L_2, and let F be a polynomial-time reduction algorithm that computes the reduction function f. We shall construct a polynomial-time algorithm A_1 that decides L_1.

Figure 34.5 illustrates how we construct A_1. For a given input $x \in \{0,1\}^*$, algorithm A_1 uses F to transform x into $f(x)$, and then it uses A_2 to test whether $f(x) \in L_2$. Algorithm A_1 takes the output from algorithm A_2 and produces that answer as its own output.

The correctness of A_1 follows from condition (34.1). The algorithm runs in polynomial time, since both F and A_2 run in polynomial time (see Exercise 34.1-5). ■

NP-completeness

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \leq_P L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is

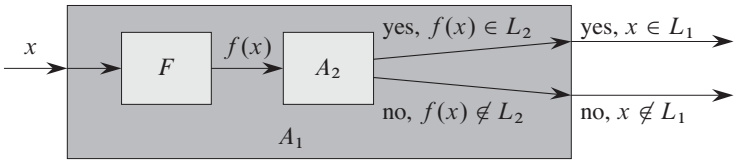


Figure 34.5 The proof of Lemma 34.3. The algorithm F is a reduction algorithm that computes the reduction function f from L_1 to L_2 in polynomial time, and A_2 is a polynomial-time algorithm that decides L_2. Algorithm A_1 decides whether $x \in L_1$ by using F to transform any input x into $f(x)$ and then using A_2 to decide whether $f(x) \in L_2$.

why the "less than or equal to" notation for reduction is mnemonic. We can now define the set of NP-complete languages, which are the hardest problems in NP.

A language $L \subseteq \{0,1\}^*$ is ***NP-complete*** if

1. $L \in$ NP, and
2. $L' \leq_P L$ for every $L' \in$ NP.

If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***. We also define NPC to be the class of NP-complete languages.

As the following theorem shows, NP-completeness is at the crux of deciding whether P is in fact equal to NP.

Theorem 34.4
If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Proof Suppose that $L \in$ P and also that $L \in$ NPC. For any $L' \in$ NP, we have $L' \leq_P L$ by property 2 of the definition of NP-completeness. Thus, by Lemma 34.3, we also have that $L' \in$ P, which proves the first statement of the theorem.

To prove the second statement, note that it is the contrapositive of the first statement. ■

It is for this reason that research into the P $\neq$ NP question centers around the NP-complete problems. Most theoretical computer scientists believe that P $\neq$ NP, which leads to the relationships among P, NP, and NPC shown in Figure 34.6. But, for all we know, someone may yet come up with a polynomial-time algorithm for an NP-complete problem, thus proving that P = NP. Nevertheless, since no polynomial-time algorithm for any NP-complete problem has yet been discov-

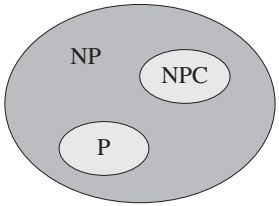


Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $\text{P} \cap \text{NPC} = \emptyset$.

ered, a proof that a problem is NP-complete provides excellent evidence that it is intractable.

Circuit satisfiability

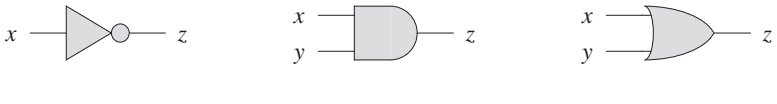
We have defined the notion of an NP-complete problem, but up to this point, we have not actually proved that any problem is NP-complete. Once we prove that at least one problem is NP-complete, we can use polynomial-time reducibility as a tool to prove other problems to be NP-complete. Thus, we now focus on demonstrating the existence of an NP-complete problem: the circuit-satisfiability problem.

Unfortunately, the formal proof that the circuit-satisfiability problem is NP-complete requires technical detail beyond the scope of this text. Instead, we shall informally describe a proof that relies on a basic understanding of boolean combinational circuits.

Boolean combinational circuits are built from boolean combinational elements that are interconnected by wires. A ***boolean combinational element*** is any circuit element that has a constant number of boolean inputs and outputs and that performs a well-defined function. Boolean values are drawn from the set $\{0, 1\}$, where 0 represents FALSE and 1 represents TRUE.

The boolean combinational elements that we use in the circuit-satisfiability problem compute simple boolean functions, and they are known as ***logic gates***. Figure 34.7 shows the three basic logic gates that we use in the circuit-satisfiability problem: the ***NOT gate*** (or ***inverter***), the ***AND gate***, and the ***OR gate***. The NOT gate takes a single binary ***input*** x, whose value is either 0 or 1, and produces a binary ***output*** z whose value is opposite that of the input value. Each of the other two gates takes two binary inputs x and y and produces a single binary output z.

We can describe the operation of each gate, and of any boolean combinational element, by a ***truth table***, shown under each gate in Figure 34.7. A truth table gives the outputs of the combinational element for each possible setting of the inputs. For



x	$\neg x$
0	1
1	0

(a)

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

(b)

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

(c)

Figure 34.7 Three basic logic gates, with binary inputs and outputs. Under each gate is the truth table that describes the gate's operation. **(a)** The NOT gate. **(b)** The AND gate. **(c)** The OR gate.

example, the truth table for the OR gate tells us that when the inputs are $x = 0$ and $y = 1$, the output value is $z = 1$. We use the symbols $\neg$ to denote the NOT function, $\wedge$ to denote the AND function, and $\vee$ to denote the OR function. Thus, for example, $0 \vee 1 = 1$.

We can generalize AND and OR gates to take more than two inputs. An AND gate's output is 1 if all of its inputs are 1, and its output is 0 otherwise. An OR gate's output is 1 if any of its inputs are 1, and its output is 0 otherwise.

A ***boolean combinational circuit*** consists of one or more boolean combinational elements interconnected by ***wires***. A wire can connect the output of one element to the input of another, thereby providing the output value of the first element as an input value of the second. Figure 34.8 shows two similar boolean combinational circuits, differing in only one gate. Part (a) of the figure also shows the values on the individual wires, given the input $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$. Although a single wire may have no more than one combinational-element output connected to it, it can feed several element inputs. The number of element inputs fed by a wire is called the ***fan-out*** of the wire. If no element output is connected to a wire, the wire is a ***circuit input***, accepting input values from an external source. If no element input is connected to a wire, the wire is a ***circuit output***, providing the results of the circuit's computation to the outside world. (An internal wire can also fan out to a circuit output.) For the purpose of defining the circuit-satisfiability problem, we limit the number of circuit outputs to 1, though in actual hardware design, a boolean combinational circuit may have multiple outputs.

Boolean combinational circuits contain no cycles. In other words, suppose we create a directed graph $G = (V, E)$ with one vertex for each combinational element and with k directed edges for each wire whose fan-out is k; the graph contains a directed edge (u, v) if a wire connects the output of element u to an input of element v. Then G must be acyclic.

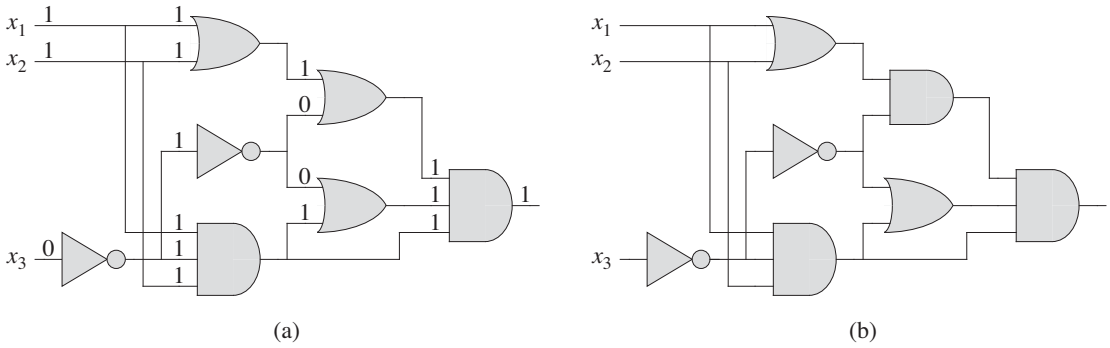


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

A ***truth assignment*** for a boolean combinational circuit is a set of boolean input values. We say that a one-output boolean combinational circuit is ***satisfiable*** if it has a ***satisfying assignment***: a truth assignment that causes the output of the circuit to be 1. For example, the circuit in Figure 34.8(a) has the satisfying assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$, and so it is satisfiable. As Exercise 34.3-1 asks you to show, no assignment of values to x_1, x_2, and x_3 causes the circuit in Figure 34.8(b) to produce a 1 output; it always produces 0, and so it is unsatisfiable.

The ***circuit-satisfiability problem*** is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?" In order to pose this question formally, however, we must agree on a standard encoding for circuits. The ***size*** of a boolean combinational circuit is the number of boolean combinational elements plus the number of wires in the circuit. We could devise a graphlike encoding that maps any given circuit C into a binary string $\langle C\rangle$ whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

$$\text{CIRCUIT-SAT} = \{\langle C\rangle : C \text{ is a satisfiable boolean combinational circuit}\} .$$

The circuit-satisfiability problem arises in the area of computer-aided hardware optimization. If a subcircuit always produces 0, that subcircuit is unnecessary; the designer can replace it by a simpler subcircuit that omits all logic gates and provides the constant 0 value as its output. You can see why we would like to have a polynomial-time algorithm for this problem.

Given a circuit C, we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if the circuit has k inputs, then we would have to check up to 2^k possible assignments. When

the size of C is polynomial in k, checking each one takes $\Omega(2^k)$ time, which is superpolynomial in the size of the circuit.[9] In fact, as we have claimed, there is strong evidence that no polynomial-time algorithm exists that solves the circuit-satisfiability problem because circuit satisfiability is NP-complete. We break the proof of this fact into two parts, based on the two parts of the definition of NP-completeness.

Lemma 34.5
The circuit-satisfiability problem belongs to the class NP.

Proof We shall provide a two-input, polynomial-time algorithm A that can verify CIRCUIT-SAT. One of the inputs to A is (a standard encoding of) a boolean combinational circuit C. The other input is a certificate corresponding to an assignment of boolean values to the wires in C. (See Exercise 34.3-4 for a smaller certificate.)

We construct the algorithm A as follows. For each logic gate in the circuit, it checks that the value provided by the certificate on the output wire is correctly computed as a function of the values on the input wires. Then, if the output of the entire circuit is 1, the algorithm outputs 1, since the values assigned to the inputs of C provide a satisfying assignment. Otherwise, A outputs 0.

Whenever a satisfiable circuit C is input to algorithm A, there exists a certificate whose length is polynomial in the size of C and that causes A to output a 1. Whenever an unsatisfiable circuit is input, no certificate can fool A into believing that the circuit is satisfiable. Algorithm A runs in polynomial time: with a good implementation, linear time suffices. Thus, we can verify CIRCUIT-SAT in polynomial time, and CIRCUIT-SAT $\in$ NP. ■

The second part of proving that CIRCUIT-SAT is NP-complete is to show that the language is NP-hard. That is, we must show that every language in NP is polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full of technical intricacies, and so we shall settle for a sketch of the proof based on some understanding of the workings of computer hardware.

A computer program is stored in the computer memory as a sequence of instructions. A typical instruction encodes an operation to be performed, addresses of operands in memory, and an address where the result is to be stored. A special memory location, called the ***program counter***, keeps track of which instruc-

[9] On the other hand, if the size of the circuit C is $\Theta(2^k)$, then an algorithm whose running time is $O(2^k)$ has a running time that is polynomial in the circuit size. Even if P $\neq$ NP, this situation would not contradict the NP-completeness of the problem; the existence of a polynomial-time algorithm for a special case does not imply that there is a polynomial-time algorithm for all cases.

tion is to be executed next. The program counter automatically increments upon fetching each instruction, thereby causing the computer to execute instructions sequentially. The execution of an instruction can cause a value to be written to the program counter, however, which alters the normal sequential execution and allows the computer to loop and perform conditional branches.

At any point during the execution of a program, the computer's memory holds the entire state of the computation. (We take the memory to include the program itself, the program counter, working storage, and any of the various bits of state that a computer maintains for bookkeeping.) We call any particular state of computer memory a ***configuration***. We can view the execution of an instruction as mapping one configuration to another. The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit, which we denote by M in the proof of the following lemma.

Lemma 34.6
The circuit-satisfiability problem is NP-hard.

Proof Let L be any language in NP. We shall describe a polynomial-time algorithm F computing a reduction function f that maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$ CIRCUIT-SAT.

Since $L \in$ NP, there must exist an algorithm A that verifies L in polynomial time. The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Let $T(n)$ denote the worst-case running time of algorithm A on length-n input strings, and let $k \geq 1$ be a constant such that $T(n) = O(n^k)$ and the length of the certificate is $O(n^k)$. (The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate, but since the length of the certificate is polynomial in the length n of the input string, the running time is polynomial in n.)

The basic idea of the proof is to represent the computation of A as a sequence of configurations. As Figure 34.9 illustrates, we can break each configuration into parts consisting of the program for A, the program counter and auxiliary machine state, the input x, the certificate y, and working storage. The combinational circuit M, which implements the computer hardware, maps each configuration c_i to the next configuration c_{i+1}, starting from the initial configuration c_0. Algorithm A writes its output—0 or 1—to some designated location by the time it finishes executing, and if we assume that thereafter A halts, the value never changes. Thus, if the algorithm runs for at most $T(n)$ steps, the output appears as one of the bits in $c_{T(n)}$.

The reduction algorithm F constructs a single combinational circuit that computes all configurations produced by a given initial configuration. The idea is to

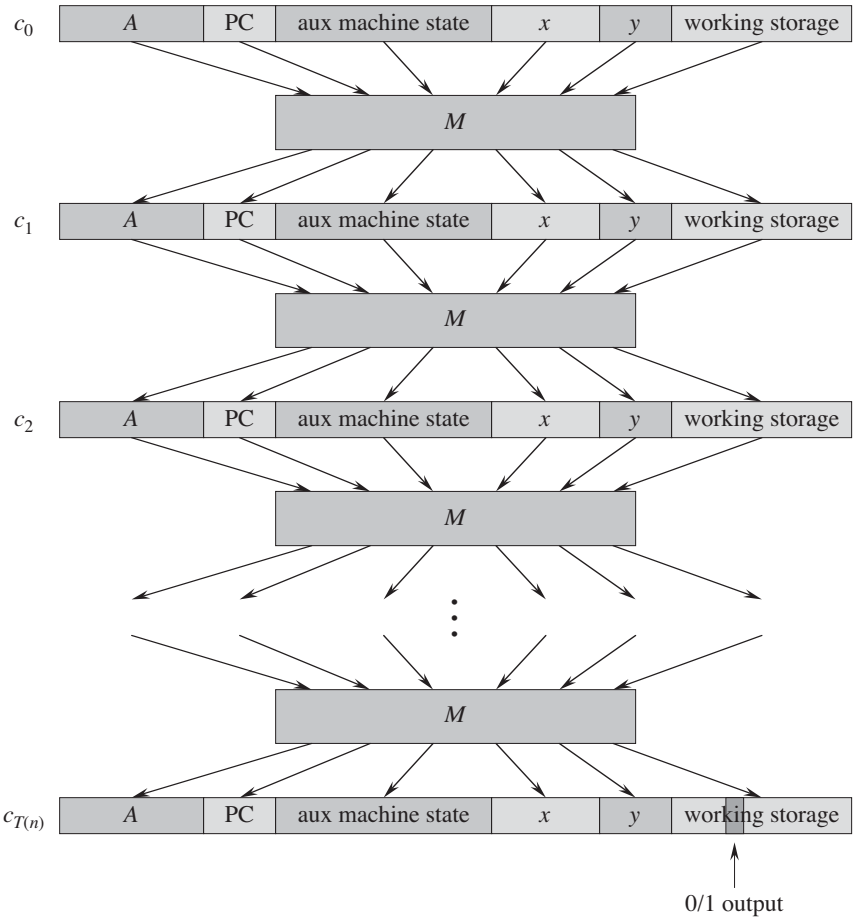


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. Except for the certificate y, the initial configuration c_0 is constant. A boolean combinational circuit M maps each configuration to the next configuration. The output is a distinguished bit in the working storage.

paste together $T(n)$ copies of the circuit M. The output of the ith circuit, which produces configuration c_i, feeds directly into the input of the $(i+1)$st circuit. Thus, the configurations, rather than being stored in the computer's memory, simply reside as values on the wires connecting copies of M.

Recall what the polynomial-time reduction algorithm F must do. Given an input x, it must compute a circuit $C = f(x)$ that is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. When F obtains an input x, it first computes $n = |x|$ and constructs a combinational circuit C' consisting of $T(n)$ copies of M. The input to C' is an initial configuration corresponding to a computation on $A(x, y)$, and the output is the configuration $c_{T(n)}$.

Algorithm F modifies circuit C' slightly to construct the circuit $C = f(x)$. First, it wires the inputs to C' corresponding to the program for A, the initial program counter, the input x, and the initial state of memory directly to these known values. Thus, the only remaining inputs to the circuit correspond to the certificate y. Second, it ignores all outputs from C', except for the one bit of $c_{T(n)}$ corresponding to the output of A. This circuit C, so constructed, computes $C(y) = A(x, y)$ for any input y of length $O(n^k)$. The reduction algorithm F, when provided an input string x, computes such a circuit C and outputs it.

We need to prove two properties. First, we must show that F correctly computes a reduction function f. That is, we must show that C is satisfiable if and only if there exists a certificate y such that $A(x, y) = 1$. Second, we must show that F runs in polynomial time.

To show that F correctly computes a reduction function, let us suppose that there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$. Then, if we apply the bits of y to the inputs of C, the output of C is $C(y) = A(x, y) = 1$. Thus, if a certificate exists, then C is satisfiable. For the other direction, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y) = 1$, from which we conclude that $A(x, y) = 1$. Thus, F correctly computes a reduction function.

To complete the proof sketch, we need only show that F runs in time polynomial in $n = |x|$. The first observation we make is that the number of bits required to represent a configuration is polynomial in n. The program for A itself has constant size, independent of the length of its input x. The length of the input x is n, and the length of the certificate y is $O(n^k)$. Since the algorithm runs for at most $O(n^k)$ steps, the amount of working storage required by A is polynomial in n as well. (We assume that this memory is contiguous; Exercise 34.3-5 asks you to extend the argument to the situation in which the locations accessed by A are scattered across a much larger region of memory and the particular pattern of scattering can differ for each input x.)

The combinational circuit M implementing the computer hardware has size polynomial in the length of a configuration, which is $O(n^k)$; hence, the size of M is polynomial in n. (Most of this circuitry implements the logic of the memory

system.) The circuit C consists of at most $t = O(n^k)$ copies of M, and hence it has size polynomial in n. The reduction algorithm F can construct C from x in polynomial time, since each step of the construction takes polynomial time. ■

The language CIRCUIT-SAT is therefore at least as hard as any language in NP, and since it belongs to NP, it is NP-complete.

Theorem 34.7
The circuit-satisfiability problem is NP-complete.

Proof Immediate from Lemmas 34.5 and 34.6 and from the definition of NP-completeness. ■

Exercises

34.3-1
Verify that the circuit in Figure 34.8(b) is unsatisfiable.

34.3-2
Show that the $\leq_P$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

34.3-3
Prove that $L \leq_P \overline{L}$ if and only if $\overline{L} \leq_P L$.

34.3-4
Show that we could have used a satisfying assignment as a certificate in an alternative proof of Lemma 34.5. Which certificate makes for an easier proof?

34.3-5
The proof of Lemma 34.6 assumes that the working storage for algorithm A occupies a contiguous region of polynomial size. Where in the proof do we exploit this assumption? Argue that this assumption does not involve any loss of generality.

34.3-6
A language L is ***complete*** for a language class C with respect to polynomial-time reductions if $L \in C$ and $L' \leq_P L$ for all $L' \in C$. Show that $\emptyset$ and $\{0,1\}^*$ are the only languages in P that are not complete for P with respect to polynomial-time reductions.

34.3-7
Show that, with respect to polynomial-time reductions (see Exercise 34.3-6), L is complete for NP if and only if $\overline{L}$ is complete for co-NP.

34.3-8
The reduction algorithm F in the proof of Lemma 34.6 constructs the circuit $C = f(x)$ based on knowledge of x, A, and k. Professor Sartre observes that the string x is input to F, but only the existence of A, k, and the constant factor implicit in the $O(n^k)$ running time is known to F (since the language L belongs to NP), not their actual values. Thus, the professor concludes that F can't possibly construct the circuit C and that the language CIRCUIT-SAT is not necessarily NP-hard. Explain the flaw in the professor's reasoning.

34.4 NP-completeness proofs

We proved that the circuit-satisfiability problem is NP-complete by a direct proof that $L \leq_P$ CIRCUIT-SAT for every language $L \in$ NP. In this section, we shall show how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. We shall illustrate this methodology by proving that various formula-satisfiability problems are NP-complete. Section 34.5 provides many more examples of the methodology.

The following lemma is the basis of our method for showing that a language is NP-complete.

Lemma 34.8
If L is a language such that $L' \leq_P L$ for some $L' \in$ NPC, then L is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

Proof Since L' is NP-complete, for all $L'' \in$ NP, we have $L'' \leq_P L'$. By supposition, $L' \leq_P L$, and thus by transitivity (Exercise 34.3-2), we have $L'' \leq_P L$, which shows that L is NP-hard. If $L \in$ NP, we also have $L \in$ NPC. ■

In other words, by reducing a known NP-complete language L' to L, we implicitly reduce every language in NP to L. Thus, Lemma 34.8 gives us a method for proving that a language L is NP-complete:

1. Prove $L \in$ NP.
2. Select a known NP-complete language L'.

3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.

(Steps 2–5 show that L is NP-hard.) This methodology of reducing from a single known NP-complete language is far simpler than the more complicated process of showing directly how to reduce from every language in NP. Proving CIRCUIT-SAT $\in$ NPC has given us a "foot in the door." Because we know that the circuit-satisfiability problem is NP-complete, we now can prove much more easily that other problems are NP-complete. Moreover, as we develop a catalog of known NP-complete problems, we will have more and more choices for languages from which to reduce.

Formula satisfiability

We illustrate the reduction methodology by giving an NP-completeness proof for the problem of determining whether a boolean formula, not a circuit, is satisfiable. This problem has the historical honor of being the first problem ever shown to be NP-complete.

We formulate the ***(formula) satisfiability*** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables: $x_1, x_2, \ldots, x_n$;
2. m boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if); and
3. parentheses. (Without loss of generality, we assume that there are no redundant parentheses, i.e., a formula contains at most one pair of parentheses per boolean connective.)

We can easily encode a boolean formula ϕ in a length that is polynomial in $n + m$. As in boolean combinational circuits, a ***truth assignment*** for a boolean formula ϕ is a set of values for the variables of ϕ, and a ***satisfying assignment*** is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a ***satisfiable*** formula. The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms,

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable boolean formula}\} .$$

As an example, the formula

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

has the satisfying assignment $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$, since

$$\begin{aligned} \phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg(1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1 \,, \end{aligned} \tag{34.2}$$

and thus this formula ϕ belongs to SAT.

The naive algorithm to determine whether an arbitrary boolean formula is satisfiable does not run in polynomial time. A formula with n variables has 2^n possible assignments. If the length of $\langle\phi\rangle$ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ time, which is superpolynomial in the length of $\langle\phi\rangle$. As the following theorem shows, a polynomial-time algorithm is unlikely to exist.

Theorem 34.9
Satisfiability of boolean formulas is NP-complete.

Proof We start by arguing that SAT $\in$ NP. Then we prove that SAT is NP-hard by showing that CIRCUIT-SAT $\leq_P$ SAT; by Lemma 34.8, this will prove the theorem.

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression, much as we did in equation (34.2) above. This task is easy to do in polynomial time. If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable. Thus, the first condition of Lemma 34.8 for NP-completeness holds.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\leq_P$ SAT. In other words, we need to show how to reduce any instance of circuit satisfiability to an instance of formula satisfiability in polynomial time. We can use induction to express any boolean combinational circuit as a boolean formula. We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas. We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Unfortunately, this straightforward method does not amount to a polynomial-time reduction. As Exercise 34.4-1 asks you to show, shared subformulas—which arise from gates whose output wires have fan-out of 2 or more—can cause the size of the generated formula to grow exponentially. Thus, the reduction algorithm must be somewhat more clever.

Figure 34.10 illustrates how we overcome this problem, using as an example the circuit from Figure 34.8(a). For each wire x_i in the circuit C, the formula ϕ

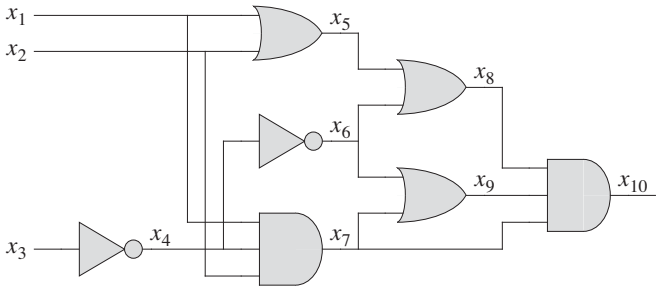


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

has a variable x_i. We can now express how each gate operates as a small formula involving the variables of its incident wires. For example, the operation of the output AND gate is $x_{10} \leftrightarrow (x_7 \land x_8 \land x_9)$. We call each of these small formulas a ***clause***.

The formula ϕ produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate. For the circuit in the figure, the formula is

$$\begin{aligned}\phi \;=\; & x_{10} \land (x_4 \leftrightarrow \neg x_3) \\ & \land (x_5 \leftrightarrow (x_1 \lor x_2)) \\ & \land (x_6 \leftrightarrow \neg x_4) \\ & \land (x_7 \leftrightarrow (x_1 \land x_2 \land x_4)) \\ & \land (x_8 \leftrightarrow (x_5 \lor x_6)) \\ & \land (x_9 \leftrightarrow (x_6 \lor x_7)) \\ & \land (x_{10} \leftrightarrow (x_7 \land x_8 \land x_9)) \;.\end{aligned}$$

Given a circuit C, it is straightforward to produce such a formula ϕ in polynomial time.

Why is the circuit C satisfiable exactly when the formula ϕ is satisfiable? If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1. Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument. Thus, we have shown that CIRCUIT-SAT $\leq_{\mathrm{P}}$ SAT, which completes the proof. ■

3-CNF satisfiability

We can prove many problems NP-complete by reducing from formula satisfiability. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases that we must consider. We often prefer to reduce from a restricted language of boolean formulas, so that we need to consider fewer cases. Of course, we must not restrict the language so much that it becomes polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

We define 3-CNF satisfiability using the following terms. A ***literal*** in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in ***conjunctive normal form***, or ***CNF***, if it is expressed as an AND of ***clauses***, each of which is the OR of one or more literals. A boolean formula is in ***3-conjunctive normal form***, or ***3-CNF***, if each clause has exactly three distinct literals.

For example, the boolean formula

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in 3-CNF. The first of its three clauses is $(x_1 \vee \neg x_1 \vee \neg x_2)$, which contains the three literals x_1, $\neg x_1$, and $\neg x_2$.

In 3-CNF-SAT, we are asked whether a given boolean formula ϕ in 3-CNF is satisfiable. The following theorem shows that a polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form.

Theorem 34.10
Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Proof The argument we used in the proof of Theorem 34.9 to show that SAT $\in$ NP applies equally well here to show that 3-CNF-SAT $\in$ NP. By Lemma 34.8, therefore, we need only show that SAT $\leq_P$ 3-CNF-SAT.

We break the reduction algorithm into three basic steps. Each step progressively transforms the input formula ϕ closer to the desired 3-conjunctive normal form.

The first step is similar to the one used to prove CIRCUIT-SAT $\leq_P$ SAT in Theorem 34.9. First, we construct a binary "parse" tree for the input formula ϕ, with literals as leaves and connectives as internal nodes. Figure 34.11 shows such a parse tree for the formula

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \,. \tag{34.3}$$

Should the input formula contain a clause such as the OR of several literals, we use associativity to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. We can now think of the binary parse tree as a circuit for computing the function.

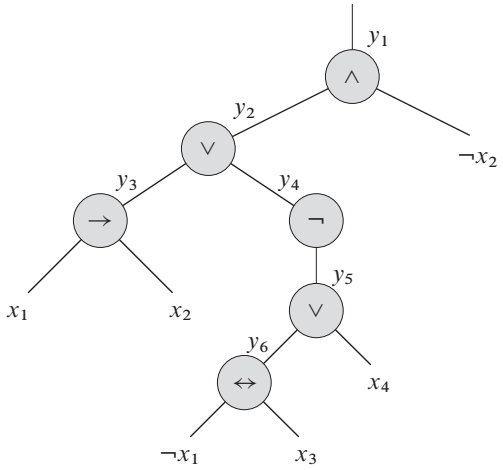


Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Mimicking the reduction in the proof of Theorem 34.9, we introduce a variable y_i for the output of each internal node. Then, we rewrite the original formula ϕ as the AND of the root variable and a conjunction of clauses describing the operation of each node. For the formula (34.3), the resulting expression is

$$\begin{aligned}\phi' \;=\; & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge\; (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge\; (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge\; (y_4 \leftrightarrow \neg y_5) \\ & \wedge\; (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge\; (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\ .\end{aligned}$$

Observe that the formula ϕ' thus obtained is a conjunction of clauses ϕ'_i, each of which has at most 3 literals. The only requirement that we might fail to meet is that each clause has to be an OR of 3 literals.

The second step of the reduction converts each clause ϕ'_i into conjunctive normal form. We construct a truth table for ϕ'_i by evaluating all possible assignments to its variables. Each row of the truth table consists of a possible assignment of the variables of the clause, together with the value of the clause under that assignment. Using the truth-table entries that evaluate to 0, we build a formula in ***disjunctive normal form*** (or ***DNF***)—an OR of ANDs—that is equivalent to $\neg\phi'_i$. We then negate this formula and convert it into a CNF formula ϕ''_i by using ***DeMorgan's***

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figure 34.12 The truth table for the clause $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

laws for propositional logic,

$$\begin{aligned}\neg(a \wedge b) &= \neg a \vee \neg b\ ,\\ \neg(a \vee b) &= \neg a \wedge \neg b\ ,\end{aligned}$$

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, we convert the clause $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ into CNF as follows. The truth table for ϕ'_1 appears in Figure 34.12. The DNF formula equivalent to $\neg\phi'_1$ is

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)\ .$$

Negating and applying DeMorgan's laws, we get the CNF formula

$$\begin{aligned}\phi''_1 = {} & (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2)\\ & \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)\ ,\end{aligned}$$

which is equivalent to the original clause ϕ'_1.

At this point, we have converted each clause ϕ'_i of the formula ϕ' into a CNF formula ϕ''_i, and thus ϕ' is equivalent to the CNF formula ϕ'' consisting of the conjunction of the ϕ''_i. Moreover, each clause of ϕ'' has at most 3 literals.

The third and final step of the reduction further transforms the formula so that each clause has *exactly* 3 distinct literals. We construct the final 3-CNF formula ϕ''' from the clauses of the CNF formula ϕ''. The formula ϕ''' also uses two auxiliary variables that we shall call p and q. For each clause C_i of ϕ'', we include the following clauses in ϕ''':

- If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ'''.
- If C_i has 2 distinct literals, that is, if $C_i = (l_1 \vee l_2)$, where l_1 and l_2 are literals, then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''. The literals p and $\neg p$ merely fulfill the syntactic requirement that each clause of ϕ''' has

exactly 3 distinct literals. Whether $p = 0$ or $p = 1$, one of the clauses is equivalent to $l_1 \vee l_2$, and the other evaluates to 1, which is the identity for AND.

- If C_i has just 1 distinct literal l, then include $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$ as clauses of ϕ'''. Regardless of the values of p and q, one of the four clauses is equivalent to l, and the other 3 evaluate to 1.

We can see that the 3-CNF formula ϕ''' is satisfiable if and only if ϕ is satisfiable by inspecting each of the three steps. Like the reduction from CIRCUIT-SAT to SAT, the construction of ϕ' from ϕ in the first step preserves satisfiability. The second step produces a CNF formula ϕ'' that is algebraically equivalent to ϕ'. The third step produces a 3-CNF formula ϕ''' that is effectively equivalent to ϕ'', since any assignment to the variables p and q produces a formula that is algebraically equivalent to ϕ''.

We must also show that the reduction can be computed in polynomial time. Constructing ϕ' from ϕ introduces at most 1 variable and 1 clause per connective in ϕ. Constructing ϕ'' from ϕ' can introduce at most 8 clauses into ϕ'' for each clause from ϕ', since each clause of ϕ' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows. The construction of ϕ''' from ϕ'' introduces at most 4 clauses into ϕ''' for each clause of ϕ''. Thus, the size of the resulting formula ϕ''' is polynomial in the length of the original formula. Each of the constructions can easily be accomplished in polynomial time. ■

Exercises

34.4-1

Consider the straightforward (nonpolynomial-time) reduction in the proof of Theorem 34.9. Describe a circuit of size n that, when converted to a formula by this method, yields a formula whose size is exponential in n.

34.4-2

Show the 3-CNF formula that results when we use the method of Theorem 34.10 on the formula (34.3).

34.4-3

Professor Jagger proposes to show that SAT $\leq_P$ 3-CNF-SAT by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula ϕ, form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to $\neg\phi$, and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to ϕ. Show that this strategy does not yield a polynomial-time reduction.

34.4-4
Show that the problem of determining whether a boolean formula is a tautology is complete for co-NP. (*Hint:* See Exercise 34.3-7.)

34.4-5
Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

34.4-6
Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

34.4-7
Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly 2 literals per clause. Show that 2-CNF-SAT $\in$ P. Make your algorithm as efficient as possible. (*Hint:* Observe that $x \vee y$ is equivalent to $\neg x \rightarrow y$. Reduce 2-CNF-SAT to an efficiently solvable problem on a directed graph.)

34.5 NP-complete problems

NP-complete problems arise in diverse domains: boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, biology, chemistry, physics, and more. In this section, we shall use the reduction methodology to provide NP-completeness proofs for a variety of problems drawn from graph theory and set partitioning.

Figure 34.13 outlines the structure of the NP-completeness proofs in this section and Section 34.4. We prove each language in the figure to be NP-complete by reduction from the language that points to it. At the root is CIRCUIT-SAT, which we proved NP-complete in Theorem 34.7.

34.5.1 The clique problem

A ***clique*** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The ***size*** of a clique is the number of vertices it contains. The ***clique problem*** is the optimization problem of finding a clique of maximum size in

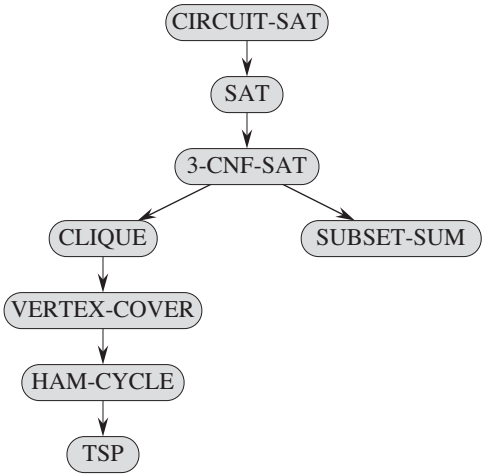


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph. The formal definition is

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ is a graph containing a clique of size } k\} \ .$$

A naive algorithm for determining whether a graph $G = (V, E)$ with $|V|$ vertices has a clique of size k is to list all k-subsets of V, and check each one to see whether it forms a clique. The running time of this algorithm is $\Omega(k^2\binom{|V|}{k})$, which is polynomial if k is a constant. In general, however, k could be near $|V|/2$, in which case the algorithm runs in superpolynomial time. Indeed, an efficient algorithm for the clique problem is unlikely to exist.

Theorem 34.11
The clique problem is NP-complete.

Proof To show that CLIQUE $\in$ NP, for a given graph $G = (V, E)$, we use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. We can check whether V' is a clique in polynomial time by checking whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq_{\text{P}}$ CLIQUE, which shows that the clique problem is NP-hard. You might be surprised that we should be able to prove such a result, since on the surface logical formulas seem to have little to do with graphs.

The reduction algorithm begins with an instance of 3-CNF-SAT. Let $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be a boolean formula in 3-CNF with k clauses. For $r =$

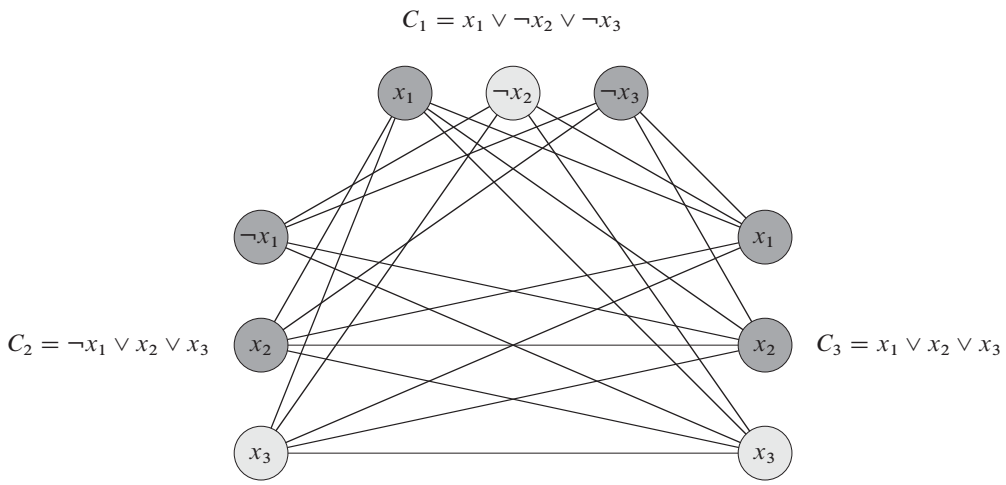


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

$1, 2, \ldots, k$, each clause C_r has exactly three distinct literals l_1^r, l_2^r, and l_3^r. We shall construct a graph G such that ϕ is satisfiable if and only if G has a clique of size k.

We construct the graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, we place a triple of vertices v_1^r, v_2^r, and v_3^r into V. We put an edge between two vertices v_i^r and v_j^s if both of the following hold:

- v_i^r and v_j^s are in different triples, that is, $r \neq s$, and
- their corresponding literals are ***consistent***, that is, l_i^r is not the negation of l_j^s.

We can easily build this graph from ϕ in polynomial time. As an example of this construction, if we have

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) ,$$

then G is the graph shown in Figure 34.14.

We must show that this transformation of ϕ into G is a reduction. First, suppose that ϕ has a satisfying assignment. Then each clause C_r contains at least one literal l_i^r that is assigned 1, and each such literal corresponds to a vertex v_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. We claim that V' is a clique. For any two vertices $v_i^r, v_j^s \in V'$, where $r \neq s$, both corresponding literals l_i^r and l_j^s map to 1 by the given satisfying assignment, and thus the literals

cannot be complements. Thus, by the construction of G, the edge (v_i^r, v_j^s) belongs to E.

Conversely, suppose that G has a clique V' of size k. No edges in G connect vertices in the same triple, and so V' contains exactly one vertex per triple. We can assign 1 to each literal l_i^r such that $v_i^r \in V'$ without fear of assigning 1 to both a literal and its complement, since G contains no edges between inconsistent literals. Each clause is satisfied, and so ϕ is satisfied. (Any variables that do not correspond to a vertex in the clique may be set arbitrarily.) ■

In the example of Figure 34.14, a satisfying assignment of ϕ has $x_2 = 0$ and $x_3 = 1$. A corresponding clique of size $k = 3$ consists of the vertices corresponding to $\neg x_2$ from the first clause, x_3 from the second clause, and x_3 from the third clause. Because the clique contains no vertices corresponding to either x_1 or $\neg x_1$, we can set x_1 to either 0 or 1 in this satisfying assignment.

Observe that in the proof of Theorem 34.11, we reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure. You might think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple. Indeed, we have shown that CLIQUE is NP-hard only in this restricted case, but this proof suffices to show that CLIQUE is NP-hard in general graphs. Why? If we had a polynomial-time algorithm that solved CLIQUE on general graphs, it would also solve CLIQUE on restricted graphs.

The opposite approach—reducing instances of 3-CNF-SAT with a special structure to general instances of CLIQUE—would not have sufficed, however. Why not? Perhaps the instances of 3-CNF-SAT that we chose to reduce from were "easy," and so we would not have reduced an NP-hard problem to CLIQUE.

Observe also that the reduction used the instance of 3-CNF-SAT, but not the solution. We would have erred if the polynomial-time reduction had relied on knowing whether the formula ϕ is satisfiable, since we do not know how to decide whether ϕ is satisfiable in polynomial time.

34.5.2 The vertex-cover problem

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The ***size*** of a vertex cover is the number of vertices in it. For example, the graph in Figure 34.15(b) has a vertex cover $\{w, z\}$ of size 2.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to

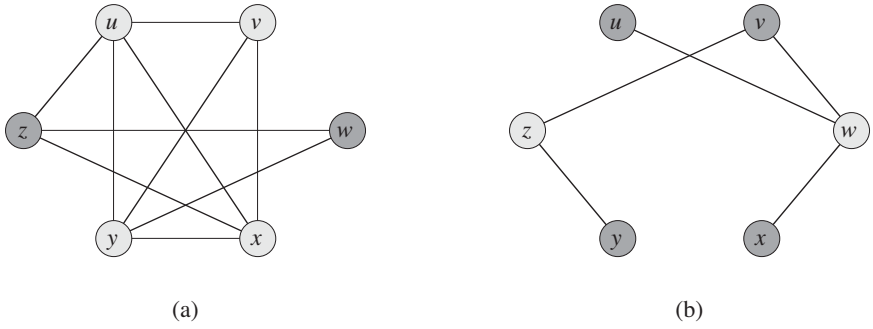


Figure 34.15 Reducing CLIQUE to VERTEX-COVER. **(a)** An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$. **(b)** The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$.

determine whether a graph has a vertex cover of a given size k. As a language, we define

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has a vertex cover of size } k\} .$$

The following theorem shows that this problem is NP-complete.

Theorem 34.12
The vertex-cover problem is NP-complete.

Proof We first show that VERTEX-COVER $\in$ NP. Suppose we are given a graph $G = (V, E)$ and an integer k. The certificate we choose is the vertex cover $V' \subseteq V$ itself. The verification algorithm affirms that $|V'| = k$, and then it checks, for each edge $(u, v) \in E$, that $u \in V'$ or $v \in V'$. We can easily verify the certificate in polynomial time.

We prove that the vertex-cover problem is NP-hard by showing that CLIQUE $\leq_P$ VERTEX-COVER. This reduction relies on the notion of the "complement" of a graph. Given an undirected graph $G = (V, E)$, we define the ***complement*** of G as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. In other words, $\overline{G}$ is the graph containing exactly those edges that are not in G. Figure 34.15 shows a graph and its complement and illustrates the reduction from CLIQUE to VERTEX-COVER.

The reduction algorithm takes as input an instance $\langle G, k\rangle$ of the clique problem. It computes the complement $\overline{G}$, which we can easily do in polynomial time. The output of the reduction algorithm is the instance $\langle \overline{G}, |V| - k\rangle$ of the vertex-cover problem. To complete the proof, we show that this transformation is indeed a

reduction: the graph G has a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. We claim that $V - V'$ is a vertex cover in $\overline{G}$. Let (u, v) be any edge in $\overline{E}$. Then, $(u, v) \notin E$, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from $\overline{E}$, every edge of $\overline{E}$ is covered by a vertex in $V - V'$. Hence, the set $V - V'$, which has size $|V| - k$, forms a vertex cover for $\overline{G}$.

Conversely, suppose that $\overline{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then, for all $u, v \in V$, if $(u, v) \in \overline{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words, $V - V'$ is a clique, and it has size $|V| - |V'| = k$. ■

Since VERTEX-COVER is NP-complete, we don't expect to find a polynomial-time algorithm for finding a minimum-size vertex cover. Section 35.1 presents a polynomial-time "approximation algorithm," however, which produces "approximate" solutions for the vertex-cover problem. The size of a vertex cover produced by the algorithm is at most twice the minimum size of a vertex cover.

Thus, we shouldn't give up hope just because a problem is NP-complete. We may be able to design a polynomial-time approximation algorithm that obtains near-optimal solutions, even though finding an optimal solution is NP-complete. Chapter 35 gives several approximation algorithms for NP-complete problems.

34.5.3 The hamiltonian-cycle problem

We now return to the hamiltonian-cycle problem defined in Section 34.2.

Theorem 34.13
The hamiltonian cycle problem is NP-complete.

Proof We first show that HAM-CYCLE belongs to NP. Given a graph $G = (V, E)$, our certificate is the sequence of $|V|$ vertices that makes up the hamiltonian cycle. The verification algorithm checks that this sequence contains each vertex in V exactly once and that with the first vertex repeated at the end, it forms a cycle in G. That is, it checks that there is an edge between each pair of consecutive vertices and between the first and last vertices. We can verify the certificate in polynomial time.

We now prove that VERTEX-COVER $\leq_P$ HAM-CYCLE, which shows that HAM-CYCLE is NP-complete. Given an undirected graph $G = (V, E)$ and an

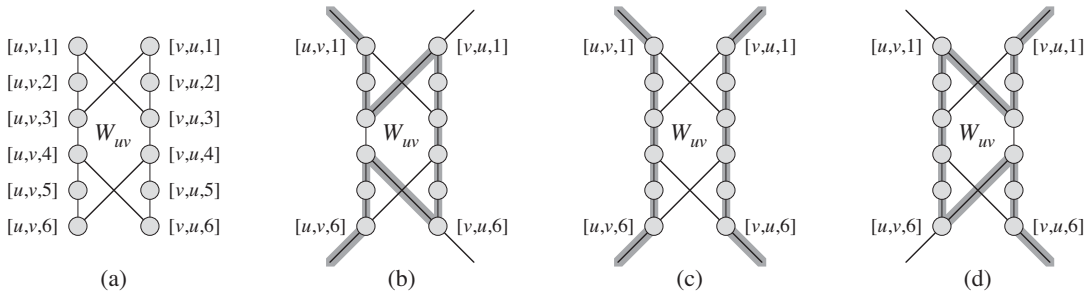


Figure 34.16 The widget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge (u, v) of graph G corresponds to widget W_{uv} in the graph G' created in the reduction. **(a)** The widget, with individual vertices labeled. **(b)–(d)** The shaded paths are the only possible ones through the widget that include all vertices, assuming that the only connections from the widget to the remainder of G' are through vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$.

integer k, we construct an undirected graph $G' = (V', E')$ that has a hamiltonian cycle if and only if G has a vertex cover of size k.

Our construction uses a ***widget***, which is a piece of a graph that enforces certain properties. Figure 34.16(a) shows the widget we use. For each edge $(u, v) \in E$, the graph G' that we construct will contain one copy of this widget, which we denote by W_{uv}. We denote each vertex in W_{uv} by $[u, v, i]$ or $[v, u, i]$, where $1 \le i \le 6$, so that each widget W_{uv} contains 12 vertices. Widget W_{uv} also contains the 14 edges shown in Figure 34.16(a).

Along with the internal structure of the widget, we enforce the properties we want by limiting the connections between the widget and the remainder of the graph G' that we construct. In particular, only vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$ will have edges incident from outside W_{uv}. Any hamiltonian cycle of G' must traverse the edges of W_{uv} in one of the three ways shown in Figures 34.16(b)–(d). If the cycle enters through vertex $[u, v, 1]$, it must exit through vertex $[u, v, 6]$, and it either visits all 12 of the widget's vertices (Figure 34.16(b)) or the six vertices $[u, v, 1]$ through $[u, v, 6]$ (Figure 34.16(c)). In the latter case, the cycle will have to reenter the widget to visit vertices $[v, u, 1]$ through $[v, u, 6]$. Similarly, if the cycle enters through vertex $[v, u, 1]$, it must exit through vertex $[v, u, 6]$, and it either visits all 12 of the widget's vertices (Figure 34.16(d)) or the six vertices $[v, u, 1]$ through $[v, u, 6]$ (Figure 34.16(c)). No other paths through the widget that visit all 12 vertices are possible. In particular, it is impossible to construct two vertex-disjoint paths, one of which connects $[u, v, 1]$ to $[v, u, 6]$ and the other of which connects $[v, u, 1]$ to $[u, v, 6]$, such that the union of the two paths contains all of the widget's vertices.

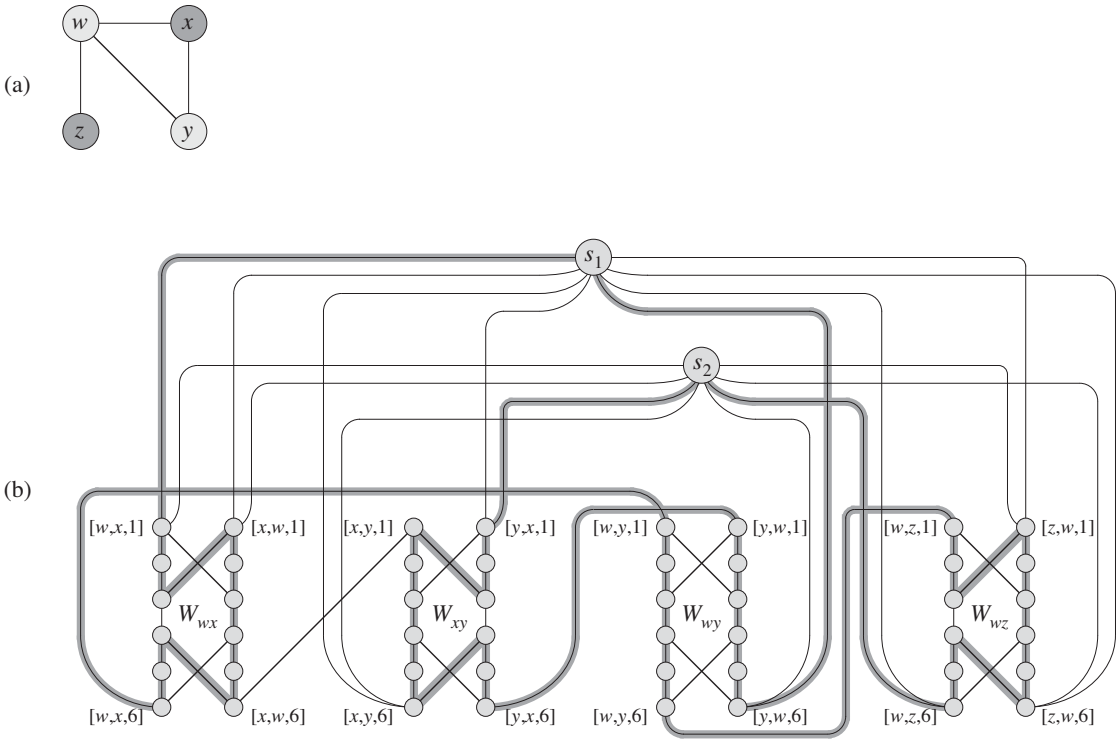


Figure 34.17 Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. **(a)** An undirected graph G with a vertex cover of size 2, consisting of the lightly shaded vertices w and y. **(b)** The undirected graph G' produced by the reduction, with the hamiltonian path corresponding to the vertex cover shaded. The vertex cover $\{w, y\}$ corresponds to edges $(s_1, [w, x, 1])$ and $(s_2, [y, x, 1])$ appearing in the hamiltonian cycle.

The only other vertices in V' other than those of widgets are ***selector vertices*** $s_1, s_2, \ldots, s_k$. We use edges incident on selector vertices in G' to select the k vertices of the cover in G.

In addition to the edges in widgets, E' contains two other types of edges, which Figure 34.17 shows. First, for each vertex $u \in V$, we add edges to join pairs of widgets in order to form a path containing all widgets corresponding to edges incident on u in G. We arbitrarily order the vertices adjacent to each vertex $u \in V$ as $u^{(1)}, u^{(2)}, \ldots, u^{(\text{degree}(u))}$, where $\text{degree}(u)$ is the number of vertices adjacent to u. We create a path in G' through all the widgets corresponding to edges incident on u by adding to E' the edges $\{([u, u^{(i)}, 6], [u, u^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u) - 1\}$. In Figure 34.17, for example, we order the vertices adjacent to w as x, y, z, and so graph G' in part (b) of the figure includes the edges

$([w, x, 6], [w, y, 1])$ and $([w, y, 6], [w, z, 1])$. For each vertex $u \in V$, these edges in G' fill in a path containing all widgets corresponding to edges incident on u in G.

The intuition behind these edges is that if we choose a vertex $u \in V$ in the vertex cover of G, we can construct a path from $[u, u^{(1)}, 1]$ to $[u, u^{(\text{degree}(u))}, 6]$ in G' that "covers" all widgets corresponding to edges incident on u. That is, for each of these widgets, say $W_{u,u^{(i)}}$, the path either includes all 12 vertices (if u is in the vertex cover but $u^{(i)}$ is not) or just the six vertices $[u, u^{(i)}, 1], [u, u^{(i)}, 2], \ldots, [u, u^{(i)}, 6]$ (if both u and $u^{(i)}$ are in the vertex cover).

The final type of edge in E' joins the first vertex $[u, u^{(1)}, 1]$ and the last vertex $[u, u^{(\text{degree}(u))}, 6]$ of each of these paths to each of the selector vertices. That is, we include the edges

$$\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ and } 1 \le j \le k\} \\ \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]) : u \in V \text{ and } 1 \le j \le k\} .$$

Next, we show that the size of G' is polynomial in the size of G, and hence we can construct G' in time polynomial in the size of G. The vertices of G' are those in the widgets, plus the selector vertices. With 12 vertices per widget, plus $k \le |V|$ selector vertices, we have a total of

$$\begin{aligned} |V'| &= 12\,|E| + k \\ &\le 12\,|E| + |V| \end{aligned}$$

vertices. The edges of G' are those in the widgets, those that go between widgets, and those connecting selector vertices to widgets. Each widget contains 14 edges, totaling $14\,|E|$ in all widgets. For each vertex $u \in V$, graph G' has $\text{degree}(u) - 1$ edges going between widgets, so that summed over all vertices in V,

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2\,|E| - |V|$$

edges go between widgets. Finally, G' has two edges for each pair consisting of a selector vertex and a vertex of V, totaling $2k\,|V|$ such edges. The total number of edges of G' is therefore

$$\begin{aligned} |E'| &= (14\,|E|) + (2\,|E| - |V|) + (2k\,|V|) \\ &= 16\,|E| + (2k - 1)\,|V| \\ &\le 16\,|E| + (2\,|V| - 1)\,|V| \ . \end{aligned}$$

Now we show that the transformation from graph G to G' is a reduction. That is, we must show that G has a vertex cover of size k if and only if G' has a hamiltonian cycle.

Suppose that $G = (V, E)$ has a vertex cover $V^* \subseteq V$ of size k. Let $V^* = \{u_1, u_2, \ldots, u_k\}$. As Figure 34.17 shows, we form a hamiltonian cycle in G' by including the following edges[10] for each vertex $u_j \in V^*$. Include edges $\left\{([u_j, u_j^{(i)}, 6], [u_j, u_j^{(i+1)}, 1]) : 1 \le i \le \text{degree}(u_j) - 1\right\}$, which connect all widgets corresponding to edges incident on u_j. We also include the edges within these widgets as Figures 34.16(b)–(d) show, depending on whether the edge is covered by one or two vertices in V^*. The hamiltonian cycle also includes the edges

$$\begin{aligned}\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \le j \le k\} \\ \cup \{(s_{j+1}, [u_j, u_j^{(\text{degree}(u_j))}, 6]) : 1 \le j \le k-1\} \\ \cup \{(s_1, [u_k, u_k^{(\text{degree}(u_k))}, 6])\} .\end{aligned}$$

By inspecting Figure 34.17, you can verify that these edges form a cycle. The cycle starts at s_1, visits all widgets corresponding to edges incident on u_1, then visits s_2, visits all widgets corresponding to edges incident on u_2, and so on, until it returns to s_1. The cycle visits each widget either once or twice, depending on whether one or two vertices of V^* cover its corresponding edge. Because V^* is a vertex cover for G, each edge in E is incident on some vertex in V^*, and so the cycle visits each vertex in each widget of G'. Because the cycle also visits every selector vertex, it is hamiltonian.

Conversely, suppose that $G' = (V', E')$ has a hamiltonian cycle $C \subseteq E'$. We claim that the set

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ for some } 1 \le j \le k\} \tag{34.4}$$

is a vertex cover for G. To see why, partition C into maximal paths that start at some selector vertex s_i, traverse an edge $(s_i, [u, u^{(1)}, 1])$ for some $u \in V$, and end at a selector vertex s_j without passing through any other selector vertex. Let us call each such path a "cover path." From how G' is constructed, each cover path must start at some s_i, take the edge $(s_i, [u, u^{(1)}, 1])$ for some vertex $u \in V$, pass through all the widgets corresponding to edges in E incident on u, and then end at some selector vertex s_j. We refer to this cover path as p_u, and by equation (34.4), we put u into V^*. Each widget visited by p_u must be W_{uv} or W_{vu} for some $v \in V$. For each widget visited by p_u, its vertices are visited by either one or two cover paths. If they are visited by one cover path, then edge $(u, v) \in E$ is covered in G by vertex u. If two cover paths visit the widget, then the other cover path must be p_v, which implies that $v \in V^*$, and edge $(u, v) \in E$ is covered by both u and v.

[10] Technically, we define a cycle in terms of vertices rather than edges (see Section B.4). In the interest of clarity, we abuse notation here and define the hamiltonian cycle in terms of edges.

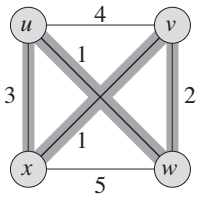


Figure 34.18 An instance of the traveling-salesman problem. Shaded edges represent a minimum-cost tour, with cost 7.

Because each vertex in each widget is visited by some cover path, we see that each edge in E is covered by some vertex in V^*. ■

34.5.4 The traveling-salesman problem

In the ***traveling-salesman problem***, which is closely related to the hamiltonian-cycle problem, a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a ***tour***, or hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. For example, in Figure 34.18, a minimum-cost tour is $\langle u, w, v, x, u\rangle$, with cost 7. The formal language for the corresponding decision problem is

$$\begin{aligned}\text{TSP} = \{\langle G, c, k\rangle : \ & G = (V, E) \text{ is a complete graph,} \\ & c \text{ is a function from } V \times V \to \mathbb{Z}, \\ & k \in \mathbb{Z}, \text{ and} \\ & G \text{ has a traveling-salesman tour with cost at most } k\}\ .\end{aligned}$$

The following theorem shows that a fast algorithm for the traveling-salesman problem is unlikely to exist.

Theorem 34.14
The traveling-salesman problem is NP-complete.

Proof We first show that TSP belongs to NP. Given an instance of the problem, we use as a certificate the sequence of n vertices in the tour. The verification algorithm checks that this sequence contains each vertex exactly once, sums up the edge costs, and checks whether the sum is at most k. This process can certainly be done in polynomial time.

To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq_P$ TSP. Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and we define the cost function c by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E , \\ 1 & \text{if } (i, j) \notin E . \end{cases}$$

(Note that because G is undirected, it has no self-loops, and so $c(v, v) = 1$ for all vertices $v \in V$.) The instance of TSP is then $\langle G', c, 0\rangle$, which we can easily create in polynomial time.

We now show that graph G has a hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a hamiltonian cycle in graph G. ■

34.5.5 The subset-sum problem

We next consider an arithmetic NP-complete problem. In the ***subset-sum problem***, we are given a finite set S of positive integers and an integer ***target*** $t > 0$. We ask whether there exists a subset $S' \subseteq S$ whose elements sum to t. For example, if $S = \{1, 2, 7, 14, 49, 98, 343, 686, 2409, 2793, 16808, 17206, 117705, 117993\}$ and $t = 138457$, then the subset $S' = \{1, 2, 7, 98, 343, 686, 2409, 17206, 117705\}$ is a solution.

As usual, we define the problem as a language:

$$\text{SUBSET-SUM} = \{\langle S, t\rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \textstyle\sum_{s \in S'} s\} .$$

As with any arithmetic problem, it is important to recall that our standard encoding assumes that the input integers are coded in binary. With this assumption in mind, we can show that the subset-sum problem is unlikely to have a fast algorithm.

Theorem 34.15
The subset-sum problem is NP-complete.

Proof To show that SUBSET-SUM is in NP, for an instance $\langle S, t\rangle$ of the problem, we let the subset S' be the certificate. A verification algorithm can check whether $t = \sum_{s \in S'} s$ in polynomial time.

We now show that 3-CNF-SAT $\leq_P$ SUBSET-SUM. Given a 3-CNF formula ϕ over variables $x_1, x_2, \ldots, x_n$ with clauses $C_1, C_2, \ldots, C_k$, each containing exactly

three distinct literals, the reduction algorithm constructs an instance $\langle S, t \rangle$ of the subset-sum problem such that ϕ is satisfiable if and only if there exists a subset of S whose sum is exactly t. Without loss of generality, we make two simplifying assumptions about the formula ϕ. First, no clause contains both a variable and its negation, for such a clause is automatically satisfied by any assignment of values to the variables. Second, each variable appears in at least one clause, because it does not matter what value is assigned to a variable that appears in no clauses.

The reduction creates two numbers in set S for each variable x_i and two numbers in S for each clause C_j. We shall create numbers in base 10, where each number contains $n+k$ digits and each digit corresponds to either one variable or one clause. Base 10 (and other bases, as we shall see) has the property we need of preventing carries from lower digits to higher digits.

As Figure 34.19 shows, we construct set S and target t as follows. We label each digit position by either a variable or a clause. The least significant k digits are labeled by the clauses, and the most significant n digits are labeled by variables.

- The target t has a 1 in each digit labeled by a variable and a 4 in each digit labeled by a clause.
- For each variable x_i, set S contains two integers v_i and v'_i. Each of v_i and v'_i has a 1 in the digit labeled by x_i and 0s in the other variable digits. If literal x_i appears in clause C_j, then the digit labeled by C_j in v_i contains a 1. If literal $\neg x_i$ appears in clause C_j, then the digit labeled by C_j in v'_i contains a 1. All other digits labeled by clauses in v_i and v'_i are 0.

  All v_i and v'_i values in set S are unique. Why? For $l \neq i$, no v_l or v'_l values can equal v_i and v'_i in the most significant n digits. Furthermore, by our simplifying assumptions above, no v_i and v'_i can be equal in all k least significant digits. If v_i and v'_i were equal, then x_i and $\neg x_i$ would have to appear in exactly the same set of clauses. But we assume that no clause contains both x_i and $\neg x_i$ and that either x_i or $\neg x_i$ appears in some clause, and so there must be some clause C_j for which v_i and v'_i differ.
- For each clause C_j, set S contains two integers s_j and s'_j. Each of s_j and s'_j has 0s in all digits other than the one labeled by C_j. For s_j, there is a 1 in the C_j digit, and s'_j has a 2 in this digit. These integers are "slack variables," which we use to get each clause-labeled digit position to add to the target value of 4.

  Simple inspection of Figure 34.19 demonstrates that all s_j and s'_j values in S are unique in set S.

Note that the greatest sum of digits in any one digit position is 6, which occurs in the digits labeled by clauses (three 1s from the v_i and v'_i values, plus 1 and 2 from

		x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	=	1	0	0	1	0	0	1
v'_1	=	1	0	0	0	1	1	0
v_2	=	0	1	0	0	0	0	1
v'_2	=	0	1	0	1	1	1	0
v_3	=	0	0	1	0	0	1	1
v'_3	=	0	0	1	1	1	0	0
s_1	=	0	0	0	1	0	0	0
s'_1	=	0	0	0	2	0	0	0
s_2	=	0	0	0	0	1	0	0
s'_2	=	0	0	0	0	2	0	0
s_3	=	0	0	0	0	0	1	0
s'_3	=	0	0	0	0	0	2	0
s_4	=	0	0	0	0	0	0	1
s'_4	=	0	0	0	0	0	0	2
t	=	1	1	1	4	4	4	4

Figure 34.19 The reduction of 3-CNF-SAT to SUBSET-SUM. The formula in 3-CNF is $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, $C_3 = (\neg x_1 \vee \neg x_2 \vee x_3)$, and $C_4 = (x_1 \vee x_2 \vee x_3)$. A satisfying assignment of ϕ is $\langle x_1 = 0, x_2 = 0, x_3 = 1\rangle$. The set S produced by the reduction consists of the base-10 numbers shown; reading from top to bottom, $S = \{1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2\}$. The target t is 1114444. The subset $S' \subseteq S$ is lightly shaded, and it contains v'_1, v'_2, and v_3, corresponding to the satisfying assignment. It also contains slack variables s_1, s'_1, s'_2, s_3, s_4, and s'_4 to achieve the target value of 4 in the digits labeled by C_1 through C_4.

the s_j and s'_j values). Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits.[11]

We can perform the reduction in polynomial time. The set S contains $2n + 2k$ values, each of which has $n + k$ digits, and the time to produce each digit is polynomial in $n + k$. The target t has $n + k$ digits, and the reduction produces each in constant time.

We now show that the 3-CNF formula ϕ is satisfiable if and only if there exists a subset $S' \subseteq S$ whose sum is t. First, suppose that ϕ has a satisfying assignment. For $i = 1, 2, \ldots, n$, if $x_i = 1$ in this assignment, then include v_i in S'. Otherwise, include v'_i. In other words, we include in S' exactly the v_i and v'_i values that cor-

[11] In fact, any base b, where $b \geq 7$, would work. The instance at the beginning of this subsection is the set S and target t in Figure 34.19 interpreted in base 7, with S listed in sorted order.

respond to literals with the value 1 in the satisfying assignment. Having included either v_i or v'_i, but not both, for all i, and having put 0 in the digits labeled by variables in all s_j and s'_j, we see that for each variable-labeled digit, the sum of the values of S' must be 1, which matches those digits of the target t. Because each clause is satisfied, the clause contains some literal with the value 1. Therefore, each digit labeled by a clause has at least one 1 contributed to its sum by a v_i or v'_i value in S'. In fact, 1, 2, or 3 literals may be 1 in each clause, and so each clause-labeled digit has a sum of 1, 2, or 3 from the v_i and v'_i values in S'. In Figure 34.19 for example, literals $\neg x_1$, $\neg x_2$, and x_3 have the value 1 in a satisfying assignment. Each of clauses C_1 and C_4 contains exactly one of these literals, and so together v'_1, v'_2, and v_3 contribute 1 to the sum in the digits for C_1 and C_4. Clause C_2 contains two of these literals, and v'_1, v'_2, and v_3 contribute 2 to the sum in the digit for C_2. Clause C_3 contains all three of these literals, and v'_1, v'_2, and v_3 contribute 3 to the sum in the digit for C_3. We achieve the target of 4 in each digit labeled by clause C_j by including in S' the appropriate nonempty subset of slack variables $\{s_j, s'_j\}$. In Figure 34.19, S' includes s_1, s'_1, s'_2, s_3, s_4, and s'_4. Since we have matched the target in all digits of the sum, and no carries can occur, the values of S' sum to t.

Now, suppose that there is a subset $S' \subseteq S$ that sums to t. The subset S' must include exactly one of v_i and v'_i for each $i = 1, 2, \ldots, n$, for otherwise the digits labeled by variables would not sum to 1. If $v_i \in S'$, we set $x_i = 1$. Otherwise, $v'_i \in S'$, and we set $x_i = 0$. We claim that every clause C_j, for $j = 1, 2, \ldots, k$, is satisfied by this assignment. To prove this claim, note that to achieve a sum of 4 in the digit labeled by C_j, the subset S' must include at least one v_i or v'_i value that has a 1 in the digit labeled by C_j, since the contributions of the slack variables s_j and s'_j together sum to at most 3. If S' includes a v_i that has a 1 in C_j's position, then the literal x_i appears in clause C_j. Since we have set $x_i = 1$ when $v_i \in S'$, clause C_j is satisfied. If S' includes a v'_i that has a 1 in that position, then the literal $\neg x_i$ appears in C_j. Since we have set $x_i = 0$ when $v'_i \in S'$, clause C_j is again satisfied. Thus, all clauses of ϕ are satisfied, which completes the proof. ■

Exercises

34.5-1

The ***subgraph-isomorphism problem*** takes two undirected graphs G_1 and G_2, and it asks whether G_1 is isomorphic to a subgraph of G_2. Show that the subgraph-isomorphism problem is NP-complete.

34.5-2

Given an integer $m \times n$ matrix A and an integer m-vector b, the ***0-1 integer-programming problem*** asks whether there exists an integer n-vector x with ele-

ments in the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. (*Hint:* Reduce from 3-CNF-SAT.)

34.5-3
The ***integer linear-programming problem*** is like the 0-1 integer-programming problem given in Exercise 34.5-2, except that the values of the vector x may be any integers rather than just 0 or 1. Assuming that the 0-1 integer-programming problem is NP-hard, show that the integer linear-programming problem is NP-complete.

34.5-4
Show how to solve the subset-sum problem in polynomial time if the target value t is expressed in unary.

34.5-5
The ***set-partition problem*** takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\overline{A} = S - A$ such that $\sum_{x \in A} x = \sum_{x \in \overline{A}} x$. Show that the set-partition problem is NP-complete.

34.5-6
Show that the hamiltonian-path problem is NP-complete.

34.5-7
The ***longest-simple-cycle problem*** is the problem of determining a simple cycle (no repeated vertices) of maximum length in a graph. Formulate a related decision problem, and show that the decision problem is NP-complete.

34.5-8
In the ***half 3-CNF satisfiability*** problem, we are given a 3-CNF formula ϕ with n variables and m clauses, where m is even. We wish to determine whether there exists a truth assignment to the variables of ϕ such that exactly half the clauses evaluate to 0 and exactly half the clauses evaluate to 1. Prove that the half 3-CNF satisfiability problem is NP-complete.

Problems

34-1 Independent set
An ***independent set*** of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that each edge in E is incident on at most one vertex in V'. The ***independent-set problem*** is to find a maximum-size independent set in G.

a. Formulate a related decision problem for the independent-set problem, and prove that it is NP-complete. (*Hint:* Reduce from the clique problem.)

b. Suppose that you are given a "black-box" subroutine to solve the decision problem you defined in part (a). Give an algorithm to find an independent set of maximum size. The running time of your algorithm should be polynomial in $|V|$ and $|E|$, counting queries to the black box as a single step.

Although the independent-set decision problem is NP-complete, certain special cases are polynomial-time solvable.

c. Give an efficient algorithm to solve the independent-set problem when each vertex in G has degree 2. Analyze the running time, and prove that your algorithm works correctly.

d. Give an efficient algorithm to solve the independent-set problem when G is bipartite. Analyze the running time, and prove that your algorithm works correctly. (*Hint:* Use the results of Section 26.3.)

34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a polynomial-time algorithm, or prove that the problem is NP-complete. The input in each case is a list of the n items in the bag, along with the value of each.

a. The bag contains n coins, but only 2 different denominations: some coins are worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide the money exactly evenly.

b. The bag contains n coins, with an arbitrary number of different denominations, but each denomination is a nonnegative integer power of 2, i.e., the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.

c. The bag contains n checks, which are, in an amazing coincidence, made out to "Bonnie or Clyde." They wish to divide the checks so that they each get the exact same amount of money.

d. The bag contains n checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

34-3 Graph coloring

Mapmakers try to use as few colors as possible when coloring countries on a map, as long as no two countries that share a border have the same color. We can model this problem with an undirected graph $G = (V, E)$ in which each vertex represents a country and vertices whose respective countries share a border are adjacent. Then, a ***k-coloring*** is a function $c : V \to \{1, 2, \ldots, k\}$ such that $c(u) \neq c(v)$ for every edge $(u, v) \in E$. In other words, the numbers $1, 2, \ldots, k$ represent the k colors, and adjacent vertices must have different colors. The ***graph-coloring problem*** is to determine the minimum number of colors needed to color a given graph.

a. Give an efficient algorithm to determine a 2-coloring of a graph, if one exists.

b. Cast the graph-coloring problem as a decision problem. Show that your decision problem is solvable in polynomial time if and only if the graph-coloring problem is solvable in polynomial time.

c. Let the language 3-COLOR be the set of graphs that can be 3-colored. Show that if 3-COLOR is NP-complete, then your decision problem from part (b) is NP-complete.

To prove that 3-COLOR is NP-complete, we use a reduction from 3-CNF-SAT. Given a formula ϕ of m clauses on n variables $x_1, x_2, \ldots, x_n$, we construct a graph $G = (V, E)$ as follows. The set V consists of a vertex for each variable, a vertex for the negation of each variable, 5 vertices for each clause, and 3 special vertices: TRUE, FALSE, and RED. The edges of the graph are of two types: "literal" edges that are independent of the clauses and "clause" edges that depend on the clauses. The literal edges form a triangle on the special vertices and also form a triangle on x_i, $\neg x_i$, and RED for $i = 1, 2, \ldots, n$.

d. Argue that in any 3-coloring c of a graph containing the literal edges, exactly one of a variable and its negation is colored $c(\text{TRUE})$ and the other is colored $c(\text{FALSE})$. Argue that for any truth assignment for ϕ, there exists a 3-coloring of the graph containing just the literal edges.

The widget shown in Figure 34.20 helps to enforce the condition corresponding to a clause $(x \vee y \vee z)$. Each clause requires a unique copy of the 5 vertices that are heavily shaded in the figure; they connect as shown to the literals of the clause and the special vertex TRUE.

e. Argue that if each of x, y, and z is colored $c(\text{TRUE})$ or $c(\text{FALSE})$, then the widget is 3-colorable if and only if at least one of x, y, or z is colored $c(\text{TRUE})$.

f. Complete the proof that 3-COLOR is NP-complete.

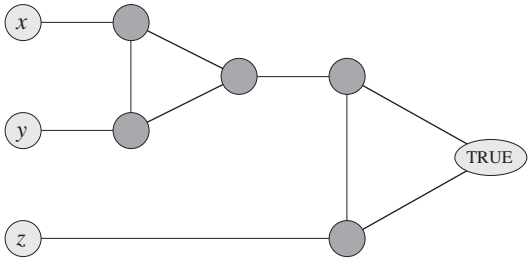


Figure 34.20 The widget corresponding to a clause $(x \vee y \vee z)$, used in Problem 34-3.

34-4 Scheduling with profits and deadlines

Suppose that we have one machine and a set of n tasks $a_1, a_2, \ldots, a_n$, each of which requires time on the machine. Each task a_j requires t_j time units on the machine (its processing time), yields a profit of p_j, and has a deadline d_j. The machine can process only one task at a time, and task a_j must run without interruption for t_j consecutive time units. If we complete task a_j by its deadline d_j, we receive a profit p_j, but if we complete it after its deadline, we receive no profit. As an optimization problem, we are given the processing times, profits, and deadlines for a set of n tasks, and we wish to find a schedule that completes all the tasks and returns the greatest amount of profit. The processing times, profits, and deadlines are all nonnegative numbers.

a. State this problem as a decision problem.

b. Show that the decision problem is NP-complete.

c. Give a polynomial-time algorithm for the decision problem, assuming that all processing times are integers from 1 to n. (*Hint:* Use dynamic programming.)

d. Give a polynomial-time algorithm for the optimization problem, assuming that all processing times are integers from 1 to n.

Chapter notes

The book by Garey and Johnson [129] provides a wonderful guide to NP-completeness, discussing the theory at length and providing a catalogue of many problems that were known to be NP-complete in 1979. The proof of Theorem 34.13 is adapted from their book, and the list of NP-complete problem domains at the beginning of Section 34.5 is drawn from their table of contents. Johnson wrote a series

of 23 columns in the *Journal of Algorithms* between 1981 and 1992 reporting new developments in NP-completeness. Hopcroft, Motwani, and Ullman [177], Lewis and Papadimitriou [236], Papadimitriou [270], and Sipser [317] have good treatments of NP-completeness in the context of complexity theory. NP-completeness and several reductions also appear in books by Aho, Hopcroft, and Ullman [5]; Dasgupta, Papadimitriou, and Vazirani [82]; Johnsonbaugh and Schaefer [193]; and Kleinberg and Tardos [208].

The class P was introduced in 1964 by Cobham [72] and, independently, in 1965 by Edmonds [100], who also introduced the class NP and conjectured that $P \neq NP$. The notion of NP-completeness was proposed in 1971 by Cook [75], who gave the first NP-completeness proofs for formula satisfiability and 3-CNF satisfiability. Levin [234] independently discovered the notion, giving an NP-completeness proof for a tiling problem. Karp [199] introduced the methodology of reductions in 1972 and demonstrated the rich variety of NP-complete problems. Karp's paper included the original NP-completeness proofs of the clique, vertex-cover, and hamiltonian-cycle problems. Since then, thousands of problems have been proven to be NP-complete by many researchers. In a talk at a meeting celebrating Karp's 60th birthday in 1995, Papadimitriou remarked, "about 6000 papers each year have the term 'NP-complete' on their title, abstract, or list of keywords. This is more than each of the terms 'compiler,' 'database,' 'expert,' 'neural network,' or 'operating system.' "

Recent work in complexity theory has shed light on the complexity of computing approximate solutions. This work gives a new definition of NP using "probabilistically checkable proofs." This new definition implies that for problems such as clique, vertex cover, the traveling-salesman problem with the triangle inequality, and many others, computing good approximate solutions is NP-hard and hence no easier than computing optimal solutions. An introduction to this area can be found in Arora's thesis [20]; a chapter by Arora and Lund in Hochbaum [172]; a survey article by Arora [21]; a book edited by Mayr, Prömel, and Steger [246]; and a survey article by Johnson [191].

35 Approximation Algorithms

Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we don't know how to find an optimal solution in polynomial time. Even if a problem is NP-complete, there may be hope. We have at least three ways to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that we can solve in polynomial time. Third, we might come up with approaches to find *near-optimal* solutions in polynomial time (either in the worst case or the expected case). In practice, near-optimality is often good enough. We call an algorithm that returns near-optimal solutions an ***approximation algorithm***. This chapter presents polynomial-time approximation algorithms for several NP-complete problems.

Performance ratios for approximation algorithms

Suppose that we are working on an optimization problem in which each potential solution has a positive cost, and we wish to find a near-optimal solution. Depending on the problem, we may define an optimal solution as one with maximum possible cost or one with minimum possible cost; that is, the problem may be either a maximization or a minimization problem.

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) \ . \tag{35.1}$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a ***$\rho(n)$-approximation algorithm***. The definitions of the approximation ratio and of a $\rho(n)$-approximation algorithm apply to both minimization and maximization problems. For a maximization problem, $0 < C \le C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approximate

solution. Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution. Because we assume that all solutions have positive cost, these ratios are always well defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$. Therefore, a 1-approximation algorithm[1] produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution that is much worse than optimal.

For many problems, we have polynomial-time approximation algorithms with small constant approximation ratios, although for other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n. An example of such a problem is the set-cover problem presented in Section 35.3.

Some NP-complete problems allow polynomial-time approximation algorithms that can achieve increasingly better approximation ratios by using more and more computation time. That is, we can trade computation time for the quality of the approximation. An example is the subset-sum problem studied in Section 35.5. This situation is important enough to deserve a name of its own.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

The running time of a polynomial-time approximation scheme can increase very rapidly as ϵ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{2/\epsilon})$. Ideally, if ϵ decreases by a constant factor, the running time to achieve the desired approximation should not increase by more than a constant factor (though not necessarily the same constant factor by which ϵ decreased).

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial in both $1/\epsilon$ and the size n of the input instance. For example, the scheme might have a running time of $O((1/\epsilon)^2 n^3)$. With such a scheme, any constant-factor decrease in ϵ comes with a corresponding constant-factor increase in the running time.

---

[1] When the approximation ratio is independent of n, we use the terms "approximation ratio of ρ" and "ρ-approximation algorithm," indicating no dependence on n.

Chapter outline

The first four sections of this chapter present some examples of polynomial-time approximation algorithms for NP-complete problems, and the fifth section presents a fully polynomial-time approximation scheme. Section 35.1 begins with a study of the vertex-cover problem, an NP-complete minimization problem that has an approximation algorithm with an approximation ratio of 2. Section 35.2 presents an approximation algorithm with an approximation ratio of 2 for the case of the traveling-salesman problem in which the cost function satisfies the triangle inequality. It also shows that without the triangle inequality, for any constant $\rho \geq 1$, a ρ-approximation algorithm cannot exist unless P $=$ NP. In Section 35.3, we show how to use a greedy method as an effective approximation algorithm for the set-covering problem, obtaining a covering whose cost is at worst a logarithmic factor larger than the optimal cost. Section 35.4 presents two more approximation algorithms. First we study the optimization version of 3-CNF satisfiability and give a simple randomized algorithm that produces a solution with an expected approximation ratio of $8/7$. Then we examine a weighted variant of the vertex-cover problem and show how to use linear programming to develop a 2-approximation algorithm. Finally, Section 35.5 presents a fully polynomial-time approximation scheme for the subset-sum problem.

35.1 The vertex-cover problem

Section 34.5.2 defined the vertex-cover problem and proved it NP-complete. Recall that a ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.

The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

Even though we don't know how to find an optimal vertex cover in a graph G in polynomial time, we can efficiently find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

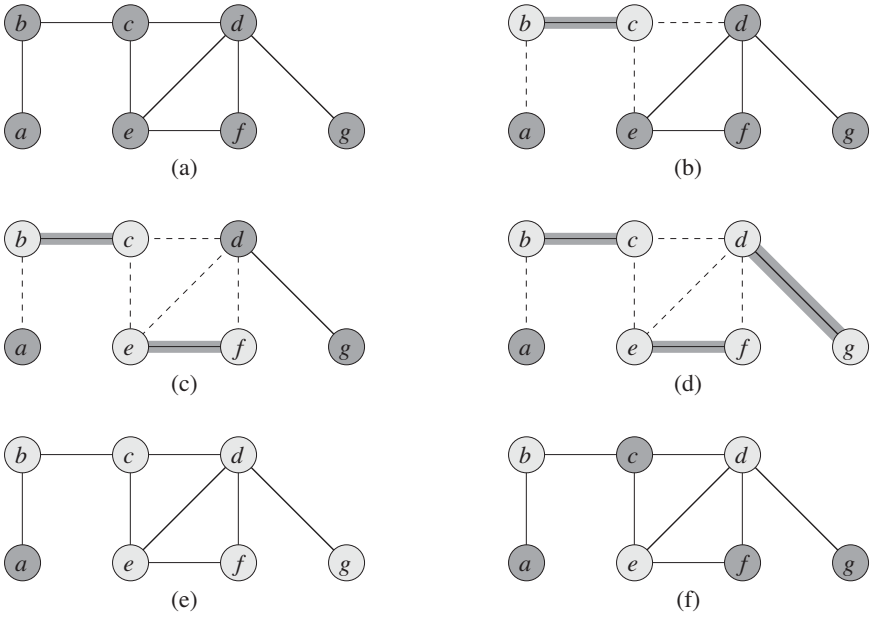


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4       let (u, v) be an arbitrary edge of E'
5       C = C ∪ {u, v}
6       remove from E' every edge incident on either u or v
7  return C
```

Figure 35.1 illustrates how APPROX-VERTEX-COVER operates on an example graph. The variable C contains the vertex cover being constructed. Line 1 initializes C to the empty set. Line 2 sets E' to be a copy of the edge set $G.E$ of the graph. The loop of lines 3–6 repeatedly picks an edge (u, v) from E', adds its

endpoints u and v to C, and deletes all edges in E' that are covered by either u or v. Finally, line 7 returns the vertex cover C. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'.

Theorem 35.1
APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof We have already shown that APPROX-VERTEX-COVER runs in polynomial time.

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in C.

To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 4 of APPROX-VERTEX-COVER picked. In order to cover the edges in A, any vertex cover—in particular, an optimal cover C^*—must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6. Thus, no two edges in A are covered by the same vertex from C^*, and we have the lower bound

$$|C^*| \geq |A| \tag{35.2}$$

on the size of an optimal vertex cover. Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2\,|A| \ . \tag{35.3}$$

Combining equations (35.2) and (35.3), we obtain

$$\begin{aligned} |C| &= 2\,|A| \\ &\leq 2\,|C^*| \ , \end{aligned}$$

thereby proving the theorem. ■

Let us reflect on this proof. At first, you might wonder how we can possibly prove that the size of the vertex cover returned by APPROX-VERTEX-COVER is at most twice the size of an optimal vertex cover, when we do not even know the size of an optimal vertex cover. Instead of requiring that we know the exact size of an optimal vertex cover, we rely on a lower bound on the size. As Exercise 35.1-2 asks you to show, the set A of edges that line 4 of APPROX-VERTEX-COVER selects is actually a maximal matching in the graph G. (A ***maximal matching*** is a matching that is not a proper subset of any other matching.) The size of a maximal matching

is, as we argued in the proof of Theorem 35.1, a lower bound on the size of an optimal vertex cover. The algorithm returns a vertex cover whose size is at most twice the size of the maximal matching A. By relating the size of the solution returned to the lower bound, we obtain our approximation ratio. We will use this methodology in later sections as well.

Exercises

35.1-1
Give an example of a graph for which APPROX-VERTEX-COVER always yields a suboptimal solution.

35.1-2
Prove that the set of edges picked in line 4 of APPROX-VERTEX-COVER forms a maximal matching in the graph G.

35.1-3 ⋆
Professor Bündchen proposes the following heuristic to solve the vertex-cover problem. Repeatedly select a vertex of highest degree, and remove all of its incident edges. Give an example to show that the professor's heuristic does not have an approximation ratio of 2. (*Hint:* Try a bipartite graph with vertices of uniform degree on the left and vertices of varying degree on the right.)

35.1-4
Give an efficient greedy algorithm that finds an optimal vertex cover for a tree in linear time.

35.1-5
From the proof of Theorem 34.12, we know that the vertex-cover problem and the NP-complete clique problem are complementary in the sense that an optimal vertex cover is the complement of a maximum-size clique in the complement graph. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for the clique problem? Justify your answer.

35.2 The traveling-salesman problem

In the traveling-salesman problem introduced in Section 34.5.4, we are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we must find a hamiltonian cycle (a tour) of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$$c(A) = \sum_{(u,v)\in A} c(u,v) .$$

In many practical situations, the least costly way to go from a place u to a place w is to go directly, with no intermediate steps. Put another way, cutting out an intermediate stop never increases the cost. We formalize this notion by saying that the cost function c satisfies the ***triangle inequality*** if, for all vertices $u, v, w \in V$,

$$c(u,w) \le c(u,v) + c(v,w) .$$

The triangle inequality seems as though it should naturally hold, and it is automatically satisfied in several applications. For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied. Furthermore, many cost functions other than euclidean distance satisfy the triangle inequality.

As Exercise 35.2-2 shows, the traveling-salesman problem is NP-complete even if we require that the cost function satisfy the triangle inequality. Thus, we should not expect to find a polynomial-time algorithm for solving this problem exactly. Instead, we look for good approximation algorithms.

In Section 35.2.1, we examine a 2-approximation algorithm for the traveling-salesman problem with the triangle inequality. In Section 35.2.2, we show that without the triangle inequality, a polynomial-time approximation algorithm with a constant approximation ratio does not exist unless P = NP.

35.2.1 The traveling-salesman problem with the triangle inequality

Applying the methodology of the previous section, we shall first compute a structure—a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesman tour. We shall then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. The following algorithm implements this approach, calling the minimum-spanning-tree algorithm MST-PRIM from Section 23.2 as a subroutine. The parameter G is a complete undirected graph, and the cost function c satisfies the triangle inequality.

APPROX-TSP-TOUR(G, c)

1 select a vertex $r \in G.V$ to be a "root" vertex
2 compute a minimum spanning tree T for G from root r
  using MST-PRIM(G, c, r)
3 let H be a list of vertices, ordered according to when they are first visited
  in a preorder tree walk of T
4 **return** the hamiltonian cycle H

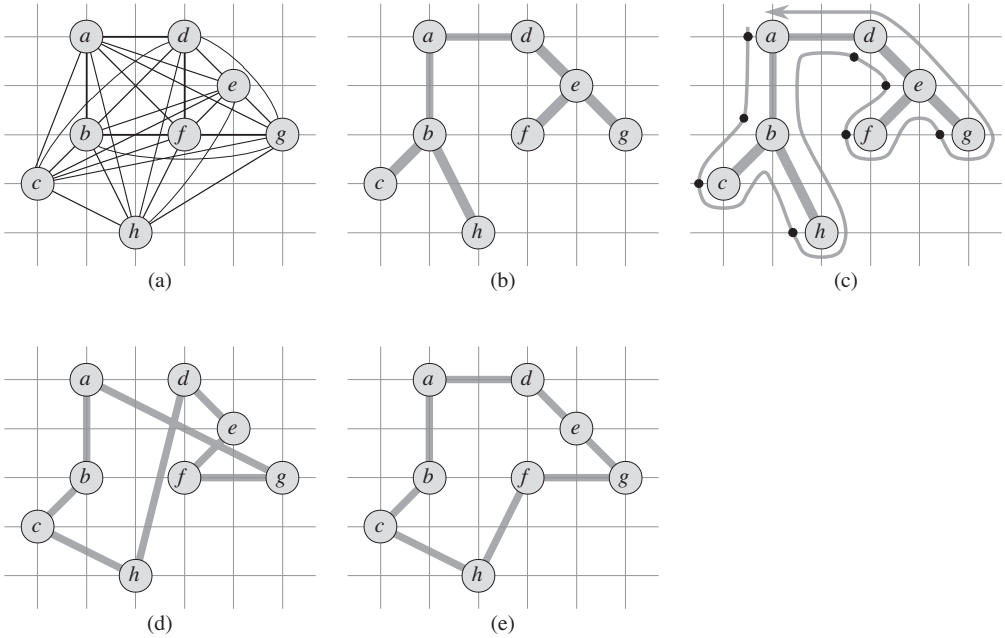


Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

Recall from Section 12.1 that a preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children.

Figure 35.2 illustrates the operation of APPROX-TSP-TOUR. Part (a) of the figure shows a complete undirected graph, and part (b) shows the minimum spanning tree T grown from root vertex a by MST-PRIM. Part (c) shows how a preorder walk of T visits the vertices, and part (d) displays the corresponding tour, which is the tour returned by APPROX-TSP-TOUR. Part (e) displays an optimal tour, which is about 23% shorter.

By Exercise 23.2-2, even with a simple implementation of MST-PRIM, the running time of APPROX-TSP-TOUR is $\Theta(V^2)$. We now show that if the cost function for an instance of the traveling-salesman problem satisfies the triangle inequality, then APPROX-TSP-TOUR returns a tour whose cost is not more than twice the cost of an optimal tour.

Theorem 35.2
APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality.

Proof We have already seen that APPROX-TSP-TOUR runs in polynomial time.

Let H^* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge from a tour, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree T computed in line 2 of APPROX-TSP-TOUR provides a lower bound on the cost of an optimal tour:

$$c(T) \leq c(H^*) . \tag{35.4}$$

A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this full walk W. The full walk of our example gives the order

$$a, b, c, b, h, b, a, d, e, f, e, g, e, d, a .$$

Since the full walk traverses every edge of T exactly twice, we have (extending our definition of the cost c in the natural manner to handle multisets of edges)

$$c(W) = 2c(T) . \tag{35.5}$$

Inequality (35.4) and equation (35.5) imply that

$$c(W) \leq 2c(H^*) , \tag{35.6}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.

Unfortunately, the full walk W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase. (If we delete a vertex v from W between visits to u and w, the resulting ordering specifies going directly from u to w.) By repeatedly applying this operation, we can remove from W all but the first visit to each vertex. In our example, this leaves the ordering

$$a, b, c, h, d, e, f, g .$$

This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since ev-

ery vertex is visited exactly once, and in fact it is the cycle computed by APPROX-TSP-TOUR. Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \le c(W) . \tag{35.7}$$

Combining inequalities (35.6) and (35.7) gives $c(H) \le 2c(H^*)$, which completes the proof. ■

In spite of the nice approximation ratio provided by Theorem 35.2, APPROX-TSP-TOUR is usually not the best practical choice for this problem. There are other approximation algorithms that typically perform much better in practice. (See the references at the end of this chapter.)

35.2.2 The general traveling-salesman problem

If we drop the assumption that the cost function c satisfies the triangle inequality, then we cannot find good approximate tours in polynomial time unless P = NP.

Theorem 35.3
If P ≠ NP, then for any constant $\rho \ge 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

Proof The proof is by contradiction. Suppose to the contrary that for some number $\rho \ge 1$, there is a polynomial-time approximation algorithm A with approximation ratio ρ. Without loss of generality, we assume that ρ is an integer, by rounding it up if necessary. We shall then show how to use A to solve instances of the hamiltonian-cycle problem (defined in Section 34.2) in polynomial time. Since Theorem 34.13 tells us that the hamiltonian-cycle problem is NP-complete, Theorem 34.4 implies that if we can solve it in polynomial time, then P = NP.

Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem. We wish to determine efficiently whether G contains a hamiltonian cycle by making use of the hypothesized approximation algorithm A. We turn G into an instance of the traveling-salesman problem as follows. Let $G' = (V, E')$ be the complete graph on V; that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \ne v\} .$$

Assign an integer cost to each edge in E' as follows:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E , \\ \rho |V| + 1 & \text{otherwise} . \end{cases}$$

We can create representations of G' and c from a representation of G in time polynomial in $|V|$ and $|E|$.

Now, consider the traveling-salesman problem (G', c). If the original graph G has a hamiltonian cycle H, then the cost function c assigns to each edge of H a cost of 1, and so (G', c) contains a tour of cost $|V|$. On the other hand, if G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E. But any tour that uses an edge not in E has a cost of at least

$$\begin{aligned}(\rho|V| + 1) + (|V| - 1) &= \rho|V| + |V| \\ &> \rho|V| \ .\end{aligned}$$

Because edges not in G are so costly, there is a gap of at least $\rho|V|$ between the cost of a tour that is a hamiltonian cycle in G (cost $|V|$) and the cost of any other tour (cost at least $\rho|V| + |V|$). Therefore, the cost of a tour that is not a hamiltonian cycle in G is at least a factor of $\rho + 1$ greater than the cost of a tour that is a hamiltonian cycle in G.

Now, suppose that we apply the approximation algorithm A to the traveling-salesman problem (G', c). Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if G contains a hamiltonian cycle, then A must return it. If G has no hamiltonian cycle, then A returns a tour of cost more than $\rho|V|$. Therefore, we can use A to solve the hamiltonian-cycle problem in polynomial time. ■

The proof of Theorem 35.3 serves as an example of a general technique for proving that we cannot approximate a problem very well. Suppose that given an NP-hard problem X, we can produce in polynomial time a minimization problem Y such that "yes" instances of X correspond to instances of Y with value at most k (for some k), but that "no" instances of X correspond to instances of Y with value greater than ρk. Then, we have shown that, unless P = NP, there is no polynomial-time ρ-approximation algorithm for problem Y.

Exercises

35.2-1

Suppose that a complete undirected graph $G = (V, E)$ with at least 3 vertices has a cost function c that satisfies the triangle inequality. Prove that $c(u, v) \geq 0$ for all $u, v \in V$.

35.2-2

Show how in polynomial time we can transform one instance of the traveling-salesman problem into another instance whose cost function satisfies the triangle inequality. The two instances must have the same set of optimal tours. Explain why such a polynomial-time transformation does not contradict Theorem 35.3, assuming that P $\neq$ NP.

35.2-3

Consider the following ***closest-point heuristic*** for building an approximate traveling-salesman tour whose cost function satisfies the triangle inequality. Begin with a trivial cycle consisting of a single arbitrarily chosen vertex. At each step, identify the vertex u that is not on the cycle but whose distance to any vertex on the cycle is minimum. Suppose that the vertex on the cycle that is nearest u is vertex v. Extend the cycle to include u by inserting u just after v. Repeat until all vertices are on the cycle. Prove that this heuristic returns a tour whose total cost is not more than twice the cost of an optimal tour.

35.2-4

In the ***bottleneck traveling-salesman problem***, we wish to find the hamiltonian cycle that minimizes the cost of the most costly edge in the cycle. Assuming that the cost function satisfies the triangle inequality, show that there exists a polynomial-time approximation algorithm with approximation ratio 3 for this problem. (*Hint:* Show recursively that we can visit all the nodes in a bottleneck spanning tree, as discussed in Problem 23-3, exactly once by taking a full walk of the tree and skipping nodes, but without skipping more than two consecutive intermediate nodes. Show that the costliest edge in a bottleneck spanning tree has a cost that is at most the cost of the costliest edge in a bottleneck hamiltonian cycle.)

35.2-5

Suppose that the vertices for an instance of the traveling-salesman problem are points in the plane and that the cost $c(u, v)$ is the euclidean distance between points u and v. Show that an optimal tour never crosses itself.

35.3 The set-covering problem

The set-covering problem is an optimization problem that models many problems that require resources to be allocated. Its corresponding decision problem generalizes the NP-complete vertex-cover problem and is therefore also NP-hard. The approximation algorithm developed to handle the vertex-cover problem doesn't apply here, however, and so we need to try other approaches. We shall examine a simple greedy heuristic with a logarithmic approximation ratio. That is, as the size of the instance gets larger, the size of the approximate solution may grow, relative to the size of an optimal solution. Because the logarithm function grows rather slowly, however, this approximation algorithm may nonetheless give useful results.

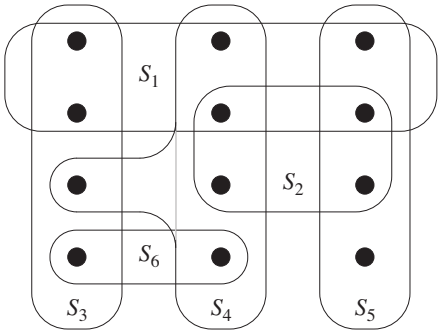


Figure 35.3 An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 black points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy algorithm produces a cover of size 4 by selecting either the sets S_1, S_4, S_5, and S_3 or the sets S_1, S_4, S_5, and S_6, in order.

An instance $(X, \mathcal{F})$ of the ***set-covering problem*** consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S \ .$$

We say that a subset $S \in \mathcal{F}$ ***covers*** its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S \ . \tag{35.8}$$

We say that any $\mathcal{C}$ satisfying equation (35.8) ***covers*** X. Figure 35.3 illustrates the set-covering problem. The size of $\mathcal{C}$ is the number of sets it contains, rather than the number of individual elements in these sets, since every subset $\mathcal{C}$ that covers X must contain all $|X|$ individual elements. In Figure 35.3, the minimum set cover has size 3.

The set-covering problem abstracts many commonly arising combinatorial problems. As a simple example, suppose that X represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem. We wish to form a committee, containing as few people as possible, such that for every requisite skill in X, at least one member of the committee has that skill. In the decision version of the set-covering problem, we ask whether a covering exists with size at most k, where k is an additional parameter specified in the problem instance. The decision version of the problem is NP-complete, as Exercise 35.3-2 asks you to show.

A greedy approximation algorithm

The greedy method works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered.

```
GREEDY-SET-COVER(X, F)
1  U = X
2  C = ∅
3  while U ≠ ∅
4      select an S ∈ F that maximizes |S ∩ U|
5      U = U − S
6      C = C ∪ {S}
7  return C
```

In the example of Figure 35.3, GREEDY-SET-COVER adds to $\mathcal{C}$, in order, the sets S_1, S_4, and S_5, followed by either S_3 or S_6.

The algorithm works as follows. The set U contains, at each stage, the set of remaining uncovered elements. The set $\mathcal{C}$ contains the cover being constructed. Line 4 is the greedy decision-making step, choosing a subset S that covers as many uncovered elements as possible (breaking ties arbitrarily). After S is selected, line 5 removes its elements from U, and line 6 places S into $\mathcal{C}$. When the algorithm terminates, the set $\mathcal{C}$ contains a subfamily of $\mathcal{F}$ that covers X.

We can easily implement GREEDY-SET-COVER to run in time polynomial in $|X|$ and $|\mathcal{F}|$. Since the number of iterations of the loop on lines 3–6 is bounded from above by $\min(|X|, |\mathcal{F}|)$, and we can implement the loop body to run in time $O(|X|\,|\mathcal{F}|)$, a simple implementation runs in time $O(|X|\,|\mathcal{F}|\min(|X|, |\mathcal{F}|))$. Exercise 35.3-3 asks for a linear-time algorithm.

Analysis

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover. For convenience, in this chapter we denote the dth harmonic number $H_d = \sum_{i=1}^{d} 1/i$ (see Section A.1) by $H(d)$. As a boundary condition, we define $H(0) = 0$.

Theorem 35.4
GREEDY-SET-COVER is a polynomial-time $\rho(n)$-approximation algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}) .$$

Proof We have already shown that GREEDY-SET-COVER runs in polynomial time.

To show that GREEDY-SET-COVER is a $\rho(n)$-approximation algorithm, we assign a cost of 1 to each set selected by the algorithm, distribute this cost over the elements covered for the first time, and then use these costs to derive the desired relationship between the size of an optimal set cover $\mathcal{C}^*$ and the size of the set cover $\mathcal{C}$ returned by the algorithm. Let S_i denote the ith subset selected by GREEDY-SET-COVER; the algorithm incurs a cost of 1 when it adds S_i to $\mathcal{C}$. We spread this cost of selecting S_i evenly among the elements covered for the first time by S_i. Let c_x denote the cost allocated to element x, for each $x \in X$. Each element is assigned a cost only once, when it is covered for the first time. If x is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Each step of the algorithm assigns 1 unit of cost, and so

$$|\mathcal{C}| = \sum_{x \in X} c_x . \tag{35.9}$$

Each element $x \in X$ is in at least one set in the optimal cover $\mathcal{C}^*$, and so we have

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x . \tag{35.10}$$

Combining equation (35.9) and inequality (35.10), we have that

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x . \tag{35.11}$$

The remainder of the proof rests on the following key inequality, which we shall prove shortly. For any set S belonging to the family $\mathcal{F}$,

$$\sum_{x \in S} c_x \leq H(|S|) . \tag{35.12}$$

From inequalities (35.11) and (35.12), it follows that

$$\begin{aligned} |\mathcal{C}| &\leq \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\leq |\mathcal{C}^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

thus proving the theorem.

All that remains is to prove inequality (35.12). Consider any set $S \in \mathcal{F}$ and any $i = 1, 2, \ldots, |\mathcal{C}|$, and let

$$u_i = |S - (S_1 \cup S_2 \cup \cdots \cup S_i)|$$

be the number of elements in S that remain uncovered after the algorithm has selected sets $S_1, S_2, \ldots, S_i$. We define $u_0 = |S|$ to be the number of elements

of S, which are all initially uncovered. Let k be the least index such that $u_k = 0$, so that every element in S is covered by at least one of the sets $S_1, S_2, \ldots, S_k$ and some element in S is uncovered by $S_1 \cup S_2 \cup \cdots \cup S_{k-1}$. Then, $u_{i-1} \geq u_i$, and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i, for $i = 1, 2, \ldots, k$. Thus,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Observe that

$$\begin{aligned} |S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| &\geq |S - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \\ &= u_{i-1} , \end{aligned}$$

because the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does (otherwise, the algorithm would have chosen S instead of S_i). Consequently, we obtain

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} .$$

We now bound this quantity as follows:

$$\begin{aligned} \sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\ &= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\ &\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} && \text{(because } j \leq u_{i-1}\text{)} \\ &= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\ &= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\ &= H(u_0) - H(u_k) && \text{(because the sum telescopes)} \\ &= H(u_0) - H(0) \\ &= H(u_0) && \text{(because } H(0) = 0\text{)} \\ &= H(|S|) , \end{aligned}$$

which completes the proof of inequality (35.12). ∎

Corollary 35.5
GREEDY-SET-COVER is a polynomial-time $(\ln|X|+1)$-approximation algorithm.

Proof Use inequality (A.14) and Theorem 35.4. ■

In some applications, $\max\{|S| : S \in \mathcal{F}\}$ is a small constant, and so the solution returned by GREEDY-SET-COVER is at most a small constant times larger than optimal. One such application occurs when this heuristic finds an approximate vertex cover for a graph whose vertices have degree at most 3. In this case, the solution found by GREEDY-SET-COVER is not more than $H(3) = 11/6$ times as large as an optimal solution, a performance guarantee that is slightly better than that of APPROX-VERTEX-COVER.

Exercises

35.3-1
Consider each of the following words as a set of letters: {`arid`, `dash`, `drain`, `heard`, `lost`, `nose`, `shun`, `slate`, `snare`, `thread`}. Show which set cover GREEDY-SET-COVER produces when we break ties in favor of the word that appears first in the dictionary.

35.3-2
Show that the decision version of the set-covering problem is NP-complete by reducing it from the vertex-cover problem.

35.3-3
Show how to implement GREEDY-SET-COVER in such a way that it runs in time $O\left(\sum_{S\in\mathcal{F}}|S|\right)$.

35.3-4
Show that the following weaker form of Theorem 35.4 is trivially true:

$$|\mathcal{C}| \le |\mathcal{C}^*| \max\{|S| : S \in \mathcal{F}\} .$$

35.3-5
GREEDY-SET-COVER can return a number of different solutions, depending on how we break ties in line 4. Give a procedure BAD-SET-COVER-INSTANCE(n) that returns an n-element instance of the set-covering problem for which, depending on how we break ties in line 4, GREEDY-SET-COVER can return a number of different solutions that is exponential in n.

35.4 Randomization and linear programming

In this section, we study two useful techniques for designing approximation algorithms: randomization and linear programming. We shall give a simple randomized algorithm for an optimization version of 3-CNF satisfiability, and then we shall use linear programming to help design an approximation algorithm for a weighted version of the vertex-cover problem. This section only scratches the surface of these two powerful techniques. The chapter notes give references for further study of these areas.

A randomized approximation algorithm for MAX-3-CNF satisfiability

Just as some randomized algorithms compute exact solutions, some randomized algorithms compute approximate solutions. We say that a randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) \; . \tag{35.13}$$

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized $\rho(n)$-approximation algorithm.*** In other words, a randomized approximation algorithm is like a deterministic approximation algorithm, except that the approximation ratio is for an expected cost.

A particular instance of 3-CNF satisfiability, as defined in Section 34.4, may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, we may want to compute how "close" to satisfiable it is, that is, we may wish to find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***. The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1. We now show that randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ yields a randomized $8/7$-approximation algorithm. According to the definition of 3-CNF satisfiability from Section 34.4, we require each clause to consist of exactly three distinct literals. We further assume that no clause contains both a variable and its negation. (Exercise 35.4-1 asks you to remove this last assumption.)

Theorem 35.6
Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ is a randomized $8/7$-approximation algorithm.

Proof Suppose that we have independently set each variable to 1 with probability $1/2$ and to 0 with probability $1/2$. For $i = 1, 2, \ldots, m$, we define the indicator random variable

$$Y_i = \mathrm{I}\{\text{clause } i \text{ is satisfied}\} ,$$

so that $Y_i = 1$ as long as we have set at least one of the literals in the ith clause to 1. Since no literal appears more than once in the same clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $\Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, we have $\Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and by Lemma 5.1, we have $\mathrm{E}[Y_i] = 7/8$. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$. Then, we have

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8 .
\end{aligned}$$

Clearly, m is an upper bound on the number of satisfied clauses, and hence the approximation ratio is at most $m/(7m/8) = 8/7$. ■

Approximating weighted vertex cover using linear programming

In the ***minimum-weight vertex-cover problem***, we are given an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. For any vertex cover $V' \subseteq V$, we define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight.

We cannot apply the algorithm used for unweighted vertex cover, nor can we use a random solution; both methods may return solutions that are far from optimal. We shall, however, compute a lower bound on the weight of the minimum-weight

vertex cover, by using a linear program. We shall then "round" this solution and use it to obtain a vertex cover.

Suppose that we associate a variable $x(v)$ with each vertex $v \in V$, and let us require that $x(v)$ equals either 0 or 1 for each $v \in V$. We put v into the vertex cover if and only if $x(v) = 1$. Then, we can write the constraint that for any edge (u, v), at least one of u and v must be in the vertex cover as $x(u) + x(v) \geq 1$. This view gives rise to the following ***0-1 integer program*** for finding a minimum-weight vertex cover:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.14}$$

subject to

$$x(u) + x(v) \geq 1 \qquad \text{for each } (u, v) \in E \tag{35.15}$$

$$x(v) \in \{0, 1\} \qquad \text{for each } v \in V\ . \tag{35.16}$$

In the special case in which all the weights $w(v)$ are equal to 1, this formulation is the optimization version of the NP-hard vertex-cover problem. Suppose, however, that we remove the constraint that $x(v) \in \{0, 1\}$ and replace it by $0 \leq x(v) \leq 1$. We then obtain the following linear program, which is known as the ***linear-programming relaxation***:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.17}$$

subject to

$$x(u) + x(v) \geq 1 \qquad \text{for each } (u, v) \in E \tag{35.18}$$

$$x(v) \leq 1 \qquad \text{for each } v \in V \tag{35.19}$$

$$x(v) \geq 0 \qquad \text{for each } v \in V\ . \tag{35.20}$$

Any feasible solution to the 0-1 integer program in lines (35.14)–(35.16) is also a feasible solution to the linear program in lines (35.17)–(35.20). Therefore, the value of an optimal solution to the linear program gives a lower bound on the value of an optimal solution to the 0-1 integer program, and hence a lower bound on the optimal weight in the minimum-weight vertex-cover problem.

The following procedure uses the solution to the linear-programming relaxation to construct an approximate solution to the minimum-weight vertex-cover problem:

APPROX-MIN-WEIGHT-VC(G, w)
1 $C = \emptyset$
2 compute $\bar{x}$, an optimal solution to the linear program in lines (35.17)–(35.20)
3 **for** each $v \in V$
4 **if** $\bar{x}(v) \geq 1/2$
5 $C = C \cup \{v\}$
6 **return** C

The APPROX-MIN-WEIGHT-VC procedure works as follows. Line 1 initializes the vertex cover to be empty. Line 2 formulates the linear program in lines (35.17)–(35.20) and then solves this linear program. An optimal solution gives each vertex v an associated value $\bar{x}(v)$, where $0 \leq \bar{x}(v) \leq 1$. We use this value to guide the choice of which vertices to add to the vertex cover C in lines 3–5. If $\bar{x}(v) \geq 1/2$, we add v to C; otherwise we do not. In effect, we are "rounding" each fractional variable in the solution to the linear program to 0 or 1 in order to obtain a solution to the 0-1 integer program in lines (35.14)–(35.16). Finally, line 6 returns the vertex cover C.

Theorem 35.7
Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the linear program in line 2, and because the **for** loop of lines 3–5 runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

Now we show that APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. Let C^* be an optimal solution to the minimum-weight vertex-cover problem, and let z^* be the value of an optimal solution to the linear program in lines (35.17)–(35.20). Since an optimal vertex cover is a feasible solution to the linear program, z^* must be a lower bound on $w(C^*)$, that is,

$$z^* \leq w(C^*) . \tag{35.21}$$

Next, we claim that by rounding the fractional values of the variables $\bar{x}(v)$, we produce a set C that is a vertex cover and satisfies $w(C) \leq 2z^*$. To see that C is a vertex cover, consider any edge $(u, v) \in E$. By constraint (35.18), we know that $x(u) + x(v) \geq 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of u and v is included in the vertex cover, and so every edge is covered.

Now, we consider the weight of the cover. We have

$$
\begin{aligned}
z^* &= \sum_{v \in V} w(v)\, \bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\, \bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C) \,. \qquad (35.22)
\end{aligned}
$$

Combining inequalities (35.21) and (35.22) gives

$$w(C) \leq 2z^* \leq 2w(C^*) \,,$$

and hence APPROX-MIN-WEIGHT-VC is a 2-approximation algorithm. ■

Exercises

35.4-1

Show that even if we allow a clause to contain both a variable and its negation, randomly setting each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ still yields a randomized $8/7$-approximation algorithm.

35.4-2

The ***MAX-CNF satisfiability problem*** is like the MAX-3-CNF satisfiability problem, except that it does not restrict each clause to have exactly 3 literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

35.4-3

In the MAX-CUT problem, we are given an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 23 and the ***weight*** of a cut as the number of edges crossing the cut. The goal is to find a cut of maximum weight. Suppose that for each vertex v, we randomly and independently place v in S with probability $1/2$ and in $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

35.4-4

Show that the constraints in line (35.19) are redundant in the sense that if we remove them from the linear program in lines (35.17)–(35.20), any optimal solution to the resulting linear program must satisfy $x(v) \leq 1$ for each $v \in V$.

35.5 The subset-sum problem

Recall from Section 34.5.5 that an instance of the subset-sum problem is a pair (S, t), where S is a set $\{x_1, x_2, \ldots, x_n\}$ of positive integers and t is a positive integer. This decision problem asks whether there exists a subset of S that adds up exactly to the target value t. As we saw in Section 34.5.5, this problem is NP-complete.

The optimization problem associated with this decision problem arises in practical applications. In the optimization problem, we wish to find a subset of $\{x_1, x_2, \ldots, x_n\}$ whose sum is as large as possible but not larger than t. For example, we may have a truck that can carry no more than t pounds, and n different boxes to ship, the ith of which weighs x_i pounds. We wish to fill the truck with as heavy a load as possible without exceeding the given weight limit.

In this section, we present an exponential-time algorithm that computes the optimal value for this optimization problem, and then we show how to modify the algorithm so that it becomes a fully polynomial-time approximation scheme. (Recall that a fully polynomial-time approximation scheme has a running time that is polynomial in $1/\epsilon$ as well as in the size of the input.)

An exponential-time exact algorithm

Suppose that we computed, for each subset S' of S, the sum of the elements in S', and then we selected, among the subsets whose sum does not exceed t, the one whose sum was closest to t. Clearly this algorithm would return the optimal solution, but it could take exponential time. To implement this algorithm, we could use an iterative procedure that, in iteration i, computes the sums of all subsets of $\{x_1, x_2, \ldots, x_i\}$, using as a starting point the sums of all subsets of $\{x_1, x_2, \ldots, x_{i-1}\}$. In doing so, we would realize that once a particular subset S' had a sum exceeding t, there would be no reason to maintain it, since no superset of S' could be the optimal solution. We now give an implementation of this strategy.

The procedure EXACT-SUBSET-SUM takes an input set $S = \{x_1, x_2, \ldots, x_n\}$ and a target value t; we'll see its pseudocode in a moment. This procedure it-

eratively computes L_i, the list of sums of all subsets of $\{x_1, \ldots, x_i\}$ that do not exceed t, and then it returns the maximum value in L_n.

If L is a list of positive integers and x is another positive integer, then we let $L + x$ denote the list of integers derived from L by increasing each element of L by x. For example, if $L = \langle 1, 2, 3, 5, 9 \rangle$, then $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. We also use this notation for sets, so that

$$S + x = \{s + x : s \in S\} \ .$$

We also use an auxiliary procedure MERGE-LISTS(L, L'), which returns the sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed. Like the MERGE procedure we used in merge sort (Section 2.3.1), MERGE-LISTS runs in time $O(|L| + |L'|)$. We omit the pseudocode for MERGE-LISTS.

EXACT-SUBSET-SUM(S, t)

```
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4       L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5       remove from L_i every element that is greater than t
6  return the largest element in L_n
```

To see how EXACT-SUBSET-SUM works, let P_i denote the set of all values obtained by selecting a (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members. For example, if $S = \{1, 4, 5\}$, then

$$\begin{aligned} P_1 &= \{0, 1\} \ , \\ P_2 &= \{0, 1, 4, 5\} \ , \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} \ . \end{aligned}$$

Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) \ , \tag{35.23}$$

we can prove by induction on i (see Exercise 35.5-1) that the list L_i is a sorted list containing every element of P_i whose value is not more than t. Since the length of L_i can be as much as 2^i, EXACT-SUBSET-SUM is an exponential-time algorithm in general, although it is a polynomial-time algorithm in the special cases in which t is polynomial in $|S|$ or all the numbers in S are bounded by a polynomial in $|S|$.

A fully polynomial-time approximation scheme

We can derive a fully polynomial-time approximation scheme for the subset-sum problem by "trimming" each list L_i after it is created. The idea behind trimming is

that if two values in L are close to each other, then since we want just an approximate solution, we do not need to maintain both of them explicitly. More precisely, we use a trimming parameter δ such that $0 < \delta < 1$. When we ***trim*** a list L by δ, we remove as many elements from L as possible, in such a way that if L' is the result of trimming L, then for every element y that was removed from L, there is an element z still in L' that approximates y, that is,

$$\frac{y}{1+\delta} \le z \le y \ . \tag{35.24}$$

We can think of such a z as "representing" y in the new list L'. Each removed element y is represented by a remaining element z satisfying inequality (35.24). For example, if $\delta = 0.1$ and

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle \ ,$$

then we can trim L to obtain

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle \ ,$$

where the deleted value 11 is represented by 10, the deleted values 21 and 22 are represented by 20, and the deleted value 24 is represented by 23. Because every element of the trimmed version of the list is also an element of the original version of the list, trimming can dramatically decrease the number of elements kept while keeping a close (and slightly smaller) representative value in the list for each deleted element.

The following procedure trims list $L = \langle y_1, y_2, \ldots, y_m \rangle$ in time $\Theta(m)$, given L and δ, and assuming that L is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list.

```
TRIM(L, δ)
1  let m be the length of L
2  L' = ⟨y_1⟩
3  last = y_1
4  for i = 2 to m
5      if y_i > last · (1 + δ)        // y_i ≥ last because L is sorted
6          append y_i onto the end of L'
7          last = y_i
8  return L'
```

The procedure scans the elements of L in monotonically increasing order. A number is appended onto the returned list L' only if it is the first element of L or if it cannot be represented by the most recent number placed into L'.

Given the procedure TRIM, we can construct our approximation scheme as follows. This procedure takes as input a set $S = \{x_1, x_2, \ldots, x_n\}$ of n integers (in arbitrary order), a target integer t, and an "approximation parameter" ϵ, where

$$0 < \epsilon < 1 \,. \tag{35.25}$$

It returns a value z whose value is within a $1 + \epsilon$ factor of the optimal solution.

```
APPROX-SUBSET-SUM(S, t, ε)
1  n = |S|
2  L_0 = ⟨0⟩
3  for i = 1 to n
4       L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
5       L_i = TRIM(L_i, ε/2n)
6       remove from L_i every element that is greater than t
7  let z* be the largest value in L_n
8  return z*
```

Line 2 initializes the list L_0 to be the list containing just the element 0. The **for** loop in lines 3–6 computes L_i as a sorted list containing a suitably trimmed version of the set P_i, with all elements larger than t removed. Since we create L_i from L_{i-1}, we must ensure that the repeated trimming doesn't introduce too much compounded inaccuracy. In a moment, we shall see that APPROX-SUBSET-SUM returns a correct approximation if one exists.

As an example, suppose we have the instance

$$S = \langle 104, 102, 201, 101 \rangle$$

with $t = 308$ and $\epsilon = 0.40$. The trimming parameter δ is $\epsilon/8 = 0.05$. APPROX-SUBSET-SUM computes the following values on the indicated lines:

line 2: $L_0 = \langle 0 \rangle$,

line 4: $L_1 = \langle 0, 104 \rangle$,
line 5: $L_1 = \langle 0, 104 \rangle$,
line 6: $L_1 = \langle 0, 104 \rangle$,

line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$,
line 5: $L_2 = \langle 0, 102, 206 \rangle$,
line 6: $L_2 = \langle 0, 102, 206 \rangle$,

line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
line 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
line 6: $L_3 = \langle 0, 102, 201, 303 \rangle$,

line 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
line 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
line 6: $L_4 = \langle 0, 101, 201, 302 \rangle$.

The algorithm returns $z^* = 302$ as its answer, which is well within $\epsilon = 40\%$ of the optimal answer $307 = 104 + 102 + 101$; in fact, it is within 2%.

Theorem 35.8
APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof The operations of trimming L_i in line 5 and removing from L_i every element that is greater than t maintain the property that every element of L_i is also a member of P_i. Therefore, the value z^* returned in line 8 is indeed the sum of some subset of S. Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem. Then, from line 6, we know that $z^* \le y^*$. By inequality (35.1), we need to show that $y^*/z^* \le 1 + \epsilon$. We must also show that the running time of this algorithm is polynomial in both $1/\epsilon$ and the size of the input.

As Exercise 35.5-2 asks you to show, for every element y in P_i that is at most t, there exists an element $z \in L_i$ such that

$$\frac{y}{(1 + \epsilon/2n)^i} \le z \le y \; . \tag{35.26}$$

Inequality (35.26) must hold for $y^* \in P_n$, and therefore there exists an element $z \in L_n$ such that

$$\frac{y^*}{(1 + \epsilon/2n)^n} \le z \le y^* \; ,$$

and thus

$$\frac{y^*}{z} \le \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.27}$$

Since there exists an element $z \in L_n$ fulfilling inequality (35.27), the inequality must hold for z^*, which is the largest value in L_n; that is,

$$\frac{y^*}{z^*} \le \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.28}$$

Now, we show that $y^*/z^* \le 1 + \epsilon$. We do so by showing that $(1 + \epsilon/2n)^n \le 1 + \epsilon$. By equation (3.14), we have $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$. Exercise 35.5-3 asks you to show that

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 \; . \tag{35.29}$$

Therefore, the function $(1 + \epsilon/2n)^n$ increases with n as it approaches its limit of $e^{\epsilon/2}$, and we have

$$\begin{aligned}\left(1+\frac{\epsilon}{2n}\right)^n &\le e^{\epsilon/2} \\ &\le 1+\epsilon/2+(\epsilon/2)^2 \quad \text{(by inequality (3.13))} \\ &\le 1+\epsilon \quad \text{(by inequality (35.25))} \ . \end{aligned} \tag{35.30}$$

Combining inequalities (35.28) and (35.30) completes the analysis of the approximation ratio.

To show that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme, we derive a bound on the length of L_i. After trimming, successive elements z and z' of L_i must have the relationship $z'/z > 1+\epsilon/2n$. That is, they must differ by a factor of at least $1+\epsilon/2n$. Each list, therefore, contains the value 0, possibly the value 1, and up to $\lfloor \log_{1+\epsilon/2n} t \rfloor$ additional values. The number of elements in each list L_i is at most

$$\begin{aligned}\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\ &\le \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 \quad \text{(by inequality (3.17))} \\ &< \frac{3n \ln t}{\epsilon} + 2 \quad \text{(by inequality (35.25))} \ . \end{aligned}$$

This bound is polynomial in the size of the input—which is the number of bits $\lg t$ needed to represent t plus the number of bits needed to represent the set S, which is in turn polynomial in n—and in $1/\epsilon$. Since the running time of APPROX-SUBSET-SUM is polynomial in the lengths of the L_i, we conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme. ■

Exercises

35.5-1

Prove equation (35.23). Then show that after executing line 5 of EXACT-SUBSET-SUM, L_i is a sorted list containing every element of P_i whose value is not more than t.

35.5-2

Using induction on i, prove inequality (35.26).

35.5-3

Prove inequality (35.29).

35.5-4

How would you modify the approximation scheme presented in this section to find a good approximation to the smallest value not less than t that is a sum of some subset of the given input list?

35.5-5

Modify the APPROX-SUBSET-SUM procedure to also return the subset of S that sums to the value z^*.

Problems

35-1 Bin packing

Suppose that we are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. We wish to pack all the objects into the minimum number of unit-size bins. Each bin can hold any subset of the objects whose total size does not exceed 1.

a. Prove that the problem of determining the minimum number of bins required is NP-hard. (*Hint:* Reduce from the subset-sum problem.)

The ***first-fit*** heuristic takes each object in turn and places it into the first bin that can accommodate it. Let $S = \sum_{i=1}^{n} s_i$.

b. Argue that the optimal number of bins required is at least $\lceil S \rceil$.

c. Argue that the first-fit heuristic leaves at most one bin less than half full.

d. Prove that the number of bins used by the first-fit heuristic is never more than $\lceil 2S \rceil$.

e. Prove an approximation ratio of 2 for the first-fit heuristic.

f. Give an efficient implementation of the first-fit heuristic, and analyze its running time.

35-2 Approximating the size of a maximum clique

Let $G = (V, E)$ be an undirected graph. For any $k \geq 1$, define $G^{(k)}$ to be the undirected graph $(V^{(k)}, E^{(k)})$, where $V^{(k)}$ is the set of all ordered k-tuples of vertices from V and $E^{(k)}$ is defined so that $(v_1, v_2, \ldots, v_k)$ is adjacent to $(w_1, w_2, \ldots, w_k)$ if and only if for $i = 1, 2, \ldots, k$, either vertex v_i is adjacent to w_i in G, or else $v_i = w_i$.

a. Prove that the size of the maximum clique in $G^{(k)}$ is equal to the kth power of the size of the maximum clique in G.

b. Argue that if there is an approximation algorithm that has a constant approximation ratio for finding a maximum-size clique, then there is a polynomial-time approximation scheme for the problem.

35-3 Weighted set-covering problem

Suppose that we generalize the set-covering problem so that each set S_i in the family $\mathcal{F}$ has an associated weight w_i and the weight of a cover $\mathcal{C}$ is $\sum_{S_i \in \mathcal{C}} w_i$. We wish to determine a minimum-weight cover. (Section 35.3 handles the case in which $w_i = 1$ for all i.)

Show how to generalize the greedy set-covering heuristic in a natural manner to provide an approximate solution for any instance of the weighted set-covering problem. Show that your heuristic has an approximation ratio of $H(d)$, where d is the maximum size of any set S_i.

35-4 Maximum matching

Recall that for an undirected graph G, a matching is a set of edges such that no two edges in the set are incident on the same vertex. In Section 26.3, we saw how to find a maximum matching in a bipartite graph. In this problem, we will look at matchings in undirected graphs in general (i.e., the graphs are not required to be bipartite).

a. A ***maximal matching*** is a matching that is not a proper subset of any other matching. Show that a maximal matching need not be a maximum matching by exhibiting an undirected graph G and a maximal matching M in G that is not a maximum matching. (*Hint:* You can find such a graph with only four vertices.)

b. Consider an undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.

In this problem, we shall concentrate on a polynomial-time approximation algorithm for maximum matching. Whereas the fastest known algorithm for maximum matching takes superlinear (but polynomial) time, the approximation algorithm here will run in linear time. You will show that the linear-time greedy algorithm for maximal matching in part (b) is a 2-approximation algorithm for maximum matching.

c. Show that the size of a maximum matching in G is a lower bound on the size of any vertex cover for G.

d. Consider a maximal matching M in $G = (V, E)$. Let

$$T = \{v \in V : \text{some edge in } M \text{ is incident on } v\} .$$

What can you say about the subgraph of G induced by the vertices of G that are not in T?

e. Conclude from part (d) that $2|M|$ is the size of a vertex cover for G.

f. Using parts (c) and (e), prove that the greedy algorithm in part (b) is a 2-approximation algorithm for maximum matching.

35-5 Parallel machine scheduling

In the ***parallel-machine-scheduling problem***, we are given n jobs, $J_1, J_2, \ldots, J_n$, where each job J_k has an associated nonnegative processing time of p_k. We are also given m identical machines, $M_1, M_2, \ldots, M_m$. Any job can run on any machine. A ***schedule*** specifies, for each job J_k, the machine on which it runs and the time period during which it runs. Each job J_k must run on some machine M_i for p_k consecutive time units, and during that time period no other job may run on M_i. Let C_k denote the ***completion time*** of job J_k, that is, the time at which job J_k completes processing. Given a schedule, we define $C_{\max} = \max_{1 \le j \le n} C_j$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, suppose that we have two machines M_1 and M_2 and that we have four jobs J_1, J_2, J_3, J_4, with $p_1 = 2$, $p_2 = 12$, $p_3 = 4$, and $p_4 = 5$. Then one possible schedule runs, on machine M_1, job J_1 followed by job J_2, and on machine M_2, it runs job J_4 followed by job J_3. For this schedule, $C_1 = 2$, $C_2 = 14$, $C_3 = 9$, $C_4 = 5$, and $C_{\max} = 14$. An optimal schedule runs J_2 on machine M_1, and it runs jobs J_1, J_3, and J_4 on machine M_2. For this schedule, $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, $C_4 = 11$, and $C_{\max} = 12$.

Given a parallel-machine-scheduling problem, we let $C^*_{\max}$ denote the makespan of an optimal schedule.

a. Show that the optimal makespan is at least as large as the greatest processing time, that is,

$$C^*_{\max} \ge \max_{1 \le k \le n} p_k .$$

b. Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{\max} \ge \frac{1}{m} \sum_{1 \le k \le n} p_k .$$

Suppose that we use the following greedy algorithm for parallel machine scheduling: whenever a machine is idle, schedule any job that has not yet been scheduled.

c. Write pseudocode to implement this greedy algorithm. What is the running time of your algorithm?

d. For the schedule returned by the greedy algorithm, show that

$$C_{\max} \le \frac{1}{m} \sum_{1 \le k \le n} p_k + \max_{1 \le k \le n} p_k \ .$$

Conclude that this algorithm is a polynomial-time 2-approximation algorithm.

35-6 Approximating a maximum spanning tree

Let $G = (V, E)$ be an undirected graph with distinct edge weights $w(u, v)$ on each edge $(u, v) \in E$. For each vertex $v \in V$, let $\max(v) = \max_{(u,v) \in E} \{w(u, v)\}$ be the maximum-weight edge incident on that vertex. Let $S_G = \{\max(v) : v \in V\}$ be the set of maximum-weight edges incident on each vertex, and let T_G be the maximum-weight spanning tree of G, that is, the spanning tree of maximum total weight. For any subset of edges $E' \subseteq E$, define $w(E') = \sum_{(u,v) \in E'} w(u, v)$.

a. Give an example of a graph with at least 4 vertices for which $S_G = T_G$.

b. Give an example of a graph with at least 4 vertices for which $S_G \neq T_G$.

c. Prove that $S_G \subseteq T_G$ for any graph G.

d. Prove that $w(T_G) \ge w(S_G)/2$ for any graph G.

e. Give an $O(V + E)$-time algorithm to compute a 2-approximation to the maximum spanning tree.

35-7 An approximation algorithm for the 0-1 knapsack problem

Recall the knapsack problem from Section 16.2. There are n items, where the ith item is worth v_i dollars and weighs w_i pounds. We are also given a knapsack that can hold at most W pounds. Here, we add the further assumptions that each weight w_i is at most W and that the items are indexed in monotonically decreasing order of their values: $v_1 \ge v_2 \ge \cdots \ge v_n$.

In the 0-1 knapsack problem, we wish to find a subset of the items whose total weight is at most W and whose total value is maximum. The fractional knapsack problem is like the 0-1 knapsack problem, except that we are allowed to take a fraction of each item, rather than being restricted to taking either all or none of

each item. If we take a fraction x_i of item i, where $0 \leq x_i \leq 1$, we contribute $x_i w_i$ to the weight of the knapsack and receive value $x_i v_i$. Our goal is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, we consider restricted instances of the 0-1 knapsack problem. Given an instance I of the knapsack problem, we form restricted instances I_j, for $j = 1, 2, \ldots, n$, by removing items $1, 2, \ldots, j-1$ and requiring the solution to include item j (all of item j in both the fractional and 0-1 knapsack problems). No items are removed in instance I_1. For instance I_j, let P_j denote an optimal solution to the 0-1 problem and Q_j denote an optimal solution to the fractional problem.

a. Argue that an optimal solution to instance I of the 0-1 knapsack problem is one of $\{P_1, P_2, \ldots, P_n\}$.

b. Prove that we can find an optimal solution Q_j to the fractional problem for instance I_j by including item j and then using the greedy algorithm in which at each step, we take as much as possible of the unchosen item in the set $\{j+1, j+2, \ldots, n\}$ with maximum value per pound v_i/w_i.

c. Prove that we can always construct an optimal solution Q_j to the fractional problem for instance I_j that includes at most one item fractionally. That is, for all items except possibly one, we either include all of the item or none of the item in the knapsack.

d. Given an optimal solution Q_j to the fractional problem for instance I_j, form solution R_j from Q_j by deleting any fractional items from Q_j. Let $v(S)$ denote the total value of items taken in a solution S. Prove that $v(R_j) \geq v(Q_j)/2 \geq v(P_j)/2$.

e. Give a polynomial-time algorithm that returns a maximum-value solution from the set $\{R_1, R_2, \ldots, R_n\}$, and prove that your algorithm is a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

Chapter notes

Although methods that do not necessarily compute exact solutions have been known for thousands of years (for example, methods to approximate the value of π), the notion of an approximation algorithm is much more recent. Hochbaum [172] credits Garey, Graham, and Ullman [128] and Johnson [190] with formalizing the concept of a polynomial-time approximation algorithm. The first such algorithm is often credited to Graham [149].

Since this early work, thousands of approximation algorithms have been designed for a wide range of problems, and there is a wealth of literature on this field. Recent texts by Ausiello et al. [26], Hochbaum [172], and Vazirani [345] deal exclusively with approximation algorithms, as do surveys by Shmoys [315] and Klein and Young [207]. Several other texts, such as Garey and Johnson [129] and Papadimitriou and Steiglitz [271], have significant coverage of approximation algorithms as well. Lawler, Lenstra, Rinnooy Kan, and Shmoys [225] provide an extensive treatment of approximation algorithms for the traveling-salesman problem.

Papadimitriou and Steiglitz attribute the algorithm APPROX-VERTEX-COVER to F. Gavril and M. Yannakakis. The vertex-cover problem has been studied extensively (Hochbaum [172] lists 16 different approximation algorithms for this problem), but all the approximation ratios are at least $2 - o(1)$.

The algorithm APPROX-TSP-TOUR appears in a paper by Rosenkrantz, Stearns, and Lewis [298]. Christofides improved on this algorithm and gave a $3/2$-approximation algorithm for the traveling-salesman problem with the triangle inequality. Arora [22] and Mitchell [257] have shown that if the points are in the euclidean plane, there is a polynomial-time approximation scheme. Theorem 35.3 is due to Sahni and Gonzalez [301].

The analysis of the greedy heuristic for the set-covering problem is modeled after the proof published by Chvátal [68] of a more general result; the basic result as presented here is due to Johnson [190] and Lovász [238].

The algorithm APPROX-SUBSET-SUM and its analysis are loosely modeled after related approximation algorithms for the knapsack and subset-sum problems by Ibarra and Kim [187].

Problem 35-7 is a combinatorial version of a more general result on approximating knapsack-type integer programs by Bienstock and McClosky [45].

The randomized algorithm for MAX-3-CNF satisfiability is implicit in the work of Johnson [190]. The weighted vertex-cover algorithm is by Hochbaum [171]. Section 35.4 only touches on the power of randomization and linear programming in the design of approximation algorithms. A combination of these two ideas yields a technique called "randomized rounding," which formulates a problem as an integer linear program, solves the linear-programming relaxation, and interprets the variables in the solution as probabilities. These probabilities then help guide the solution of the original problem. This technique was first used by Raghavan and Thompson [290], and it has had many subsequent uses. (See Motwani, Naor, and Raghavan [261] for a survey.) Several other notable recent ideas in the field of approximation algorithms include the primal-dual method (see Goemans and Williamson [135] for a survey), finding sparse cuts for use in divide-and-conquer algorithms [229], and the use of semidefinite programming [134].

As mentioned in the chapter notes for Chapter 34, recent results in probabilistically checkable proofs have led to lower bounds on the approximability of many problems, including several in this chapter. In addition to the references there, the chapter by Arora and Lund [23] contains a good description of the relationship between probabilistically checkable proofs and the hardness of approximating various problems.