Part 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 matrices 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 26 presents an algorithmic model for parallel computing based on task-parallel computing, and more specifically, fork-join parallelism. 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 fork-join algorithms, including algorithms for matrix multiplication and merge sorting.

An algorithm that receives its input over time, rather than having the entire input available at the start, is called an "online" algorithm. Chapter 27 examines techniques used in online algorithms, starting with the "toy" problem of how long to wait for an elevator before taking the stairs. It then studies the "move-to-front" heuristic for maintaining a linked list and finishes with the online version of the caching problem we saw back in Section 15.4. The analyses of these online algorithms are remarkable in that they prove that these algorithms, which do not know their future inputs, perform within a constant factor of optimal algorithms that know the future inputs.

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 how to model problems as linear programs, where the goal is to maximize or minimize an objective, given limited resources and competing constraints. Linear programming arises in a variety of practical application areas. The chapter also addresses the concept of "duality" which, by establishing that a maximization problem and minimization problem have the same objective value, helps to show that solutions to each are optimal.

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 derives a parallel circuit to compute the FFT.

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 finishes with the Miller-Rabin randomized primality test, which enables finding large primes efficiently—an essential requirement for the RSA system.

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. The chapter finishes by studying suffix arrays, which can not only find a pattern in a text string, but can do quite a bit more, such as finding the longest repeated substring in a text and finding the longest common substring appearing in two texts.

Chapter 33 examines three algorithms within the expansive field of machine learning. Machine-learning algorithms are designed to take in vast amounts of data, devise hypotheses about patterns in the data, and test these hypotheses. The chapter starts with k-means clustering, which groups data elements into k classes based on how similar they are to each other. It then shows how to use the technique of multiplicative weights to make predictions accurately based on a set of "experts" of varying quality. Perhaps surprisingly, even without knowing which experts are reliable and which are not, you can predict almost as accurately as the most reliable expert. The chapter finishes with gradient descent, an optimization technique that finds a local minimum value for a function. Gradient descent has many applications, including finding parameter settings for many machine-learning models.

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, using them to prove several classic problems NP-complete: determining whether a graph has a hamiltonian cycle (a cycle that includes every vertex), determining whether a boolean formula is satisfiable (whether there exists an assignment of boolean values to its variables that causes the formula to evaluate to TRUE), 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-salesperson problem (find a shortest route that starts and ends at the same location and visits each of a set of locations once) 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 investing increasing amounts of computation time yields 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-salesperson problem, the set-covering problem, and the subset-sum problem.

26 Parallel Algorithms

The vast majority of algorithms in this book are ***serial algorithms*** suitable for running on a uniprocessor computer that executes only one instruction at a time. This chapter extends our algorithmic model to encompass ***parallel algorithms***, where multiple instructions can execute simultaneously. Specifically, we'll explore the elegant model of task-parallel algorithms, which are amenable to algorithmic design and analysis. Our study focuses on fork-join parallel algorithms, the most basic and best understood kind of task-parallel algorithm. Fork-join parallel algorithms can be expressed cleanly using simple linguistic extensions to ordinary serial code. Moreover, they can be implemented efficiently in practice.

Parallel computers—computers with multiple processing units—are ubiquitous. Handheld, laptop, desktop, and cloud machines are all ***multicore computers***, or simply, ***multicores***, containing multiple processing "cores." Each processing core is a full-fledged processor that can directly access any location in a common ***shared memory***. Multicores can be aggregated into larger systems, such as clusters, by using a network to interconnect them. These multicore clusters usually have a ***distributed memory***, where one multicore's memory cannot be accessed directly by a processor in another multicore. Instead, the processor must explicitly send a message over the cluster network to a processor in the remote multicore to request any data it requires. The most powerful clusters are supercomputers, comprising many thousands of multicores. But since shared-memory programming tends to be conceptually easier than distributed-memory programming, and multicore machines are widely available, this chapter focuses on parallel algorithms for multicores.

One approach to programming multicores is ***thread parallelism***. This processor-centric parallel-programming model employs a software abstraction of "virtual processors," or ***threads*** that share a common memory. Each thread maintains its own program counter and can execute code independently of the other threads. The operating system loads a thread onto a processing core for execution and switches it out when another thread needs to run.

Unfortunately, programming a shared-memory parallel computer using threads tends to be difficult and error-prone. One reason is that it can be complicated to dynamically partition the work among the threads so that each thread receives approximately the same load. For any but the simplest of applications, the programmer must use complex communication protocols to implement a scheduler that load-balances the work.

Task-parallel programming

The difficulty of thread programming has led to the creation of ***task-parallel platforms***, which provide a layer of software on top of threads to coordinate, schedule, and manage the processors of a multicore. Some task-parallel platforms are built as runtime libraries, but others provide full-fledged parallel languages with compiler and runtime support.

Task-parallel programming allows parallelism to be specified in a "processor-oblivious" fashion, where the programmer identifies what computational tasks may run in parallel but does not indicate which thread or processor performs the task. Thus, the programmer is freed from worrying about communication protocols, load balancing, and other vagaries of thread programming. The task-parallel platform contains a scheduler, which automatically load-balances the tasks across the processors, thereby greatly simplifying the programmer's chore. ***Task-parallel algorithms*** provide a natural extension to ordinary serial algorithms, allowing performance to be reasoned about mathematically using "work/span analysis."

Fork-join parallelism

Although the functionality of task-parallel environments is still evolving and increasing, almost all support ***fork-join parallelism***, which is typically embodied in two linguistic features: ***spawning*** and ***parallel loops***. Spawning allows a subroutine to be "forked": executed like a subroutine call, except that the caller can continue to execute while the spawned subroutine computes its result. A parallel loop is like an ordinary **for** loop, except that multiple iterations of the loop can execute at the same time.

Fork-join parallel algorithms employ spawning and parallel loops to describe parallelism. A key aspect of this parallel model, inherited from the task-parallel model but different from the thread model, is that the programmer does not specify which tasks in a computation *must* run in parallel, only which tasks *may* run in parallel. The underlying runtime system uses threads to load-balance the tasks across the processors. This chapter investigates parallel algorithms described in the fork-join model, as well as how the underlying runtime system can schedule task-parallel computations (which include fork-join computations) efficiently.

Fork-join parallelism offers several important advantages:

- The fork-join programming model is a simple extension of the familiar serial programming model used in most of this book. To describe a fork-join parallel algorithm, the pseudocode in this book needs just three added keywords: **parallel**, **spawn**, and **sync**. Deleting these parallel keywords from the parallel pseudocode results in ordinary serial pseudocode for the same problem, which we call the "serial projection" of the parallel algorithm.
- The underlying task-parallel model provides a theoretically clean way to quantify parallelism based on the notions of "work" and "span."
- Spawning allows many divide-and-conquer algorithms to be parallelized naturally. Moreover, just as serial divide-and-conquer algorithms lend themselves to analysis using recurrences, so do parallel algorithms in the fork-join model.
- The fork-join programming model is faithful to how multicore programming has been evolving in practice. A growing number of multicore environments support one variant or another of fork-join parallel programming, including Cilk [290, 291, 383, 396], Habanero-Java [466], the Java Fork-Join Framework [279], OpenMP [81], Task Parallel Library [289], Threading Building Blocks [376], and X10 [82].

Section 26.1 introduces parallel pseudocode, shows how the execution of a task-parallel computation can be modeled as a directed acyclic graph, and presents the metrics of work, span, and parallelism, which you can use to analyze parallel algorithms. Section 26.2 investigates how to multiply matrices in parallel, and Section 26.3 tackles the tougher problem of designing an efficient parallel merge sort.

26.1 The basics of fork-join parallelism

Our exploration of parallel programming begins with the problem of computing Fibonacci numbers recursively in parallel. We'll look at a straightforward serial Fibonacci calculation, which, although inefficient, serves as a good illustration of how to express parallelism in pseudocode.

Recall that the Fibonacci numbers are defined by equation (3.31) on page 69:

$$F_i = \begin{cases} 0 & \text{if } i = 0\,, \\ 1 & \text{if } i = 1\,, \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2\,. \end{cases}$$

To calculate the nth Fibonacci number recursively, you could use the ordinary serial algorithm in the procedure FIB on the facing page. You would not really want to

compute large Fibonacci numbers this way, because this computation does needless repeated work, but parallelizing it can be instructive.

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

To analyze this algorithm, let $T(n)$ denote the running time of $\text{FIB}(n)$. Since $\text{FIB}(n)$ 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 establish by using the substitution method (see Section 4.3). To show that $T(n) = O(F_n)$, we'll adopt the inductive hypothesis that $T(n) \le aF_n - b$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

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

if we choose b large enough to dominate the upper-bound constant in the $\Theta(1)$ term. We can then choose a large enough to upper-bound the $\Theta(1)$ base case for small n. To show that $T(n) = \Omega(F_n)$, we use the inductive hypothesis $T(n) \ge aF_n - b$. Substituting and following reasoning similar to the asymptotic upper-bound argument, we establish this hypothesis by choosing b smaller than the lower-bound constant in the $\Theta(1)$ term and a small enough to lower-bound the $\Theta(1)$ base case for small n. Theorem 3.1 on page 56 then establishes that $T(n) = \Theta(F_n)$, as desired. Since $F_n = \Theta(\phi^n)$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio, by equation (3.34) on page 69, it follows that

$$T(n) = \Theta(\phi^n) . \tag{26.1}$$

Thus this procedure is a particularly slow way to compute Fibonacci numbers, since it runs in exponential time. (See Problem 31-3 on page 954 for faster ways.)

Let's see why the algorithm is inefficient. Figure 26.1 shows the tree of recursive procedure instances created when computing F_6 with the FIB procedure. The call to $\text{FIB}(6)$ recursively calls $\text{FIB}(5)$ and then $\text{FIB}(4)$. But, the call to $\text{FIB}(5)$ also

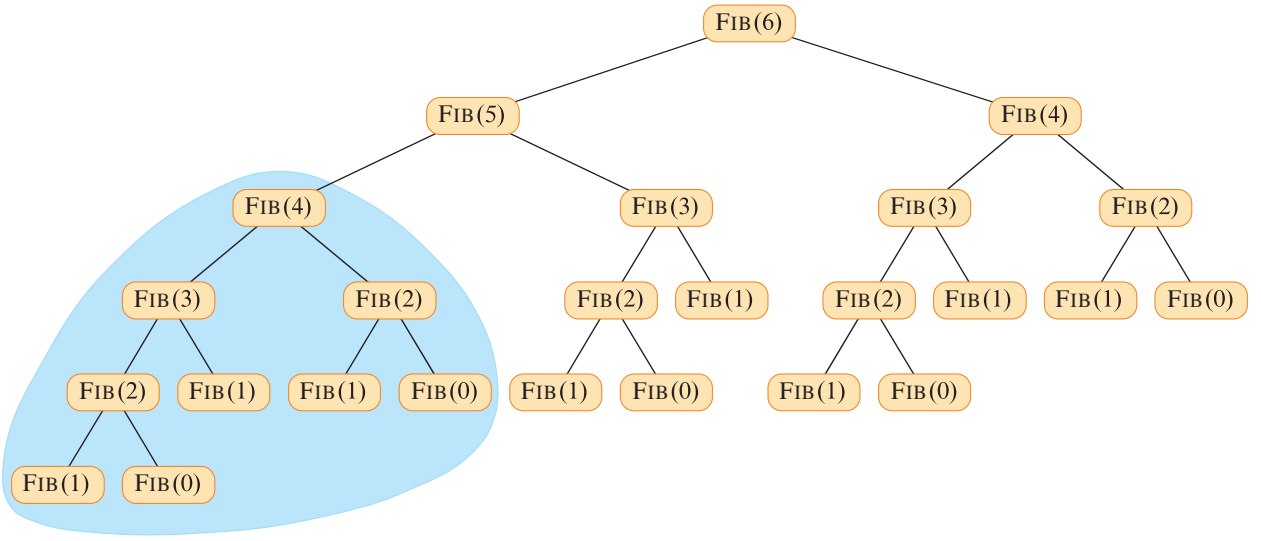


Figure 26.1 The invocation tree for FIB(6). Each node in the tree represents a procedure instance whose children are the procedure instances it calls during its execution. Since each instance of FIB with the same argument does the same work to produce the same result, the inefficiency of this algorithm for computing the Fibonacci numbers can be seen by the vast number of repeated calls to compute the same thing. The portion of the tree shaded blue appears in task-parallel form in Figure 26.2.

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 (recall the definition of "memoize" from page 368), the second call to FIB(4) replicates the work that the first call performs, which is wasteful.

Although the FIB procedure is a poor way to compute Fibonacci numbers, it can help us warm up to parallelism concepts. Perhaps the most basic concept is to understand is that if two parallel tasks operate on entirely different data, then—absent other interference—they each produce the same outcomes when executed at the same time as when they run serially one after the other. Within FIB(n), for example, the two recursive calls in line 3 to FIB($n-1$) and in line 4 to FIB($n-2$) can safely execute in parallel because the computation performed by one in no way affects the other.

Parallel keywords

The P-FIB procedure on the next page computes Fibonacci numbers, but using the ***parallel keywords*** **spawn** and **sync** to indicate parallelism in the pseudocode.

If the keywords **spawn** and **sync** are deleted from P-FIB, the resulting pseudocode text is identical to FIB (other than renaming the procedure in the header

```
P-FIB(n)
1  if n ≤ 1
2      return n
3  else x = spawn P-FIB(n − 1)      // don't wait for subroutine to return
4       y = P-FIB(n − 2)            // in parallel with spawned subroutine
5       sync                        // wait for spawned subroutine to finish
6       return x + y
```

and in the two recursive calls). We define the ***serial projection***[1] of a parallel algorithm to be the serial algorithm that results from ignoring the parallel directives, which in this case can be done by omitting the keywords **spawn** and **sync**. For **parallel for** loops, which we'll see later on, we omit the keyword **parallel**. Indeed, our parallel pseudocode possesses the elegant property that its serial projection is always ordinary serial pseudocode to solve the same problem.

Semantics of parallel keywords

Spawning occurs when the keyword **spawn** precedes a procedure call, as in line 3 of P-FIB. 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 finish, as would 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 in parallel with its spawned children, only that it *may*. The parallel 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 in parallel by assigning them to available pro-

---

[1] In mathematics, a projection is an idempotent function, that is, a function f such that $f \circ f = f$. In this case, the function f maps the set $\mathcal{P}$ of fork-join programs to the set $\mathcal{P}_S \subset \mathcal{P}$ of serial programs, which are themselves fork-join programs with no parallelism. For a fork-join program $x \in \mathcal{P}$, since we have $f(f(x)) = f(x)$, the serial projection, as we have defined it, is indeed a mathematical projection.

cessors as the computation unfolds. We'll discuss the theory behind task-parallel schedulers shortly (on page 759).

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 finish before proceeding to the statement after the **sync**—the "join" of a fork-join parallel computation. The P-FIB procedure requires a **sync** before the **return** statement in line 6 to avoid the anomaly that would occur if x and y were summed before P-FIB$(n-1)$ had finished and its return value had been assigned to x. In addition to explicit join synchronization provided by the **sync** statement, it is convenient to assume that every procedure executes a **sync** implicitly before it returns, thus ensuring that all children finish before their parent finishes.

A graph model for parallel execution

It helps to view the execution of a parallel computation—the dynamic stream of runtime instructions executed by processors under the direction of a parallel program—as a directed acyclic graph $G = (V, E)$, called a ***(parallel) trace***.[2] Conceptually, the vertices in V are executed instructions, and the edges in E represent dependencies between instructions, where $(u, v) \in E$ means that the parallel program required instruction u to execute before instruction v.

It's sometimes inconvenient, especially if we want to focus on the parallel structure of a computation, for a vertex of a trace to represent only one executed instruction. Consequently, if a chain of instructions contains no parallel or procedural control (no **spawn**, **sync**, procedure call, or **return**—via either an explicit **return** statement or the return that happens implicitly upon reaching the end of a procedure), we group the entire chain into a single ***strand***. As an example, Figure 26.2 shows the trace that results from computing P-FIB(4) in the portion of Figure 26.1 shaded blue. Strands do not include instructions that involve parallel or procedural control. These control dependencies must be represented as edges in the trace.

When a parent procedure calls a child, the trace contains an edge (u, v) from the strand u in the parent that executes the call to the first strand v of the spawned child, as illustrated in Figure 26.2 by the edge from the orange strand in P-FIB(4) to the blue strand in P-FIB(2). When the last strand v' in the child returns, the trace contains an edge (v', u') to the strand u', where u' is the successor strand of u in the parent, as with the edge from the white strand in P-FIB(2) to the white strand in P-FIB(4).

---

[2] Also called a ***computation dag*** in the literature.

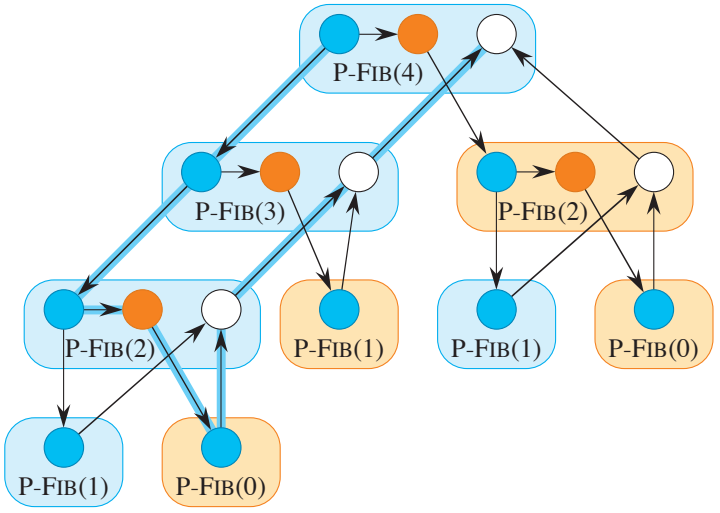


Figure 26.2 The trace of P-FIB(4) corresponding to the shaded portion of Figure 26.1. Each circle represents one strand, with blue circles representing any instructions executed in the part of the procedure (instance) up to the spawn of P-FIB($n-1$) in line 3; orange circles representing the instructions executed in 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 instructions executed in the part of the procedure after the **sync**, where it sums x and y, up to the point where it returns the result. Strands belonging to the same procedure are grouped into a rounded rectangle, blue for spawned procedures and tan for called procedures. Assuming that each strand takes unit time, the work is 17 time units, since there are 17 strands, and the span is 8 time units, since the critical path—shown with blue edges—contains 8 strands.

When the parent spawns a child, however, the trace is a little different. The edge (u, v) goes from parent to child as with a call, such as the edge from the blue strand in P-FIB(4) to the blue strand in P-FIB(3), but the trace contains another edge (u, u') as well, indicating that u's successor strand u' can continue to execute while v is executing. The edge from the blue strand in P-FIB(4) to the orange strand in P-FIB(4) illustrates one such edge. As with a call, there is an edge from the last strand v' in the child, but with a spawn, it no longer goes to u's successor. Instead, the edge is (v', x), where x is the strand immediately following the **sync** in the parent that ensures that the child has finished, as with the edge from the white strand in P-FIB(3) to the white strand in P-FIB(4).

You can figure out what parallel control created a particular trace. If a strand has two successors, one of them must have been spawned, and if a strand has multiple predecessors, 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 and procedural

control. If G contains a directed path from strand u to strand v, we say that the two strands are ***(logically) in series***. If there is no path in G either from u to v or from v to u, the strands are ***(logically) in parallel***.

A fork-join parallel trace can be pictured as a dag of strands embedded in an ***invocation tree*** of procedure instances. For example, Figure 26.1 shows the invocation tree for FIB(6), which also serves as the invocation tree for P-FIB(6), the edges between procedure instances now representing either calls or spawns. Figure 26.2 zooms in on the subtree that is shaded blue, showing the strands that constitute each procedure instance in P-FIB(4). All directed edges connecting strands run either within a procedure or along undirected edges of the invocation tree in Figure 26.1. (More general task-parallel traces that are not fork-join traces may contain some directed edges that do not run along the undirected tree edges.)

Our analyses generally assume that parallel algorithms execute on an ***ideal parallel computer***, which consists of a set of processors and a ***sequentially consistent*** shared memory. To understand sequential consistency, you first need to know that memory is accessed by ***load instructions***, which copy data from a location in the memory to a register within a processor, and by ***store instructions***, which copy data from a processor register to a location in the memory. A single line of pseudocode can entail several such instructions. For example, the line $x = y + z$ could result in load instructions to fetch each of y and z from memory into a processor, an instruction to add them together inside the processor, and a store instruction to place the result x back into memory. In a parallel computer, several processors might need to load or store at the same time. Sequential consistency means that even if multiple processors attempt to access the memory simultaneously, the shared memory behaves as if exactly one instruction from one of the processors is executed at a time, even though the actual transfer of data may happen at the same time. It is as if the instructions were executed one at a time sequentially according to some global linear order among all the processors that preserves the individual orders in which each processor executes its own instructions.

For task-parallel computations, which are scheduled onto processors automatically by a runtime system, the sequentially consistent shared memory behaves as if a parallel computation's executed instructions were executed one by one in the order of a topological sort (see Section 20.4) of its trace. That is, you can reason about the execution by imagining that the individual instructions (not generally the strands, which may aggregate many instructions) are interleaved in some linear order that preserves the partial order of the trace. Depending on scheduling, the linear order could vary from one run of the program to the next, but the behavior of any execution is always as if the instructions executed serially in a linear order consistent with the dependencies within the trace.

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'll define precisely a little later), the overhead of scheduling is generally minimal in practice.

Performance measures

We can gauge the theoretical efficiency of a task-parallel algorithm using ***work/span analysis***, which is based on two metrics: "work" and "span." The ***work*** of a task-parallel 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. If each strand takes unit time, the work is just the number of vertices in the trace. The ***span*** is the fastest possible time to execute the computation on an unlimited number of processors, which corresponds to the sum of the times taken by the strands along a longest path in the trace, where "longest" means that each strand is weighted by its execution time. Such a longest path is called the ***critical path*** of the trace, and thus the span is the weight of the longest (weighted) path in the trace. (Section 22.2, pages 617–619 shows how to find a critical path in a dag $G = (V, E)$ in $\Theta(V + E)$ time.) For a trace in which each strand takes unit time, the span equals the number of strands on the critical path. For example, the trace of Figure 26.2 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 task-parallel 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 task-parallel computation on P processors, we 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 task-parallel 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 $P T_P$ work. Since the total work to do is T_1, we have $P T_P \geq T_1$. Dividing by P yields the ***work law***:

$$T_P \geq T_1/P. \tag{26.2}$$

- 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:

$$T_P \geq T_\infty \,. \tag{26.3}$$

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 runs on P processors than on one 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 a P-processor ideal parallel computer 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***. ***Perfect linear speedup*** occurs when $T_1/T_P = P$.

The ratio T_1/T_∞ of the work to the span gives the ***parallelism*** of the parallel 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. 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 $T_1/T_P \leq T_1/T_\infty < P$. 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, if the number of processors exceeds the parallelism, adding even more processors makes the speedup less perfect.

As an example, consider the computation P-FIB(4) in Figure 26.2, 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 performance is impossible, no matter how many processors execute the computation. For larger input sizes, however, we'll see that P-FIB(n) exhibits substantial parallelism.

We define the ***(parallel) slackness*** of a task-parallel 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. Restating the bounds on speedup, if the slackness is less than 1, perfect linear speedup is impossible, because $T_1/(PT_\infty) < 1$ and the span law imply that $T_1/T_P \leq T_1/T_\infty < P$. Indeed, as the slackness decreases from 1 and approaches 0, the speedup of the computation diverges further and further from perfect linear speedup. If the slackness is less than 1, additional parallelism in an algorithm can have a great impact on its

execution efficiency. If the slackness is greater than 1, however, the work per processor is the limiting constraint. We'll see that as the slackness increases from 1, a good scheduler can achieve closer and closer to perfect linear speedup. But once the slackness is much greater than 1, the advantage of additional parallelism shows diminishing returns.

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 fork-join parallel-programming model provides no way for a programmer to specify which strands to execute on which processors. Instead, we rely on the runtime system'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 scheduler maps strands to processors directly.

A task-parallel scheduler must schedule the computation without knowing in advance when procedures will be spawned or when they will finish—that is, it must operate ***online***. Moreover, a good scheduler operates in a distributed fashion, where the threads implementing the scheduler cooperate to load-balance the computation. Provably good online, distributed schedulers exist, but analyzing them is complicated. Instead, to keep our analysis simple, we'll consider an online ***centralized*** scheduler that knows the global state of the computation at any moment.

In particular, we'll analyze ***greedy schedulers***, which assign as many strands to processors as possible in each time step, never leaving a processor idle if there is work that can be done. We'll classify each step of a greedy scheduler as follows:

- ***Complete step***: At least P strands are ***ready*** to execute, meaning that all strands on which they depend have finished execution. A greedy scheduler assigns any P of the ready strands to the processors, completely utilizing all the processor resources.
- ***Incomplete step***: Fewer than P strands are ready to execute. A greedy scheduler assigns each ready strand to its own processor, leaving some processors idle for the step, but executing all the ready strands.

The work law tells us that the fastest running time T_P that we can hope for on P processors must be at least T_1/P. The span law tells us that the fastest possible running time must be at least T_∞. 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 26.1
On an ideal parallel computer with P processors, a greedy scheduler executes a task-parallel computation with work T_1 and span T_∞ in time

$$T_P \leq T_1/P + T_\infty \,. \tag{26.4}$$

Proof Without loss of generality, assume that each strand takes unit time. (If necessary, replace each longer strand by a chain of unit-time strands.) We'll consider complete and incomplete steps separately.

In each complete step, the P processors together perform a total of P work. Thus, if the number of complete steps is k, the total work executing all the complete steps is kP. Since the greedy scheduler doesn't execute any strand more than once and only T_1 work needs to be performed, it follows that $kP \leq T_1$, from which we can conclude that the number k of complete steps is at most T_1/P.

Now, let's consider an incomplete step. Let G be the trace for the entire computation, let G' be the subtrace of G that has yet to be executed at the start of the incomplete step, and let G'' be the subtrace remaining to be executed after the incomplete step. Consider the set R of strands that are ready at the beginning of the incomplete step, where $|R| < P$. By definition, if a strand is ready, all its predecessors in trace G have executed. Thus the predecessors of strands in R do not belong to G'. A longest path in G' must necessarily start at a strand in R, since every other strand in G' has a predecessor and thus could not start a longest path. Because the greedy scheduler executes all ready strands during the incomplete step, the strands of G'' are exactly those in G' minus the strands in R. Consequently, the length of a longest path in G'' must be 1 less than the length of a longest path in G'. In other words, every incomplete step decreases the span of the trace remaining to be executed by 1. Hence, the number of incomplete steps can be at most T_∞.

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

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

Corollary 26.2
The running time T_P of any task-parallel computation scheduled by a greedy scheduler on a P-processor ideal parallel computer 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 (26.2) and (26.3)—give $T_P^* \geq \max\{T_1/P, T_\infty\}$, Theorem 26.1 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 task-parallel computation as the slackness grows.

Corollary 26.3
Let T_P be the running time of a task-parallel 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 $P \ll T_1/T_\infty$, or equivalently, the parallel slackness is much greater than 1, we have $T_P \approx T_1/P$, a speedup of approximately P.

Proof If we suppose that $P \ll T_1/T_\infty$, then it follows that $T_\infty \ll T_1/P$, and hence Theorem 26.1 gives $T_P \leq T_1/P + T_\infty \approx T_1/P$. Since the work law (26.2) dictates that $T_P \geq T_1/P$, we conclude that $T_P \approx T_1/P$, which is a speedup of $T_1/T_P \approx P$. ■

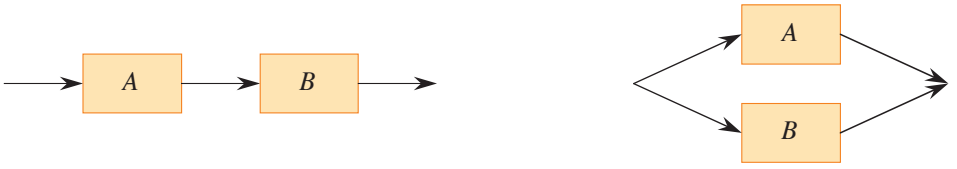
The $\ll$ symbol denotes "much less," but how much is "much less"? As a rule of thumb, a slackness of at least 10—that is, 10 times more parallelism than processors—generally suffices to achieve good speedup. Then, the span term in the greedy bound, inequality (26.4), is less than 10% of the work-per-processor 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 26-2 shows, sometimes reducing extreme parallelism yields algorithms that are better with respect to other concerns and which still scale up well on reasonable numbers of processors.

Analyzing parallel algorithms

We now have all the tools we need to analyze parallel algorithms using work/span analysis, allowing us to bound an algorithm's running time on any number 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 serial projection of the parallel algorithm. You should already be familiar with analyzing work, since that is what most of this textbook is about! Analyzing the span is the new thing that parallelism engenders, but it's generally no harder once you get the hang of it. Let's 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 serial projection of P-FIB is effectively the original FIB procedure, and hence, we have $T_1(n) = T(n) = \Theta(\phi^n)$ from equation (26.1).

Figure 26.3 illustrates how to analyze the span. If two traces are joined in series, their spans add to form the span of their composition, whereas if they are joined



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 26.3 Series-parallel composition of parallel traces. **(a)** When two traces 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 traces 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.

in parallel, the span of their composition is the maximum of the spans of the two traces. As it turns out, the trace of any fork-join parallel computation can be built up from single strands by series-parallel composition.

Armed with an understanding of series-parallel composition, we can analyze the span of 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 second equality above follows from the first because P-FIB$(n-1)$ uses P-FIB$(n-2)$ in its computation, so that the span of P-FIB$(n-1)$ must be at least as large as the span of P-FIB$(n-2)$.)

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, Corollary 26.3 tells us that 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 for which all the iterations can operate in parallel. Although the **spawn** and **sync** keywords can be used to parallelize such loops, it is more convenient to specify directly that the iterations of such loops can run in parallel. Our pseudocode provides this functionality via the **parallel** keyword, which precedes the **for** keyword in a **for** loop statement.

As an example, consider the problem of multiplying a square $n \times n$ 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 $i = 1, 2, \ldots, n$. The P-MAT-VEC procedure performs matrix-vector multiplication (actually, $y = y + Ax$) by computing all the entries of y in parallel. The **parallel for** keywords in line 1 of P-MAT-VEC indicate that the n iterations of the loop body, which includes a serial **for** loop, may be run in parallel. The initialization $y = 0$, if desired, should be performed before calling the procedure (and can be done with a **parallel for** loop).

```
P-MAT-VEC(A, x, y, n)
1  parallel for i = 1 to n          // parallel loop
2      for j = 1 to n               // serial loop
3          y_i = y_i + a_ij x_j
```

Compilers for fork-join parallel programs can implement **parallel for** loops in terms of **spawn** and **sync** by using recursive spawning. For example, for the **parallel for** loop in lines 1–3, a compiler can generate the auxiliary subroutine P-MAT-VEC-RECURSIVE and call P-MAT-VEC-RECURSIVE$(A, x, y, n, 1, n)$ in the place where the loop would be in the compiled code. As Figure 26.4 illustrates, this procedure recursively spawns the first half of the iterations of the loop to execute in parallel (line 5) with the second half of the iterations (line 6) and then executes a **sync** (line 7), thereby creating a binary tree of parallel execution. Each leaf represents a base case, which is the serial **for** loop of lines 2–3.

```
P-MAT-VEC-RECURSIVE(A, x, y, n, i, i')
1  if i == i'                       // just one iteration to do?
2      for j = 1 to n               // mimic P-MAT-VEC serial loop
3          y_i = y_i + a_ij x_j
4  else mid = ⌊(i + i')/2⌋          // parallel divide-and-conquer
5      spawn P-MAT-VEC-RECURSIVE(A, x, y, n, i, mid)
6      P-MAT-VEC-RECURSIVE(A, x, y, n, mid + 1, i')
7      sync
```

To calculate the work $T_1(n)$ of P-MAT-VEC on an $n \times n$ matrix, simply compute the running time of its serial projection, which comes from replacing the **parallel**

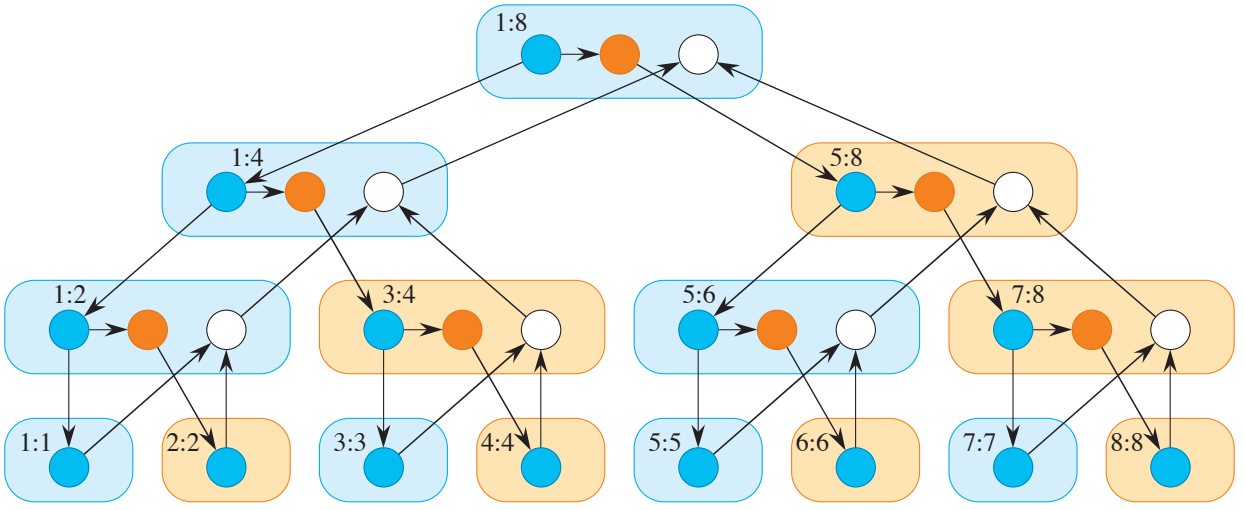


Figure 26.4 A trace for the computation of P-MAT-VEC-RECURSIVE$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn, in blue, or call, in tan) of the procedure. The blue circles represent strands corresponding to the part of the procedure up to the spawn of P-MAT-VEC-RECURSIVE in line 5. The orange circles represent strands corresponding to the part of the procedure that calls P-MAT-VEC-RECURSIVE in line 6 up to the **sync** in line 7, where it suspends until the spawned subroutine in line 5 returns. The white circles represent strands corresponding to the (negligible) part of the procedure after the **sync** up to the point where it returns.

for loop in line 1 with an ordinary **for** loop. The running time of the resulting serial pseudocode is $\Theta(n^2)$, which means that $T_1(n) = \Theta(n^2)$. This analysis seems to ignore the overhead for recursive spawning in implementing the parallel loops, however. Indeed, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serial projection, 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 one less than the number of leaves (see Exercise B.5-3 on page 1175). Each internal node performs constant work to divide the iteration range, and each leaf corresponds to a base case, which takes at least constant time ($\Theta(n)$ time in this case). Thus, by amortizing the overhead of recursive spawning over the work of the iterations in the leaves, we see that the overall work increases by at most a constant factor.

To reduce the overhead of recursive spawning, task-parallel platforms sometimes ***coarsen*** the leaves of the recursion by executing several iterations in a single leaf, either automatically or under programmer control. This optimization comes at the expense of reducing the parallelism. If the computation has sufficient parallel slackness, however, near-perfect linear speedup won't be sacrificed.

Although recursive spawning doesn't affect the work of a parallel loop asymptotically, we must take it into account when analyzing the span. Consider a parallel loop with n iterations in which the ith iteration has span $iter_\infty(i)$. Since the depth of recursion is logarithmic in the number of iterations, the parallel loop's span is

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

For example, let's compute the span of the doubly nested loops in lines 1–3 of P-MAT-VEC. The span for the **parallel for** loop control is $\Theta(\lg n)$. For each iteration of the outer parallel loop, the inner serial **for** loop contains n iterations of line 3. Since each iteration takes constant time, the total span for the inner serial **for** loop is $\Theta(n)$, no matter which iteration of the outer **parallel for** loop it's in. Thus, taking the maximum over all iterations of the outer loop and adding in the $\Theta(\lg n)$ for loop control yields an overall span of $T_\infty n = \Theta(n) + \Theta(\lg n) = \Theta(n)$ for the procedure. Since the work is $\Theta(n^2)$, the parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$. (Exercise 26.1-7 asks you to provide an implementation with even more parallelism.)

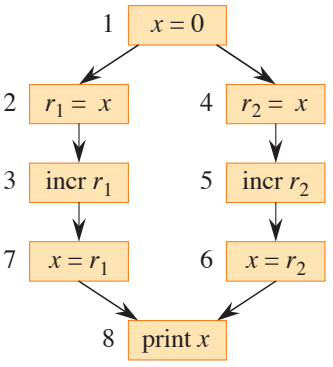
Race conditions

A parallel 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 when the input is the same. A parallel algorithm that is intended to be deterministic may nevertheless act nondeterministically, however, if it contains a difficult-to-diagnose bug called a "determinacy race."

Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured several others, and the Northeast Blackout of 2003, which left over 50 million people in the United States 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, sometimes with dire consequences.

A ***determinacy race*** occurs when two logically parallel instructions access the same memory location and at least one of the instructions modifies the value stored in the location. The toy procedure RACE-EXAMPLE on the following page illustrates a determinacy race. 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 a call of RACE-EXAMPLE should always print the value 2 (its serial projection 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:



(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 26.5 Illustration of the determinacy race in RACE-EXAMPLE. **(a)** A trace 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.

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

- Load x from memory into one of the processor's registers.
- Increment the value in the register.
- Store the value in the register back into x in memory.

Figure 26.5(a) illustrates a trace representing the execution of RACE-EXAMPLE, with the strands broken down to individual instructions. Recall that since an ideal parallel computer supports sequential consistency, you can view the parallel execution of a parallel algorithm as an interleaving of instructions that respects the dependencies in the trace. Part (b) of the figure shows the values in an execution of the computation that elicits the anomaly. The value x is kept 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 loads 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 loads 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 the value 2 that the serial projection would print.

Let's recap what happened. By sequential consistency, the effect of the parallel execution is as if the executed instructions of the two processors are interleaved. If processor 1 executes all its instructions before processor 2, a trivial interleaving, the value 2 is printed. Conversely, if processor 2 executes all its instructions before processor 1, the value 2 is still printed. When the instructions of the two processors interleave nontrivially, however, it is possible, as in this example execution, that one of the updates to x is lost, resulting in the value 1 being printed.

Of course, many executions do not elicit the bug. That's the problem with determinacy races. Generally, most instruction orderings produce correct results, such as any where the instructions on the left branch execute before the instructions on the right branch, or vice versa. But some orderings generate improper results when the instructions interleave. Consequently, races can be extremely hard to test for. Your program may fail, but you may be unable to reliably reproduce the failure in subsequent tests, confounding your attempts to locate the bug in your code and fix it. Task-parallel programming environments often provide race-detection productivity tools to help you isolate race bugs.

Many parallel programs in the real world are intentionally nondeterministic. They contain determinacy races, but they mitigate the dangers of nondeterminism through the use of mutual-exclusion locks and other methods of synchronization. For our purposes, however, we'll insist on an absence of determinacy races in the algorithms we develop. Nondeterministic programs are indeed interesting, but nondeterministic programming is a more advanced topic and unnecessary for a wide swath of interesting parallel algorithms.

To ensure that algorithms are deterministic, any two strands that operate in parallel should be ***mutually noninterfering***: they only read, and do not modify, any memory locations accessed by both of them. Consequently, in a **parallel for** construct, such as the outer loop of P-MAT-VEC, we want all the iterations of the body, including any code an iteration executes in subroutines, to be mutually noninterfering. And between a **spawn** and its corresponding **sync**, we want the code executed by the spawned child and the code executed by the parent to be mutually noninterfering, once again including invoked subroutines.

As an example of how easy it is to write code with unintentional races, the P-MAT-VEC-WRONG procedure on the next page is a faulty parallel implementation of matrix-vector multiplication that achieves a span of $\Theta(\lg n)$ by parallelizing the inner **for** loop. This procedure is incorrect, unfortunately, due to determinacy races when updating y_i in line 3, which executes in parallel for all n values of j.

Index variables of **parallel for** loops, such as i in line 1 and j in line 2, do not cause races between iterations. Conceptually, each iteration of the loop creates an independent variable to hold the index of that iteration during that iteration's

```
P-MAT-VEC-WRONG(A, x, y, n)
1  parallel for i = 1 to n
2      parallel for j = 1 to n
3          y_i = y_i + a_ij x_j        // determinacy race
```

execution of the loop body. Even if two parallel iterations both access the same index variable, they really are accessing different variable instances—hence different memory locations—and no race occurs.

A parallel algorithm with races can sometimes be deterministic. 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. For simplicity, however, we generally prefer code without determinacy races, even if the races are benign. And good parallel programmers frown on code with determinacy races that cause nondeterministic behavior, if deterministic code that performs comparably is an option.

But nondeterministic code does have its place. For example, you can't implement a parallel hash table, a highly practical data structure, without writing code containing determinacy races. Much research has centered around how to extend the fork-join model to incorporate limited "structured" nondeterminism while avoiding the full measure of complications that arise when nondeterminism is completely unrestricted.

A chess lesson

To illustrate the power of work/span analysis, this section closes with a true story that occurred during the development of one of the first world-class parallel chess-playing programs [106] many years ago. The timings below have been simplified for exposition.

The chess program was developed and tested on a 32-processor computer, but it was designed to run on a supercomputer with 512 processors. Since the supercomputer availability was limited and expensive, the developers ran benchmarks on the small computer and extrapolated performance to the large computer.

At one point, the developers incorporated an optimization into the program that reduced its running time on an important benchmark on the small 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 the 512 processors of the large machine. As a result, they abandoned the "optimization."

Here is their work/span analysis. The original version of the program had work $T_1 = 2048$ seconds and span $T_\infty = 1$ second. Let's treat inequality (26.4) on

page 760 as the equation $T_P = T_1/P + T_\infty$, which we can use as an approximation to the running time on P processors. Then indeed we have $T_{32} = 2048/32 + 1 = 65$. With the optimization, the work becomes $T_1' = 1024$ seconds, and the span becomes $T_\infty' = 8$ seconds. Our approximation gives $T_{32}' = 1024/32 + 8 = 40$.

The relative speeds of the two versions switch when we estimate their running times on 512 processors, however. The first version has a running time of $T_{512} = 2048/512+1 = 5$ seconds, and the second version runs in $T_{512}' = 1024/512+8 = 10$ seconds. The optimization that speeds up the program on 32 processors makes the program run for twice as long on 512 processors! The optimized version's span of 8, which is not the dominant term in the running time on 32 processors, becomes the dominant term on 512 processors, nullifying the advantage from using more processors. The optimization does not scale up.

The moral of the story is that work/span analysis, and measurements of work and span, can be superior to measured running times alone in extrapolating an algorithm's scalability.

Exercises

26.1-1

What does a trace for the execution of a serial algorithm look like?

26.1-2

Suppose that line 4 of P-FIB spawns P-FIB$(n-2)$, rather than calling it as is done in the pseudocode. How would the trace of P-FIB(4) in Figure 26.2 change? What is the impact on the asymptotic work, span, and parallelism?

26.1-3

Draw the trace 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 trace on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.

26.1-4

Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proved in Theorem 26.1:

$$T_P \le \frac{T_1 - T_\infty}{P} + T_\infty \,. \tag{26.5}$$

26.1-5

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

26.1-6

Professor Karan measures her deterministic task-parallel 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 (26.2), the span law (26.3), and inequality (26.5) from Exercise 26.1-4.)

26.1-7

Give a parallel 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.

26.1-8

Analyze the work, span, and parallelism of the procedure P-TRANSPOSE, which transposes an $n \times n$ matrix A in place.

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

26.1-9

Suppose that instead of a **parallel for** loop in line 2, the P-TRANSPOSE procedure in Exercise 26.1-8 had an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

26.1-10

For what number of processors do the two versions of the chess program run equally fast, assuming that $T_P = T_1/P + T_\infty$?

26.2 Parallel matrix multiplication

In this section, we'll explore how to parallelize the three matrix-multiplication algorithms from Sections 4.1 and 4.2. We'll see that each algorithm can be parallelized in a straightforward fashion using either parallel loops or recursive spawning. We'll analyze them using work/span analysis, and we'll see that each parallel algorithm attains the same performance on one processor as its corresponding serial algorithm, while scaling up to large numbers of processors.

A parallel algorithm for matrix multiplication using parallel loops

The first algorithm we'll study is P-MATRIX-MULTIPLY, which simply parallelizes the two outer loops in the procedure MATRIX-MULTIPLY on page 81.

```
P-MATRIX-MULTIPLY(A, B, C, n)
1  parallel for i = 1 to n                  // compute entries in each of n rows
2      parallel for j = 1 to n              // compute n entries in row i
3          for k = 1 to n
4              c_ij = c_ij + a_ik · b_kj    // add in another term of equation (4.1)
```

Let's analyze P-MATRIX-MULTIPLY. Since the serial projection of the algorithm is just MATRIX-MULTIPLY, the work is the same as the running time of MATRIX-MULTIPLY: $T_1(n) = \Theta(n^3)$. 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 1, then down the tree of recursion for the **parallel for** loop starting in line 2, and then executes all n iterations of the ordinary **for** loop starting in line 3, 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 26.2-3 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 parallel divide-and-conquer algorithm for matrix multiplication

Section 4.1 shows how to multiply $n \times n$ matrices serially in $\Theta(n^3)$ time using a divide-and-conquer strategy. Let's see how to parallelize that algorithm using recursive spawning instead of calls.

The serial MATRIX-MULTIPLY-RECURSIVE procedure on page 83 takes as input three $n \times n$ matrices A, B, and C and performs the matrix calculation $C = C + A \cdot B$ by recursively performing eight multiplications of $n/2 \times n/2$ submatrices of A and B. The P-MATRIX-MULTIPLY-RECURSIVE procedure on the following page implements the same divide-and-conquer strategy, but it uses spawning to perform the eight multiplications in parallel. To avoid determinacy races in updating the elements of C, it creates a temporary matrix D to store four of the submatrix products. At the end, it adds C and D together to produce the final result. (Problem 26-2 asks you to eliminate the temporary matrix D at the expense of some parallelism.)

Lines 2–3 of P-MATRIX-MULTIPLY-RECURSIVE handle the base case of multiplying 1×1 matrices. The remainder of the procedure deals with the recursive case. Line 4 allocates a temporary matrix D, and lines 5–7 zero it. Line 8 partitions each of the four matrices A, B, C, and D into $n/2 \times n/2$ submatrices. (As

```
P-MATRIX-MULTIPLY-RECURSIVE(A, B, C, n)
 1  if n == 1                          // just one element in each matrix?
 2      c_11 = c_11 + a_11 · b_11
 3      return
 4  let D be a new n × n matrix        // temporary matrix
 5  parallel for i = 1 to n            // set D = 0
 6      parallel for j = 1 to n
 7          d_ij = 0
 8  partition A, B, C, and D 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 D_11, D_12, D_21, D_22; respectively
 9  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_11, B_11, C_11, n/2)
10  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_11, B_12, C_12, n/2)
11  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_21, B_11, C_21, n/2)
12  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_21, B_12, C_22, n/2)
13  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_12, B_21, D_11, n/2)
14  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_12, B_22, D_12, n/2)
15  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_22, B_21, D_21, n/2)
16  spawn P-MATRIX-MULTIPLY-RECURSIVE(A_22, B_22, D_22, n/2)
17  sync                               // wait for spawned submatrix products
18  parallel for i = 1 to n            // update C = C + D
19      parallel for j = 1 to n
20          c_ij = c_ij + d_ij
```

with MATRIX-MULTIPLY-RECURSIVE on page 83, we're glossing over the subtle issue of how to use index calculations to represent submatrix sections of a matrix.) The spawned recursive call in line 9 sets $C_{11} = C_{11} + A_{11} \cdot B_{11}$, so that C_{11} accumulates the first of the two terms in equation (4.5) on page 82. Similarly, lines 10–12 cause each of C_{12}, C_{21}, and C_{22} in parallel to accumulate the first of the two terms in equations (4.6)–(4.8), respectively. Line 13 sets the submatrix D_{11} to the submatrix product $A_{12} \cdot B_{21}$, so that D_{11} equals the second of the two terms in equation (4.5). Lines 14–16 set each of D_{12}, D_{21}, and D_{22} in parallel to the second of the two terms in equations (4.6)–(4.8), respectively. The **sync** statement in line 17 ensures that all the spawned submatrix products in lines 9–16 have been computed, after which the doubly nested **parallel for** loops in lines 18–20 add the elements of D to the corresponding elements of C.

Let's analyze the P-MATRIX-MULTIPLY-RECURSIVE procedure. We start by analyzing the work $M_1(n)$, echoing the serial running-time analysis of its progenitor MATRIX-MULTIPLY-RECURSIVE. The recursive case allocates and zeros the

temporary matrix D in $\Theta(n^2)$ time, partitions in $\Theta(1)$ time, performs eight recursive multiplications of $n/2 \times n/2$ matrices, and finishes up with the $\Theta(n^2)$ work from adding two $n \times n$ matrices. Thus the work outside the spawned recursive calls is $\Theta(n^2)$, and the recurrence for the work $M_1(n)$ becomes

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

by case 1 of the master theorem (Theorem 4.1). Not surprisingly, the work of this parallel algorithm is asymptotically the same as the running time of the procedure MATRIX-MULTIPLY on page 81, with its triply nested loops.

Let's determine the span $M_\infty(n)$ of P-MATRIX-MULTIPLY-RECURSIVE. Because the eight parallel recursive spawns all execute on matrices of the same size, the maximum span for any recursive spawn is just the span of a single one of them, or $M_\infty(n/2)$. The span for the doubly nested **parallel for** loops in lines 5–7 is $\Theta(\lg n)$ because each loop control adds $\Theta(\lg n)$ to the constant span of line 7. Similarly, the doubly nested **parallel for** loops in lines 18–20 add another $\Theta(\lg n)$. Matrix partitioning by index calculation has $\Theta(1)$ span, which is dominated by the $\Theta(\lg n)$ span of the nested loops. We obtain the recurrence

$$M_\infty(n) = M_\infty(n/2) + \Theta(\lg n) \,. \tag{26.6}$$

Since this recurrence falls under case 2 of the master theorem with $k = 1$, the solution is $M_\infty(n) = \Theta(\lg^2 n)$.

The parallelism of P-MATRIX-MULTIPLY-RECURSIVE is $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$, which is huge. (Problem 26-2 asks you to simplify this parallel algorithm at the expense of just a little less parallelism.)

Parallelizing Strassen's method

To parallelize Strassen's algorithm, we can follow the same general outline as on pages 86–87, but use spawning. You may find it helpful to compare each step below with the corresponding step there. We'll analyze costs as we go along to develop recurrences $T_1(n)$ and $T_\infty(n)$ for the overall work and span, respectively.

1. If $n = 1$, the matrices each contain a single element. Perform a single scalar multiplication and a single scalar addition, and return. Otherwise, partition the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (4.2) on page 82. This step takes $\Theta(1)$ work and $\Theta(1)$ span by index calculation.
2. Create $n/2 \times n/2$ matrices $S_1, S_2, \ldots, S_{10}$, each of which is the sum or difference of two submatrices from step 1. Create and zero the entries of seven $n/2 \times n/2$ matrices $P_1, P_2, \ldots, P_7$ to hold seven $n/2 \times n/2$ matrix products. All

17 matrices can be created, and the P_i initialized, with doubly nested **parallel for** loops using $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

3. Using the submatrices from step 1 and the matrices $S_1, S_2, \ldots, S_{10}$ created in step 2, recursively spawn computations of each of the seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$, taking $7T_1(n/2)$ work and $T_\infty(n/2)$ span.
4. Update the four submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding or subtracting various P_i matrices. Using doubly nested **parallel for** loops, computing all four submatrices takes $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

Let's analyze this algorithm. Since the serial projection is the same as the original serial algorithm, the work is just the running time of the serial projection, namely, $\Theta(n^{\lg 7})$. As we did with 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 (26.6) as we did for P-MATRIX-MULTIPLY-RECURSIVE, with solution $\Theta(\lg^2 n)$. Thus the parallel version of Strassen's method has parallelism $\Theta(n^{\lg 7}/\lg^2 n)$, which is large. Although the parallelism is slightly less than that of P-MATRIX-MULTIPLY-RECURSIVE, that's just because the work is also less.

Exercises

26.2-1

Draw the trace for computing P-MATRIX-MULTIPLY on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Assuming that each strand executes in unit time, analyze the work, span, and parallelism of this computation.

26.2-2

Repeat Exercise 26.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

26.2-3

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

26.2-4

Give pseudocode for an efficient parallel 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 equal 1. Analyze your algorithm.

26.2-5
Give pseudocode for an efficient parallel version of the Floyd-Warshall algorithm (see Section 23.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

26.3 Parallel 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 method, it seems like a terrific candidate for implementing using fork-join parallelism.

The procedure P-MERGE-SORT modifies merge sort to spawn the first recursive call. Like its serial counterpart MERGE-SORT on page 39, the P-MERGE-SORT procedure sorts the subarray $A[p:r]$. After the **sync** statement in line 8 ensures that the two recursive spawns in lines 5 and 7 have finished, P-MERGE-SORT calls the P-MERGE procedure, a parallel merging algorithm, which is on page 779, but you don't need to bother looking at it right now.

```
P-MERGE-SORT(A, p, r)
 1  if p ≥ r                    // zero or one element?
 2      return
 3  q = ⌊(p + r)/2⌋             // midpoint of A[p : r]
 4  // Recursively sort A[p : q] in parallel.
 5  spawn P-MERGE-SORT(A, p, q)
 6  // Recursively sort A[q + 1 : r] in parallel.
 7  spawn P-MERGE-SORT(A, q + 1, r)
 8  sync                        // wait for spawns
 9  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
10  P-MERGE(A, p, q, r)
```

First, let's use work/span analysis to get some intuition for why we need a parallel merge procedure. After all, it may seem as though there should be plenty of parallelism just by parallelizing MERGE-SORT without worrying about parallelizing the merge. But what would happen if the call to P-MERGE in line 10 of P-MERGE-SORT were replaced by a call to the serial MERGE procedure on page 36? Let's call the pseudocode so modified P-NAIVE-MERGE-SORT.

Let $T_1(n)$ be the (worst-case) work of P-NAIVE-MERGE-SORT on an n-element subarray, where $n = r - p + 1$ is the number of elements in $A[p:r]$, and let $T_\infty(n)$

be the span. Because MERGE is serial with running time $\Theta(n)$, both its work and span are $\Theta(n)$. Since the serial projection of P-NAIVE-MERGE-SORT is exactly MERGE-SORT, its work is $T_1(n) = \Theta(n \lg n)$. The two recursive calls in lines 5 and 7 run in parallel, and so its span is given by the recurrence

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

by case 1 of the master theorem. Thus the parallelism of P-NAIVE-MERGE-SORT is $T_1(n)/T_\infty(n) = \Theta(\lg n)$, which is an unimpressive amount of parallelism. To sort a million elements, for example, since $\lg 10^6 \approx 20$, it might achieve linear speedup on a few processors, but it would not scale up to dozens of processors.

The parallelism bottleneck in P-NAIVE-MERGE-SORT is plainly the MERGE procedure. If we asymptotically reduce the span of merging, the master theorem dictates that the span of parallel merge sort will also get smaller. When you look at the pseudocode for MERGE, it may seem that merging is inherently serial, but it's not. We can fashion a parallel merging algorithm. The goal is to reduce the span of parallel merging asymptotically, but if we want an efficient parallel algorithm, we must ensure that the $\Theta(n)$ bound on work doesn't increase.

Figure 26.6 depicts the divide-and-conquer strategy that we'll use in P-MERGE. The heart of the algorithm is a recursive auxiliary procedure P-MERGE-AUX that merges two sorted subarrays of an array A into a subarray of another array B in parallel. Specifically, P-MERGE-AUX merges $A[p_1 : r_1]$ and $A[p_2 : r_2]$ into subarray $B[p_3 : r_3]$, where $r_3 = p_3 + (r_1 - p_1 + 1) + (r_2 - p_2 + 1) - 1 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$.

The key idea of the recursive merging algorithm in P-MERGE-AUX is to split each of the two sorted subarrays of A around a pivot x, such that all the elements in the lower part of each subarray are at most x and all the elements in the upper part of each subarray are at least x. The procedure can then recurse in parallel on two subtasks: merging the two lower parts, and merging the two upper parts. The trick is to find a pivot x so that the recursion is not too lopsided. We don't want a situation such as that in QUICKSORT on page 183, where bad partitioning elements lead to a dramatic loss of asymptotic efficiency. We could opt to partition around a random element, as RANDOMIZED-QUICKSORT on page 192 does, but because the input subarrays are sorted, P-MERGE-AUX can quickly determine a pivot that always works well.

Specifically, the recursive merging algorithm picks the pivot x as the middle element of the larger of the two input subarrays, which we can assume without loss of generality is $A[p_1 : r_1]$, since otherwise, the two subarrays can just switch roles. That is, $x = A[q_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because $A[p_1 : r_1]$ is sorted, x is a median of the subarray elements: every element in $A[p_1 : q_1 - 1]$ is no more than x, and every element in $A[q_1 + 1 : r_1]$ is no less than x. Then the

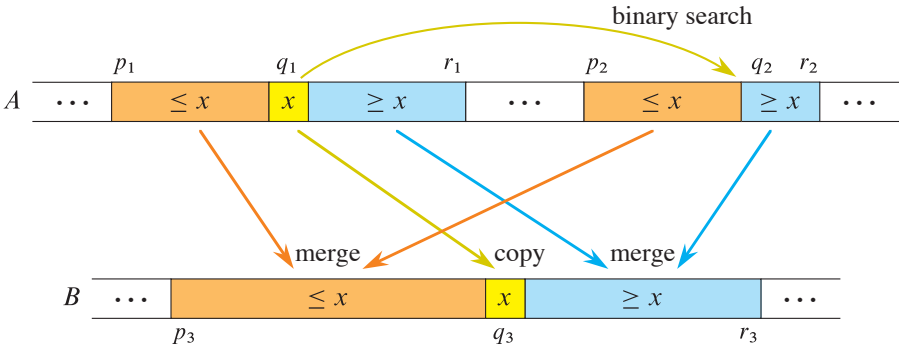


Figure 26.6 The idea behind P-MERGE-AUX, which merges two sorted subarrays $A[p_1 : r_1]$ and $A[p_2 : r_2]$ into the subarray $B[p_3 : r_3]$ in parallel. Letting $x = A[q_1]$ (shown in yellow) be a median of $A[p_1 : r_1]$ and q_2 be a place in $A[p_2 : r_2]$ such that x would fall between $A[q_2 - 1]$ and $A[q_2]$, every element in the subarrays $A[p_1 : q_1 - 1]$ and $A[p_2 : q_2 - 1]$ (shown in orange) is at most x, and every element in the subarrays $A[q_1 + 1 : r_1]$ and $A[q_2 + 1 : r_2]$ (shown in blue) is at least x. To merge, compute the index q_3 where x belongs in $B[p_3 : r_3]$, copy x into $B[q_3]$, and then recursively merge $A[p_1 : q_1 - 1]$ with $A[p_2 : q_2 - 1]$ into $B[p_3 : q_3 - 1]$ and $A[q_1 + 1 : r_1]$ with $A[q_2 : r_2]$ into $B[q_3 + 1 : r_3]$.

algorithm finds the "split point" q_2 in the smaller subarray $A[p_2 : r_2]$ such that all the elements in $A[p_2 : q_2 - 1]$ (if any) are at most x and all the elements in $A[q_2 : r_2]$ (if any) are at least x. Intuitively, the subarray $A[p_2 : r_2]$ would still be sorted if x were inserted between $A[q_2 - 1]$ and $A[q_2]$ (although the algorithm doesn't do that). Since $A[p_2 : r_2]$ is sorted, a minor variant of binary search (see Exercise 2.3-6) with x as the search key can find the split point q_2 in $\Theta(\lg n)$ time in the worst case. As we'll see when we get to the analysis, even if x splits $A[p_2 : r_2]$ badly—x is either smaller than all the subarray elements or larger—we'll still have at least $1/4$ of the elements in each of the two recursive merges. Thus the larger of the recursive merges operates on at most $3/4$ elements, and the recursion is guaranteed to bottom out after $\Theta(\lg n)$ recursive calls.

Now let's put these ideas into pseudocode. We start with the serial procedure FIND-SPLIT-POINT(A, p, r, x) on the next page, which takes as input a sorted subarray $A[p : r]$ and a key x. The procedure returns a split point of $A[p : r]$: an index q in the range $p \le q \le r + 1$ such that all the elements in $A[p : q - 1]$ (if any) are at most x and all the elements in $A[q : r]$ (if any) are at least x.

The FIND-SPLIT-POINT procedure uses binary search to find the split point. Lines 1 and 2 establish the range of indices for the search. Each time through the **while** loop, line 5 compares the middle element of the range with the search key x, and lines 6 and 7 narrow the search range to either the lower half or the upper half of the subarray, depending on the result of the test. In the end, after the range has been narrowed to a single index, line 8 returns that index as the split point.

```
FIND-SPLIT-POINT(A, p, r, x)
1  low = p                          // low end of search range
2  high = r + 1                     // high end of search range
3  while low < high                 // more than one element?
4      mid = ⌊(low + high)/2⌋       // midpoint of range
5      if x ≤ A[mid]                // is answer q ≤ mid?
6          high = mid               // narrow search to A[low : mid]
7      else low = mid + 1           // narrow search to A[mid + 1 : high]
8  return low
```

Because FIND-SPLIT-POINT contains no parallelism, its span is just its serial running time, which is also its work. On a subarray $A[p:r]$ of size $n = r - p + 1$, each iteration of the **while** loop halves the search range, which means that the loop terminates after $\Theta(\lg n)$ iterations. Since each iteration takes constant time, the algorithm runs in $\Theta(\lg n)$ (worst-case) time. Thus the procedure has work and span $\Theta(\lg n)$.

Let's now look at the pseudocode for the parallel merging procedure P-MERGE on the next page. Most of the pseudocode is devoted to the recursive procedure P-MERGE-AUX. The procedure P-MERGE itself is just a "wrapper" that sets up for P-MERGE-AUX. It allocates a new array $B[p:r]$ to hold the output of P-MERGE-AUX in line 1. It then calls P-MERGE-AUX in line 2, passing the indices of the two subarrays to be merged and providing B as the output destination of the merged result, starting at index p. After P-MERGE-AUX returns, lines 3–4 perform a parallel copy of the output $B[p:r]$ into the subarray $A[p:r]$, which is where P-MERGE-SORT expects it.

The P-MERGE-AUX procedure is the interesting part of the algorithm. Let's start by understanding the parameters of this recursive parallel procedure. The input array A and the four indices p_1, r_1, p_2, r_2 specify the subarrays $A[p_1:r_1]$ and $A[p_2:r_2]$ to be merged. The array B and the index p_3 indicate that the merged result should be stored into $B[p_3:r_3]$, where $r_3 = p_3 + (r_1 - p_1) + (r_2 - p_2) + 1$, as we saw earlier. The end index r_3 of the output subarray is not needed by the pseudocode, but it helps conceptually to name the end index, as in the comment in line 13.

The procedure begins by checking the base case of the recursion and doing some bookkeeping to simplify the rest of the pseudocode. Lines 1 and 2 test whether the two subarrays are both empty, in which case the procedure returns. Line 3 checks whether the first subarray contains fewer elements than the second subarray. Since the number of elements in the first subarray is $r_1 - p_1 + 1$ and the number in the second subarray is $r_2 - p_2 + 1$, the test omits the two "+1's." If the first subarray

```
P-MERGE(A, p, q, r)
 1  let B[p : r] be a new array                    // allocate scratch array
 2  P-MERGE-AUX(A, p, q, q + 1, r, B, p)           // merge from A into B
 3  parallel for i = p to r                        // copy B back to A in parallel
 4      A[i] = B[i]

P-MERGE-AUX(A, p1, r1, p2, r2, B, p3)
 1  if p1 > r1 and p2 > r2                         // are both subarrays empty?
 2      return
 3  if r1 − p1 < r2 − p2                           // second subarray bigger?
 4      exchange p1 with p2                        // swap subarray roles
 5      exchange r1 with r2
 6  q1 = ⌊(p1 + r1)/2⌋                             // midpoint of A[p1 : r1]
 7  x = A[q1]                                      // median of A[p1 : r1] is pivot x
 8  q2 = FIND-SPLIT-POINT(A, p2, r2, x)            // split A[p2 : r2] around x
 9  q3 = p3 + (q1 − p1) + (q2 − p2)                // where x belongs in B ...
10  B[q3] = x                                      // ... put it there
11  // Recursively merge A[p1 : q1 − 1] and A[p2 : q2 − 1] into B[p3 : q3 − 1].
12  spawn P-MERGE-AUX(A, p1, q1 − 1, p2, q2 − 1, B, p3)
13  // Recursively merge A[q1 + 1 : r1] and A[q2 : r2] into B[q3 + 1 : r3].
14  spawn P-MERGE-AUX(A, q1 + 1, r1, q2, r2, B, q3 + 1)
15  sync                                           // wait for spawns
```

is the smaller of the two, lines 4 and 5 switch the roles of the subarrays so that $A[p_1, r_1]$ refers to the larger subarray for the balance of the procedure.

We're now at the crux of P-MERGE-AUX: implementing the parallel divide-and-conquer strategy. As we continue our pseudocode walk, you may find it helpful to refer again to Figure 26.6.

First the divide step. Line 6 computes the midpoint q_1 of $A[p_1 : r_1]$, which indexes a median $x = A[q_1]$ of this subarray to be used as the pivot, and line 7 determines x itself. Next, line 8 uses the FIND-SPLIT-POINT procedure to find the index q_2 in $A[p_2 : r_2]$ such that all elements in $A[p_2 : q_2 - 1]$ are at most x and all the elements in $A[q_2 : r_2]$ are at least x. Line 9 computes the index q_3 of the element that divides the output subarray $B[p_3 : r_3]$ into $B[p_3 : q_3 - 1]$ and $B[q_3 + 1 : r_3]$, and then line 10 puts x directly into $B[q_3]$, which is where it belongs in the output.

Next is the conquer step, which is where the parallel recursion occurs. Lines 12 and 14 each spawn P-MERGE-AUX to recursively merge from A into B, the first to merge the smaller elements and the second to merge the larger elements. The

sync statement in line 15 ensures that the subproblems finish before the procedure returns.

There is no combine step, as $B[p:r]$ already contains the correct sorted output.

Work/span analysis of parallel merging

Let's first analyze the worst-case span $T_\infty(n)$ of P-MERGE-AUX on input subarrays that together contain a total of n elements. The call to FIND-SPLIT-POINT in line 8 contributes $\Theta(\lg n)$ to the span in the worst case, and the procedure performs at most a constant amount of additional serial work outside of the two recursive spawns in lines 12 and 14.

Because the two recursive spawns operate logically in parallel, only one of them contributes to the overall worst-case span. We claimed earlier that neither recursive invocation ever operates on more than $3n/4$ elements. Let's see why. Let $n_1 = r_1 - p_1 + 1$ and $n_2 = r_2 - p_2 + 1$, where $n = n_1 + n_2$, be the sizes of the two subarrays when line 6 starts executing, that is, after we have established that $n_2 \le n_1$ by swapping the roles of the two subarrays, if necessary. Since the pivot x is a median of of $A[p_1:r_1]$, in the worst case, a recursive merge involves at most $n_1/2$ elements of $A[p_1:r_1]$, but it might involve all n_2 of the elements of $A[p_2:r_2]$. Thus we can bound the number of elements involved in a recursive invocation of P-MERGE-AUX by

$$\begin{aligned} n_1/2 + n_2 &= (2n_1 + 4n_2)/4 \\ &\le (3n_1 + 3n_2)/4 \quad (\text{since } n_2 \le n_1) \\ &= 3n/4\,, \end{aligned}$$

proving the claim.

The worst-case span of P-MERGE-AUX can therefore be described by the following recurrence:

$$T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)\,. \tag{26.7}$$

Because this recurrence falls under case 2 of the master theorem with $k = 1$, its solution is $T_\infty(n) = \Theta(\lg^2 n)$.

Now let's verify that the work $T_1(n)$ of P-MERGE-AUX on n elements is linear. A lower bound of $\Omega(n)$ is straightforward, since each of the n elements is copied from array A to array B. We'll show that $T_1(n) = O(n)$ by deriving a recurrence for the worst-case work. The binary search in line 8 costs $\Theta(\lg n)$ in the worst case, which dominates the other work outside of the recursive spawns. For the recursive spawns, observe that although lines 12 and 14 might merge different numbers of elements, the two recursive spawns together merge at most $n - 1$ elements (since $x = A[q]$ is not merged). Moreover, as we saw when analyzing the span, a recursive spawn operates on at most $3n/4$ elements. We therefore obtain the recurrence

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)\,, \tag{26.8}$$

where α lies in the range $1/4 \le \alpha \le 3/4$. The value of α can vary from one recursive invocation to another.

We'll use the substitution method (see Section 4.3) to prove that the above recurrence (26.8) has solution $T_1(n) = O(n)$. (You could also use the Akra-Bazzi method from Section 4.7.) Assume that $T_1(n) \le c_1 n - c_2 \lg n$ for some positive constants c_1 and c_2. Using the properties of logarithms on pages 66–67—in particular, to deduce that $\lg\alpha + \lg(1-\alpha) = -\Theta(1)$—substitution yields

$$\begin{aligned}
T_1(n) &\le (c_1\alpha n - c_2\lg(\alpha n)) + (c_1(1-\alpha)n - c_2\lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1(\alpha + (1-\alpha))n - c_2(\lg(\alpha n) + \lg((1-\alpha)n)) + \Theta(\lg n) \\
&= c_1 n - c_2(\lg\alpha + \lg n + \lg(1-\alpha) + \lg n) + \Theta(\lg n) \\
&= c_1 n - c_2\lg n - c_2(\lg n + \lg\alpha + \lg(1-\alpha)) + \Theta(\lg n) \\
&= c_1 n - c_2\lg n - c_2(\lg n - \Theta(1)) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg n\,,
\end{aligned}$$

if we choose c_2 large enough that the $c_2(\lg n - \Theta(1))$ term dominates the $\Theta(\lg n)$ term for sufficiently large n. Furthermore, we can choose c_1 large enough to satisfy the implied $\Theta(1)$ base cases of the recurrence, completing the induction. The lower and upper bounds of $\Omega(n)$ and $O(n)$ give $T_1(n) = \Theta(n)$, asymptotically the same work as for serial merging.

The execution of the pseudocode in the P-MERGE procedure itself does not add asymptotically to the work and span of P-MERGE-AUX. The **parallel for** loop in lines 3–4 has $\Theta(\lg n)$ span due to the loop control, and each iteration runs in constant time. Thus the $\Theta(\lg^2 n)$ span of P-MERGE-AUX dominates, yielding $\Theta(\lg^2 n)$ span overall for P-MERGE. The **parallel for** loop contains $\Theta(n)$ work, matching the asymptotic work of P-MERGE-AUX and yielding $\Theta(n)$ work overall for P-MERGE.

Analysis of parallel merge sort

The "heavy lifting" is done. Now that we have determined the work and span of P-MERGE, we can analyze P-MERGE-SORT. Let $T_1(n)$ and $T_\infty(n)$ be the work and span, respectively, of P-MERGE-SORT on an array of n elements. The call to P-MERGE in line 10 of P-MERGE-SORT dominates the costs of lines 1–3, for both work and span. Thus we obtain the recurrence

$$T_1(n) = 2T_1(n/2) + \Theta(n)$$

for the work of P-MERGE-SORT, and we obtain the recurrence

$$T_\infty(n) = T_\infty(n/2) + \Theta(\lg^2 n)$$

for its span. The work recurrence has solution $T_1(n) = \Theta(n \lg n)$ by case 2 of the master theorem with $k = 0$. The span recurrence has solution $T_\infty(n) = \Theta(\lg^3 n)$, also by case 2 of the master theorem, but with $k = 2$.

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

$$\begin{aligned} T_1(n)/T_\infty(n) &= \Theta(n \lg n)/\Theta(\lg^3 n) \\ &= \Theta(n/\lg^2 n) \,, \end{aligned}$$

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. For example, you could switch to an efficient serial sort, perhaps quicksort, when the number of elements to be sorted is sufficiently small.

Exercises

26.3-1

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

26.3-2

Instead of finding a median element in the larger subarray, as P-MERGE does, suppose that the merge procedure finds a median of all the elements in the two sorted subarrays using the result of Exercise 9.3-10. Give pseudocode for an efficient parallel merging procedure that uses this median-finding procedure. Analyze your algorithm.

26.3-3

Give an efficient parallel algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 184. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (*Hint:* You might need an auxiliary array and might need to make more than one pass over the input elements.)

26.3-4

Give a parallel version of FFT on page 890. Make your implementation as parallel as possible. Analyze your algorithm.

★ *26.3-5*

Show how to parallelize SELECT from Section 9.3. Make your implementation as parallel as possible. Analyze your algorithm.

Problems

26-1 Implementing parallel loops using recursive spawning

Consider the parallel procedure SUM-ARRAYS 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, n)
1  parallel for i = 1 to n
2      C[i] = A[i] + B[i]
```

a. Rewrite the parallel loop in SUM-ARRAYS using recursive spawning in the manner of P-MAT-VEC-RECURSIVE. Analyze the parallelism.

Consider another implementation of the parallel loop in SUM-ARRAYS given by the procedure SUM-ARRAYS′, where the value *grain-size* must be specified.

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

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

b. Suppose that you set $grain\text{-}size = 1$. What is the resulting parallelism?

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.

26-2 Avoiding a temporary matrix in recursive matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure on page 772 must allocate a temporary matrix D of size $n \times n$, which can adversely affect the constants hidden by the Θ-notation. The procedure has high parallelism, however: $\Theta(n^3/\log^2 n)$.

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. Parallelize MATRIX-MULTIPLY-RECURSIVE without using temporary matrices so that it retains its $\Theta(n^3)$ work. (*Hint:* Spawn the recursive calls, but insert a **sync** in a judicious location to avoid races.)

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, and discuss whether the trade-off would be worthwhile.

26-3 Parallel matrix algorithms

Before attempting this problem, it may be helpful to read Chapter 28.

a. Parallelize the LU-DECOMPOSITION procedure on page 827 by giving pseudocode for a parallel 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 830.

c. Do the same for LUP-SOLVE on page 824.

d. Using equation (28.14) on page 835, write pseudocode for a parallel algorithm to invert a symmetric positive-definite matrix. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.

26-4 Parallel reductions and scan (prefix) computations

A ***$\otimes$-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 REDUCE 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. Design and analyze a parallel algorithm P-REDUCE that uses recursive spawning to perform the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span.

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

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

Parallelizing SCAN is not straightforward. For example, simply 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 procedures P-SCAN-1 and P-SCAN-1-AUX perform the $\otimes$-scan in parallel, albeit inefficiently.

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

The procedures P-SCAN-2 and P-SCAN-2-AUX use recursive spawning to perform a more efficient $\otimes$-scan.

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

To improve on both P-SCAN-1 and P-SCAN-2, perform the $\otimes$-scan in two distinct passes over the data. The first pass gathers the terms for various contiguous subarrays of x into a temporary array t, and the second pass uses the terms in t to compute the final result y. The pseudocode in the procedures P-SCAN-3, P-SCAN-UP, and P-SCAN-DOWN on the facing page implements this strategy, but certain expressions have been omitted.

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 the 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.

f. Describe how to rewrite P-SCAN-3 so that it doesn't require the use of the temporary array t.

★ ***g.*** Give an algorithm P-SCAN-4(x, n) for a scan that operates in place. It should place its output in x and require only constant auxiliary storage.

h. Describe an efficient parallel algorithm that uses a $+$-scan to determine whether a string of parentheses is well formed. For example, the string (( ) ( )) ( )

is well formed, but the string (( ))) (( ) is not. (*Hint:* Interpret (as a 1 and) as a -1, and then perform a $+$-scan.)

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

26-5 Parallelizing 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 14.4 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 $c[i-1, j-1]$.

This problem examines how to use recursive spawning to parallelize a simple stencil calculation on an $n \times n$ array A in which 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 the entries upon which $A[i, j]$ depends have been filled in, the entry $A[i, j]$ can be computed in $\Theta(1)$ time (as in the LCS-LENGTH procedure of Section 14.4).

Partition the $n \times n$ array A into four $n/2 \times n/2$ subarrays as follows:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}. \tag{26.9}$$

You can immediately fill in subarray A_{11} recursively, since it does not depend on the entries in the other three subarrays. Once the computation of A_{11} finishes, you can 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, you can fill in A_{22} recursively.

a. Give parallel pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (26.9) 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 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 parallel algorithm for this simple stencil calculation that achieves $\Theta(n/\lg n)$ parallelism. Argue using notions of work and span that

the problem has $\Theta(n)$ inherent parallelism. Unfortunately, simple fork-join parallelism does not let you achieve this maximal parallelism.

26-6 Randomized parallel algorithms

Like serial algorithms, parallel algorithms can employ random-number generators. This problem explores how to adapt the measures of work, span, and parallelism to handle the expected behavior of randomized task-parallel algorithms. It also asks you to design and analyze a parallel algorithm for randomized quicksort.

a. Explain how to modify the work law (26.2), span law (26.3), and greedy scheduler bound (26.4) to work with expectations when T_P, T_1, and T_∞ are all random variables.

b. Consider a randomized parallel algorithm for which 1% of the time, $T_1 = 10^4$ and $T_{10,000} = 1$, but for the remaining 99% of the time, $T_1 = T_{10,000} = 10^9$. Argue that the ***speedup*** of a randomized parallel 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 task-parallel algorithm should be defined as the ratio $\mathrm{E}[T_1]/\mathrm{E}[T_\infty]$.

d. Parallelize the RANDOMIZED-QUICKSORT algorithm on page 192 by using recursive spawning to produce P-RANDOMIZED-QUICKSORT. (Do not parallelize RANDOMIZED-PARTITION.)

e. Analyze your parallel algorithm for randomized quicksort. (*Hint:* Review the analysis of RANDOMIZED-SELECT on page 230.)

f. Parallelize RANDOMIZED-SELECT on page 230. Make your implementation as parallel as possible. Analyze your algorithm. (*Hint:* Use the partitioning algorithm from Exercise 26.3-3.)

Chapter notes

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 [58, 217] is another popular algorithmic programming model, which features operations on vectors and matrices as primitives. The notion of sequential consistency is due to Lamport [275].

Graham [197] and Brent [71] showed that there exist schedulers achieving the bound of Theorem 26.1. Eager, Zahorjan, and Lazowska [129] 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 [57] developed an algorithmic programming model based on work and span (which he called "depth") for data-parallel programming. Blumofe and Leiserson [63] gave a distributed scheduling algorithm for task-parallel computations 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 [20] and Blelloch, Gibbons, and Matias [61] also provided provably good algorithms for scheduling task-parallel computations. The recent literature contains many algorithms and strategies for scheduling parallel programs.

The parallel pseudocode and programming model were influenced by Cilk [290, 291, 383, 396]. The open-source project OpenCilk (www.opencilk.org) provides Cilk programming as an extension to the C and C++ programming languages. All of the parallel algorithms in this chapter can be coded straightforwardly in Cilk.

Concerns about nondeterministic parallel programs were expressed by Lee [281] and Bocchino, Adve, Adve, and Snir [64]. The algorithms literature contains many algorithmic strategies (see, for example, [60, 85, 118, 140, 160, 282, 283, 412, 461]) for detecting races and extending the fork-join model to avoid or safely embrace various kinds of nondeterminism. Blelloch, Fineman, Gibbons, and Shun [59] showed that deterministic parallel algorithms can often be as fast as, or even faster than, their nondeterministic counterparts.

Several of the parallel algorithms in this chapter appeared in unpublished lecture notes by C. E. Leiserson and H. Prokop and were originally implemented in Cilk. The parallel merge-sorting algorithm was inspired by an algorithm due to Akl [12].

27 Online Algorithms

Most problems described in this book have assumed that the entire input was available before the algorithm executes. In many situations, however, the input becomes available not in advance, but only as the algorithm executes. This idea was implicit in much of the discussion of data structures in Part III. The reason that you want to design, for example, a data structure that can handle n INSERT, DELETE, and SEARCH operations in $O(\lg n)$ time per operation is most likely because you are going to receive n such operation requests without knowing in advance what operations will be coming. This idea was also implicit in amortized analysis in Chapter 16, where we saw how to maintain a table that can grow or shrink in response to a sequence of insertion and deletion operations, yet with a constant amortized cost per operation.

An ***online algorithm*** receives its input progressively over time, rather than having the entire input available at the start, as in an ***offline algorithm***. Online algorithms pertain to many situations in which information arrives gradually. A stock trader must make decisions today, without knowing what the prices will be tomorrow, yet wants to achieve good returns. A computer system must schedule arriving jobs without knowing what work will need to be done in the future. A store must decide when to order more inventory without knowing what the future demand will be. A driver for a ride-hailing service must decide whether to pick up a fare without knowing who will request rides in the future. In each of these situations, and many more, algorithmic decisions must be made without knowledge of the future.

There are several approaches for dealing with unknown future inputs. One approach is to form a probabilistic model of future inputs and design an algorithm that assumes future inputs conform to the model. This technique is common, for example, in the field of queuing theory, and it is also related to machine learning. Of course, you might not be able to develop a workable probabilistic model, or even if you can, some inputs might not conform to it. This chapter takes a differ-

ent approach. Instead of assuming anything about the future input, we employ a conservative strategy of limiting how poor a solution any input can entail.

This chapter, therefore, adopts a worst-case approach, designing online algorithms that guarantee the quality of the solution for all possible future inputs. We'll analyze online algorithms by comparing the solution produced by the online algorithm with a solution produced by an optimal algorithm that knows the future inputs, and taking a worst-case ratio over all possible instances. We call this methodology ***competitive analysis***. We'll use a similar approach when we study approximation algorithms in Chapter 35, where we'll compare the solution returned by an algorithm that might be suboptimal with the value of the optimal solution, and determine a worst-case ratio over all possible instances.

We start with a "toy" problem: deciding between whether to take the elevator or the stairs. This problem will introduce the basic methodology of thinking about online algorithms and how to analyze them via competitive analysis. We will then look at two problems that use competitive analysis. The first is how to maintain a search list so that the access time is not too large, and the second is about strategies for deciding which cache blocks to evict from a cache or other kind of fast computer memory.

27.1 Waiting for an elevator

Our first example of an online algorithm models a problem that you likely have encountered yourself: whether you should wait for an elevator to arrive or just take the stairs. Suppose that you enter a building and wish to visit an office that is k floors up. You have two choices: walk up the stairs or take the elevator. Let's assume, for convenience, that you can climb the stairs at the rate of one floor per minute. The elevator travels much faster than you can climb the stairs: it can ascend all k floors in just one minute. Your dilemma is that you do not know how long it will take for the elevator to arrive at the ground floor and pick you up. Should you take the elevator or the stairs? How do you decide?

Let's analyze the problem. Taking the stairs takes k minutes, no matter what. Suppose you know that the elevator takes at most $B-1$ minutes to arrive for some value of B that is considerably higher than k. (The elevator could be going up when you call for it and then stop at several floors on its way down.) To keep things simple, let's also assume that the number of minutes for the elevator to arrive is an integer. Therefore, waiting for the elevator and taking it k floors up takes anywhere from one minute (if the elevator is already at the ground floor) to $(B-1)+1 = B$ minutes (the worst case). Although you know B and k, you don't know how long the elevator will take to arrive this time. You can use competitive

analysis to inform your decision regarding whether to take the stairs or elevator. In the spirit of competitive analysis, you want to be sure that, no matter what the future brings (i.e., how long the elevator takes to arrive), you will not wait much longer than a seer who knows when the elevator will arrive.

Let us first consider what the seer would do. If the seer knows that the elevator is going to arrive in at most $k - 1$ minutes, the seer waits for the elevator, and otherwise, the seer takes the stairs. Letting m denote the number of minutes it takes for the elevator to arrive at the ground floor, we can express the time that the seer spends as the function

$$t(m) = \begin{cases} m + 1 & \text{if } m \le k - 1 \ , \\ k & \text{if } m \ge k \ . \end{cases} \tag{27.1}$$

We typically evaluate online algorithms by their ***competitive ratio***. Let $\mathcal{U}$ denote the set (universe) of all possible inputs, and consider some input $I \in \mathcal{U}$. For a minimization problem, such as the stairs-versus-elevator problem, if an online algorithm A produces a solution with value $A(I)$ on input I and the solution from an algorithm F that knows the future has value $F(I)$ on the same input, then the competitive ratio of algorithm A is

$$\max \{A(I)/F(I) : I \in \mathcal{U}\} \ .$$

If an online algorithm has a competitive ratio of c, we say that it is ***c-competitive***. The competitive ratio is always at least 1, so that we want an online algorithm with a competitive ratio as close to 1 as possible.

In the stairs-versus-elevator problem, the only input is the time for the elevator to arrive. Algorithm F knows this information, but an online algorithm has to make a decision without knowing when the elevator will arrive. Consider the algorithm "always take the stairs," which always takes exactly k minutes. Using equation (27.1), the competitive ratio is

$$\max \{k/t(m) : 0 \le m \le B - 1\} \ . \tag{27.2}$$

Enumerating the terms in equation (27.2) gives the competitive ratio as

$$\max \left\{ \frac{k}{1}, \frac{k}{2}, \frac{k}{3}, \ldots, \frac{k}{(k-1)}, \frac{k}{k}, \frac{k}{k}, \ldots, \frac{k}{k} \right\} = k \ ,$$

so that the competitive ratio is k. The maximum is achieved when the elevator arrives immediately. In this case, taking the stairs requires k minutes, but the optimal solution takes just 1 minute.

Now let's consider the opposite approach: "always take the elevator." If it takes m minutes for the elevator to arrive at the ground floor, then this algorithm will always take $m + 1$ minutes. Thus the competitive ratio becomes

$$\max \{(m + 1)/t(m) : 0 \le m \le B - 1\} \ ,$$

which we can again enumerate as

$$\max\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{k}{k}, \frac{k+1}{k}, \frac{k+2}{k}, \ldots, \frac{B}{k}\right\} = \frac{B}{k} .$$

Now the maximum is achieved when the elevator takes $B - 1$ minutes to arrive, compared with the optimal approach of taking the stairs, which requires k minutes.

Hence, the algorithm "always take the stairs" has competitive ratio k, and the algorithm "always take the elevator" has competitive ratio B/k. Because we prefer the algorithm with smaller competitive ratio, if $k = 10$ and $B = 300$, we prefer "always take the stairs," with competitive ratio 10, over "always take the elevator," with competitive ratio 30. Taking the stairs is not always better, or necessarily more often better. It's just that taking the stairs guards better against the worst-case future.

These two approaches of always taking the stairs and always taking the elevator are extreme solutions, however. Instead, you can "hedge your bets" and guard even better against a worst-case future. In particular, you can wait for the elevator for a while, and then if it doesn't arrive, take the stairs. How long is "a while"? Let's say that "a while" is k minutes. Then the time $h(m)$ required by this hedging strategy, as a function of the number m of minutes before the elevator arrives, is

$$h(m) = \begin{cases} m+1 & \text{if } m \le k , \\ 2k & \text{if } m > k . \end{cases}$$

In the second case, $h(m) = 2k$ because you wait for k minutes and then climb the stairs for k minutes. The competitive ratio is now

$$\max\{h(m)/t(m) : 0 \le m \le B - 1\} .$$

Enumerating this ratio yields

$$\max\left\{\frac{1}{1}, \frac{2}{2}, \ldots, \frac{k}{k}, \frac{k+1}{k}, \frac{2k}{k}, \frac{2k}{k}, \frac{2k}{k}, \ldots, \frac{2k}{k}\right\} = 2 .$$

The competitive ratio is now *independent* of k and B.

This example illustrates a common philosophy in online algorithms: we want an algorithm that guards against any possible worst case. Initially, waiting for the elevator guards against the case when the elevator arrives quickly, but eventually switching to the stairs guards against the case when the elevator takes a long time to arrive.

Exercises

27.1-1

Suppose that when hedging your bets, you wait for p minutes, instead of for k minutes, before taking the stairs. What is the competitive ratio as a function of p and k? How should you choose p to minimize the competitive ratio?

27.1-2

Imagine that you decide to take up downhill skiing. Suppose that a pair of skis costs r dollars to rent for a day and b dollars to buy, where $b > r$. If you knew in advance how many days you would ever ski, your decision whether to rent or buy would be easy. If you'll ski for at least $\lceil b/r \rceil$ days, then you should buy skis, and otherwise you should rent. This strategy minimizes the total that you ever spend. In reality, you don't know in advance how many days you'll eventually ski. Even after you have skied several times, you still don't know how many more times you'll ever ski. Yet you don't want to waste your money. Give and analyze an algorithm that has a competitive ratio of 2, that is, an algorithm guaranteeing that, no matter how many times you ski, you never spend more than twice what you would have spent if you knew from the outset how many times you'll ski.

27.1-3

In "concentration solitaire," a game for one person, you have n pairs of matching cards. The backs of the cards are all the same, but the fronts contain pictures of animals. One pair has pictures of aardvarks, one pair has pictures of bears, one pair has pictures of camels, and so on. At the start of the game, the cards are all placed face down. In each round, you can turn two cards face up to reveal their pictures. If the pictures match, then you remove that pair from the game. If they don't match, then you turn both of them over, hiding their pictures once again. The game ends when you have removed all n pairs, and your score is how many rounds you needed to do so. Suppose that you can remember the picture on every card that you have seen. Give an algorithm to play concentration solitaire that has a competitive ratio of 2.

27.2 Maintaining a search list

The next example of an online algorithm pertains to maintaining the order of elements in a linked list, as in Section 10.2. This problem often arises in practice for hash tables when collisions are resolved by chaining (see Section 11.2), since each slot contains a linked list. Reordering the linked list of elements in each slot of the hash table can boost the performance of searches measurably.

The list-maintenance problem can be set up as follows. You are given a list L of n elements $\{x_1, x_2, \ldots, x_n\}$. We'll assume that the list is doubly linked, although the algorithms and analysis work just as well for singly linked lists. Denote the position of element x_i in the list L by $r_L(x_i)$, where $1 \le r_L(x_i) \le n$. Calling LIST-SEARCH(L, x_i) on page 260 thus takes $\Theta(r_L(x_i))$ time.

If you know in advance something about the distribution of search requests, then it makes sense to arrange the list ahead of time to put the more frequently searched elements closer to the front, which minimizes the total cost (see Exercise 27.2-1). If instead you don't know anything about the search sequence, then no matter how you arrange the list, it is possible that every search is for whatever element appears at the tail of the list. The total searching time would then be $\Theta(nm)$, where m is the number of searches.

If you notice patterns in the access sequence or you observe differences in the frequencies in which elements are accessed, then you might want to rearrange the list as you perform searches. For example, if you discover that every search is for a particular element, you could move that element to the front of the list. In general, you could rearrange the list after each call to LIST-SEARCH. But how would you do so without knowing the future? After all, no matter how you move elements around, every search could be for the last element.

But it turns out that some search sequences are "easier" than others. Rather than just evaluate performance on the worst-case sequence, let's compare a reorganization scheme with whatever an optimal offline algorithm would do if it knew the search sequence in advance. That way, if the sequence is fundamentally hard, the optimal offline algorithm will also find it hard, but if the sequence is easy, we can hope to do reasonably well.

To ease analysis, we'll drop the asymptotic notation and say that the cost is just i to search for the ith element in the list. Let's also assume that the only way to reorder the elements in the list is by swapping two adjacent elements in the list. Because the list is doubly linked, each swap incurs a cost of 1. Thus, for example, a search for the sixth element followed by moving it forward two places (entailing two swaps) incurs a total cost 8. The goal is to minimize the total cost of calls to LIST-SEARCH plus the total number of swaps performed.

The online algorithm that we'll explore is MOVE-TO-FRONT(L, x). This procedure first searches for x in the doubly linked list L, and then it moves x to the front of the list.[1] If x is located at position $r = r_L(x)$ before the call, MOVE-TO-FRONT swaps x with the element in position $r - 1$, then with the element in position $r - 2$,

[1] The path-compression heuristic in Section 19.3 resembles MOVE-TO-FRONT, although it would be more accurately expressed as "move-to-next-to-front." Unlike MOVE-TO-FRONT in a doubly linked list, path compression can relocate multiple elements to become "next-to-front."

	FORESEE					MOVE-TO-FRONT				
element searched	L	search cost	swap cost	search + swap cost	cumulative cost	L	search cost	swap cost	search + swap cost	cumulative cost
5	$\langle 1,2,3,4,5\rangle$	5	0	5	5	$\langle 1,2,3,4,5\rangle$	5	4	9	9
3	$\langle 1,2,3,4,5\rangle$	3	3	6	11	$\langle 5,1,2,3,4\rangle$	4	3	7	16
4	$\langle 4,1,2,3,5\rangle$	1	0	1	12	$\langle 3,5,1,2,4\rangle$	5	4	9	25
4	$\langle 4,1,2,3,5\rangle$	1	0	1	13	$\langle 4,3,5,1,2\rangle$	1	0	1	26

Figure 27.1 The costs incurred by the procedures FORESEE and MOVE-TO-FRONT when searching for the elements 5, 3, 4, and 4, starting with the list $L = \langle 1, 2, 3, 4, 5\rangle$. If FORESEE instead moved 3 to the front after the search for 5, the cumulative cost would not change, nor would the cumulative cost change if 4 moved to the second position after the search for 5.

and so on, until it finally swaps x with the element in position 1. Thus if the call MOVE-TO-FRONT$(L, 8)$ executes on the list $L = \langle 5, 3, 12, 4, 8, 9, 22\rangle$, the list becomes $\langle 8, 5, 3, 12, 4, 9, 22\rangle$. The call MOVE-TO-FRONT$(L, k)$ costs $2r_L(k) - 1$: it costs $r_L(k)$ to search for k, and it costs 1 for each of the $r_L(k) - 1$ swaps that move k to the front of the list.

We'll see that MOVE-TO-FRONT has a competitive ratio of 4. Let's think about what this means. MOVE-TO-FRONT performs a series of operations on a doubly linked list, accumulating cost. For comparison, suppose that there is an algorithm FORESEE that knows the future. Like MOVE-TO-FRONT, it also searches the list and moves elements around, but after each call it optimally rearranges the list for the future. (There may be more than one optimal order.) Thus FORESEE and MOVE-TO-FRONT maintain different lists of the same elements.

Consider the example shown in Figure 27.1. Starting with the list $\langle 1, 2, 3, 4, 5\rangle$, four searches occur, for the elements 5, 3, 4, and 4. The hypothetical procedure FORESEE, after searching for 3, moves 4 to the front of the list, knowing that a search for 4 is imminent. It thus incurs a swap cost of 3 upon its second call, after which no further swap costs accrue. MOVE-TO-FRONT incurs swap costs in each step, moving the found element to the front. In this example, MOVE-TO-FRONT has a higher cost in each step, but that is not necessarily always the case.

The key to proving the competitive bound is to show that at any point, the total cost of MOVE-TO-FRONT is not much higher than that of FORESEE. Surprisingly, we can determine a bound on the costs incurred by MOVE-TO-FRONT relative to FORESEE even though MOVE-TO-FRONT cannot see the future.

If we compare any particular step, MOVE-TO-FRONT and FORESEE may be operating on very different lists and do very different things. If we focus on the search for 4 above, we observe that FORESEE actually moves it to the front of the list early, paying to move the element to the front before it is accessed. To capture this con-

cept, we use the idea of an ***inversion***: a pair of elements, say a and b, in which a appears before b in one list, but b appears before a in another list. For two lists L and L', let $I(L, L')$, called the ***inversion count***, denote the number of inversions between the two lists, that is, the number of pairs of elements whose order differs in the two lists. For example, with lists $L = \langle 5, 3, 1, 4, 2\rangle$ and $L' = \langle 3, 1, 2, 4, 5\rangle$, then out of the $\binom{5}{2} = 10$ pairs, exactly five of them—$(1, 5), (2, 4), (2, 5), (3, 5), (4, 5)$—are inversions, since these pairs, and only these pairs, appear in different orders in the two lists. Thus the inversion count is $I(L, L') = 5$.

In order to analyze the algorithm, we define the following notation. Let L_i^M be the list maintained by MOVE-TO-FRONT immediately after the ith search, and similarly, let L_i^F be FORESEE's list immediately after the ith search. Let c_i^M and c_i^F be the costs incurred by MOVE-TO-FRONT and FORESEE on their ith calls, respectively. We don't know how many swaps FORESEE performs in its ith call, but we'll denote that number by t_i. Therefore, if the ith operation is a search for element x, then

$$c_i^M = 2r_{L_{i-1}^M}(x) - 1 , \tag{27.3}$$

$$c_i^F = r_{L_{i-1}^F}(x) + t_i . \tag{27.4}$$

In order to compare these costs more carefully, let's break down the elements into subsets, depending on their positions in the two lists before the ith search, relative to the element x being searched for in the ith search. We define three sets:

$$\begin{aligned} BB &= \{\text{elements before } x \text{ in both } L_{i-1}^M \text{ and } L_{i-1}^F\} , \\ BA &= \{\text{elements before } x \text{ in } L_{i-1}^M \text{ but after } x \text{ in } L_{i-1}^F\} , \\ AB &= \{\text{elements after } x \text{ in } L_{i-1}^M \text{ but before } x \text{ in } L_{i-1}^F\} . \end{aligned}$$

We can now relate the position of element x in L_{i-1}^F and L_{i-1}^M to the sizes of these sets:

$$r_{L_{i-1}^M}(x) = |BB| + |BA| + 1 , \tag{27.5}$$

$$r_{L_{i-1}^F}(x) = |BB| + |AB| + 1 . \tag{27.6}$$

When a swap occurs in one of the lists, it changes the relative positions of the two elements involved, which in turn changes the inversion count. Suppose that elements x and y are swapped in some list. Then the only possible difference in the inversion count between this list and *any* other list depends on whether (x, y) is an inversion. In fact, the inversion count of (x, y) with respect to any other list *must* change. If (x, y) is an inversion before the swap, it no longer is afterward, and vice versa. Therefore, if two consecutive elements x and y swap positions in a list L, then for any other list L', the value of the inversion count $I(L, L')$ either increases by 1 or decreases by 1.

As we compare MOVE-TO-FRONT and FORESEE searching and modifying their lists, we'll think about MOVE-TO-FRONT executing on its list for the ith time and then FORESEE executing on its list for the ith time. After MOVE-TO-FRONT has executed for the ith time and before FORESEE has executed for the ith time, we'll compare $I(L_{i-1}^M, L_{i-1}^F)$ (the inversion count immediately before the ith call of MOVE-TO-FRONT) with $I(L_i^M, L_{i-1}^F)$ (the inversion count after the ith call of MOVE-TO-FRONT but before the ith call of FORESEE). We'll concern ourselves later with what FORESEE does.

Let us analyze what happens to the inversion count after executing the ith call of MOVE-TO-FRONT, and suppose that it searches for element x. More precisely, we'll compute $I(L_i^M, L_{i-1}^F) - I(L_{i-1}^M, L_{i-1}^F)$, the change in the inversion count, which gives a rough idea of how much MOVE-TO-FRONT's list becomes more or less like FORESEE's list. After searching, MOVE-TO-FRONT performs a series of swaps with each of the elements on the list L_{i-1}^M that precedes x. Using the notation above, the number of such swaps is $|BB| + |BA|$. Bearing in mind that the list L_{i-1}^F has yet to be changed by the ith call of FORESEE, let's see how the inversion count changes.

Consider a swap with an element $y \in BB$. Before the swap, y precedes x in both L_{i-1}^M and L_{i-1}^F. After the swap, x precedes y in L_i^M, and L_{i-1}^F does not change. Therefore, the inversion count increases by 1 for each element in BB. Now consider a swap with an element $z \in BA$. Before the swap, z precedes x in L_{i-1}^M but x precedes z in L_{i-1}^F. After the swap, x precedes z in both lists. Therefore, the inversion count decreases by 1 for each element in BA. Thus altogether, the inversion count increases by

$$I(L_i^M, L_{i-1}^F) - I(L_{i-1}^M, L_{i-1}^F) = |BB| - |BA| \ . \tag{27.7}$$

We have laid the groundwork needed to analyze MOVE-TO-FRONT.

Theorem 27.1
Algorithm MOVE-TO-FRONT has a competitive ratio of 4.

Proof The proof uses a potential function, as described in Chapter 16 on amortized analysis. The value Φ_i of the potential function after the ith calls of MOVE-TO-FRONT and FORESEE depends on the inversion count:

$$\Phi_i = 2I(L_i^M, L_i^F) \ .$$

(Intuitively, the factor of 2 embodies the notion that each inversion represents a cost of 2 for MOVE-TO-FRONT relative to FORESEE: 1 for searching and 1 for swapping.) By equation (27.7), after the ith call of MOVE-TO-FRONT, but before the ith call of FORESEE, the potential increases by $2(|BB| - |BA|)$. Since the inversion count of the two lists is nonnegative, we have $\Phi_i \geq 0$ for all $i \geq 0$.

Assuming that MOVE-TO-FRONT and FORESEE start with the same list, the initial potential Φ_0 is 0, so that $\Phi_i \geq \Phi_0$ for all i.

Drawing from equation (16.2) on page 456, the amortized cost $\hat{c}_i^M$ of the ith MOVE-TO-FRONT operation is

$$\hat{c}_i^M = c_i^M + \Phi_i - \Phi_{i-1} ,$$

where c_i^M, the actual cost of the ith MOVE-TO-FRONT operation, is given by equation (27.3):

$$c_i^M = 2r_{L_{i-1}^M}(x) - 1 .$$

Now, let's consider the potential change $\Phi_i - \Phi_{i-1}$. Since both L^M and L^F change, let's consider the changes to one list at a time. Recall that when MOVE-TO-FRONT moves element x to the front, it increases the potential by exactly $2(|BB| - |BA|)$. We now consider how the optimal algorithm FORESEE changes its list L^F: it performs t_i swaps. Each swap performed by FORESEE either increases or decreases the potential by 2, and thus the increase in potential by FORESEE in the ith call can be at most $2t_i$. We therefore have

$$\begin{aligned}
\hat{c}_i^M &= c_i^M + \Phi_i - \Phi_{i-1} \\
&\leq 2r_{L_{i-1}^M}(x) - 1 + 2(|BB| - |BA| + t_i) \\
&= 2r_{L_{i-1}^M}(x) - 1 + 2(|BB| - (r_{L_{i-1}^M}(x) - 1 - |BB|) + t_i) \\
&\qquad\qquad \text{(by equation (27.5))} \\
&= 4\,|BB| + 1 + 2t_i \\
&\leq 4\,|BB| + 4\,|AB| + 4 + 4t_i \qquad \text{(increasing some terms)} \\
&= 4(|BB| + |AB| + 1 + t_i) \\
&= 4(r_{L_{i-1}^F}(x) + t_i) \qquad \text{(by equation (27.6))} \\
&= 4c_i^F \qquad \text{(by equation (27.4)) .} \qquad (27.8)
\end{aligned}$$

We now finish the proof as in Chapter 16 by showing that the total amortized cost provides an upper bound on the total actual cost, because the initial potential function is 0 and the potential function is always nonnegative. By equation (16.3) on page 456, for any sequence of m MOVE-TO-FRONT operations, we have

$$\begin{aligned}
\sum_{i=1}^{m} \hat{c}_i^M &= \sum_{i=1}^{m} c_i^M + \Phi_m - \Phi_0 \\
&\geq \sum_{i=1}^{m} c_i^M \qquad \text{(because } \Phi_m \geq \Phi_0) . \qquad (27.9)
\end{aligned}$$

Therefore, we have

$$\begin{aligned}\sum_{i=1}^{m} c_i^M &\le \sum_{i=1}^{m} \widehat{c}_i^M && \text{(by equation (27.9))}\\ &\le \sum_{i=1}^{m} 4c_i^F && \text{(by equation (27.8))}\\ &= 4\sum_{i=1}^{m} c_i^F \ .\end{aligned}$$

Thus the total cost of the m MOVE-TO-FRONT operations is at most 4 times the total cost of the m FORESEE operations, so MOVE-TO-FRONT is 4-competitive. ■

Isn't it amazing that we can compare MOVE-TO-FRONT with the optimal algorithm FORESEE when we have no idea of the swaps that FORESEE makes? We were able to relate the performance of MOVE-TO-FRONT to the optimal algorithm by capturing how particular properties (swaps in this case) must evolve relative to the optimal algorithm, without actually knowing the optimal algorithm.

The online algorithm MOVE-TO-FRONT has a competitive ratio of 4: on any input sequence, it incurs a cost at most 4 times that of any other algorithm. On a particular input sequence, it could cost much less than 4 times the optimal algorithm, perhaps even matching the optimal algorithm.

Exercises

27.2-1

You are given a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements, and you wish to make a static list L (no rearranging once the list is created) containing the elements of S that is good for searching. Suppose that you have a probability distribution, where $p(x_i)$ is the probability that a given search searches for element x_i. Argue that the expected cost for m searches is

$$m\sum_{i=1}^{n} p(x_i) \cdot r_L(x_i) \ .$$

Prove that this sum is minimized when the elements of L are sorted in decreasing order with respect to $p(x_i)$.

27.2-2

Professor Carnac claims that since FORESEE is an optimal algorithm that knows the future, then at each step it must incur no more cost than MOVE-TO-FRONT. Either prove that Professor Carnac is correct or provide a counterexample.

27.2-3

Another way to maintain a linked list for efficient searching is for each element to maintain a ***frequency count***: the number of times that the element has been searched for. The idea is to rearrange list elements after searches so that the list is always sorted by decreasing frequency count, from largest to smallest. Either show that this algorithm is $O(1)$-competitive, or prove that it is not.

27.2-4

The model in this section charged a cost of 1 for each swap. We can consider an alternative cost model in which, after accessing x, you can move x anywhere earlier in the list, and there is no cost for doing so. The only cost is the cost of the actual accesses. Show that MOVE-TO-FRONT is 2-competitive in this cost model, assuming that the number requests is sufficiently large. (*Hint:* Use the potential function $\Phi_i = I(L_i^M, L_i^F)$.)

27.3 Online caching

In Section 15.4, we studied the caching problem, in which ***blocks*** of data from the main memory of a computer are stored in the ***cache***: a small but faster memory. In that section, we studied the offline version of the problem, in which we assumed that we knew the sequence of memory requests in advance, and we designed an algorithm to minimize the number of cache misses. In almost all computer systems, caching is, in fact, an online problem. We do not generally know the series of cache requests in advance; they are presented to the algorithm only as the requests for blocks are actually made. To gain a better understanding of this more realistic scenario, we analyze online algorithms for caching. We will first see that all deterministic online algorithms for caching have a lower bound of $\Omega(k)$ for the competitive ratio, where k is the size of the cache. We will then present an algorithm with a competitive ratio of $\Theta(n)$, where the input size is n, and one with a competitive ratio of $O(k)$, which matches the lower bound. We will end by showing how to use randomization to design an algorithm with a much better competitive ratio of $\Theta(\lg k)$. We will also discuss the assumptions that underlie randomized online algorithms, via the notion of an adversary, such as we saw in Chapter 11 and will see in Chapter 31.

You can find the terminology used to describe the caching problem in Section 15.4, which you might wish to review before proceeding.

27.3.1 Deterministic caching algorithms

In the caching problem, the input comprises a sequence of n memory requests, for data in blocks $b_1, b_2, \ldots, b_n$, in that order. The blocks requested are not necessarily distinct: each block may appear multiple times within the request sequence. After block b_i is requested, it resides in a cache that can hold up to k blocks, where k is a fixed cache size. We assume that $n > k$, since otherwise we are assured that the cache can hold all the requested blocks at once. When a block b_i is requested, if it is already in the cache, then a ***cache hit*** occurs and the cache remains unchanged. If b_i is not in the cache, then a ***cache miss*** occurs. If the cache contains fewer than k blocks upon a cache miss, block b_i is placed into the cache, which now contains one block more than before. If a cache miss occurs with an already full cache, however, some block must be evicted from the cache before b_i can enter. Thus, a caching algorithm must decide which block to evict from the cache upon a cache miss when the cache is full. The goal is to minimize the number of cache misses over the entire request sequence. The caching algorithms considered in this chapter differ only in which block they decide to evict upon a cache miss. We do not consider abilities such as prefetching, in which a block is brought into the cache before an upcoming request in order to avert a future cache miss.

There are many online caching policies to determine which block to evict, including the following:

- First-in, first-out (FIFO): evict the block that has been in the cache the longest time.
- Last-in, first-out (LIFO): evict the block that has been in the cache the shortest time.
- Least Recently Used (LRU): evict the block whose last use is furthest in the past.
- Least Frequently Used (LFU): evict the block that has been accessed the fewest times, breaking ties by choosing the block that has been in the cache the longest.

To analyze these algorithms, we assume that the cache starts out empty, so that no evictions occur during the first k requests. We wish to compare the performance of an online algorithm to an optimal offline algorithm that knows the future requests. As we will soon see, all these deterministic online algorithms have a lower bound of $\Omega(k)$ for their competitive ratio. Some deterministic algorithms also have a competitive ratio with an $O(k)$ upper bound, but some other deterministic algorithms are considerably worse, having a competitive ratio of $\Theta(n/k)$.

We now proceed to analyze the LIFO and LRU policies. In addition to assuming that $n > k$, we will assume that at least k distinct blocks are requested. Otherwise, the cache never fills up and no blocks are evicted, so that all algorithms exhibit the same behavior. We begin by showing that LIFO has a large competitive ratio.

Theorem 27.2
LIFO has a competitive ratio of $\Theta(n/k)$ for the online caching problem with n requests and a cache of size k.

Proof We first show a lower bound of $\Omega(n/k)$. Suppose that the input consists of $k+1$ blocks, numbered $1, 2, \ldots, k+1$, and the request sequence is

$$1,\ 2,\ 3,\ 4,\ \ldots,\ k,\ k+1,k,k\quad+1,k,k\quad+1,\ \ldots\ ,$$

where after the initial $1, 2, \ldots, k, k+1$, the remainder of the sequence alternates between k and $k+1$, with a total of n requests. The sequence ends on block k if n and k are either both even or both odd, and otherwise, the sequence ends on block $k+1$. That is, $b_i = i$ for $i = 1, 2, \ldots k-1$, $b_i = k+1$ for $i = k+1, k+3, \ldots$ and $b_i = k$ for $i = k, k+2, \ldots$. How many blocks does LIFO evict? After the first k requests (which are considered to be cache misses), the cache is filled with blocks $1, 2, \ldots, k$. The $(k+1)$st request, which is for block $k+1$, causes block k to be evicted. The $(k+2)$nd request, which is for block k, forces block $k+1$ to be evicted, since that block was just placed into the cache. This behavior continues, alternately evicting blocks k and $k+1$ for the remaining requests. LIFO, therefore, suffers a cache miss on every one of the n requests.

The optimal offline algorithm knows the entire sequence of requests in advance. Upon the first request of block $k+1$, it just evicts any block except block k, and then it never evicts another block. Thus, the optimal offline algorithm evicts only once. Since the first k requests are considered cache misses, the total number of cache misses is $k+1$. The competitive ratio, therefore, is $n/(k+1)$, or $\Omega(n/k)$.

For the upper bound, observe that on any input of size n, any caching algorithm incurs at most n cache misses. Because the input contains at least k distinct blocks, any caching algorithm, including the optimal offline algorithm, must incur at least k cache misses. Therefore, LIFO has a competitive ratio of $O(n/k)$. ■

We call such a competitive ratio ***unbounded***, because it grows with the input size. Exercise 27.3-2 asks you to show that LFU also has an unbounded competitive ratio.

FIFO and LRU have a much better competitive ratio of $\Theta(k)$. There is a big difference between competitive ratios of $\Theta(n/k)$ and $\Theta(k)$. The cache size k is independent of the input sequence and does not grow as more requests arrive over time. A competitive ratio that depends on n, on the other hand, does grow with the size of the input sequence and thus can get quite large. It is preferable to use an algorithm with a competitive ratio that does not grow with the input sequence's size, when possible.

We now show that LRU has a competitive ratio of $\Theta(k)$, first showing the upper bound.

Theorem 27.3
LRU has a competitive ratio of $O(k)$ for the online caching problem with n requests and a cache of size k.

Proof To analyze LRU, we will divide the sequence of requests into ***epochs***. Epoch 1 begins with the first request. Epoch i, for $i > 1$, begins upon encountering the $(k+1)$st distinct request since the beginning of epoch $i-1$. Consider the following example of requests with $k = 3$:

$$1, 2, 1, 5, 4, 4, 1, 2, 4, 2, 3, 4, 5, 2, 2, 1, 2, 2\,. \tag{27.10}$$

The first $k = 3$ distinct requests are for blocks 1, 2 and 5, so epoch 2 begins with the first request for block 4. In epoch 2, the first 3 distinct requests are for blocks 4, 1, and 2. Requests for these blocks recur until the request for block 3, and with this request epoch 3 begins. Thus, this example has four epochs:

$$1,2,1,5 \qquad 4, 4, 1, 2, 4, 2 \qquad 3,4,5 \qquad 2, 2, 1, 2, 2\,. \tag{27.11}$$

Now we consider the behavior of LRU. In each epoch, the first time a request for a particular block appears, it may cause a cache miss, but subsequent requests for that block within the epoch cannot cause a cache miss, since the block is now one of the k most recently used. For example, in epoch 2, the first request for block 4 causes a cache miss, but the subsequent requests for block 4 do not. (Exercise 27.3-1 asks you to show the contents of the cache after each request.) In epoch 3, requests for blocks 3 and 5 cause cache misses, but the request for block 4 does not, because it was recently accessed in epoch 2. Since only the first request for a block within an epoch can cause a cache miss and the cache holds k blocks, each epoch incurs at most k cache misses.

Now consider the behavior of the optimal algorithm. The first request in each epoch must cause a cache miss, even for an optimal algorithm. The miss occurs because, by the definition of an epoch, there *must* have been k other blocks accessed since the last access to this block.

Since, for each epoch, the optimal algorithm incurs at least one miss and LRU incurs at most k, the competitive ratio is at most $k/1 = O(k)$. ■

Exercise 27.3-3 asks you to show that FIFO also has a competitive ratio of $O(k)$.

We could show lower bounds of $\Omega(k)$ on LRU and FIFO, but in fact, we can make a much stronger statement: *any* deterministic online caching algorithm must have a competitive ratio of $\Omega(k)$. The proof relies on an adversary who knows the online algorithm being used and can tailor the future requests to cause the online algorithm to incur more cache misses than the optimal offline algorithm.

Consider a scenario in which the cache has size k and the set of possible blocks to request is $\{1, 2, \ldots, k+1\}$. The first k requests are for blocks $1, 2, \ldots, k$, so

that both the adversary and the deterministic online algorithm place these blocks into the cache. The next request is for block $k + 1$. In order to make room in the cache for block $k + 1$, the online algorithm evicts some block b_1 from the cache. The adversary, knowing that the online algorithm has just evicted block b_1, makes the next request be for b_1, so that the online algorithm must evict some other block b_2 to clear room in the cache for b_1. As you might have guessed, the adversary makes the next request be for block b_2, so that the online algorithm evicts some other block b_3 to make room for b_2. The online algorithm and the adversary continue in this manner. The online algorithm incurs a cache miss on every request and therefore incurs n cache misses over the n requests.

Now let's consider an optimal offline algorithm, which knows the future. As discussed in Section 15.4, this algorithm is known as furthest-in-future, and it always evicts the block whose next request is furthest in the future. Since there are only $k + 1$ unique blocks, when furthest-in-future evicts a block, we know that it will not be accessed during at least the next k requests. Thus, after the first k cache misses, the optimal algorithm incurs a cache miss at most once every k requests. Therefore, the number of cache misses over n requests is at most $k + n/k$.

Since the deterministic online algorithm incurs n cache misses and the optimal offline algorithm incurs at most $k + n/k$ cache misses, the competitive ratio is at least

$$\frac{n}{k + n/k} = \frac{nk}{n + k^2} .$$

For $n \geq k^2$, the above expression is at least

$$\frac{nk}{n + k^2} \geq \frac{nk}{2n} = \frac{k}{2} .$$

Thus, for sufficiently long request sequences, we have shown the following:

Theorem 27.4

Any deterministic online algorithm for caching with a cache size of k has competitive ratio $\Omega(k)$. ■

Although we can analyze the common caching strategies from the point of view of competitive analysis, the results are somewhat unsatisfying. Yes, we can distinguish between algorithms with a competitive ratio of $\Theta(k)$ and those with unbounded competitive ratios. In the end, however, all of these competitive ratios are rather high. The online algorithms we have seen so far are deterministic, and it is this property that the adversary is able to exploit.

27.3.2 Randomized caching algorithms

If we don't limit ourselves to deterministic online algorithms, we can use randomization to develop an online caching algorithm with a significantly smaller competitive ratio. Before describing the algorithm, let's discuss randomization in online algorithms in general. Recall that we analyze online algorithms with respect to an adversary who knows the online algorithm and can design requests knowing the decisions made by the online algorithm. With randomization, we must ask whether the adversary also knows the random choices made by the online algorithm. An adversary who does not know the random choices is ***oblivious***, and an adversary who knows the random choices is ***nonoblivious***. Ideally, we prefer to design algorithms against a nonoblivious adversary, as this adversary is stronger than an oblivious one. Unfortunately, a nonoblivious adversary mitigates much of the power of randomness, as an adversary who knows the outcome of random choices typically can act as if the online algorithm is deterministic. The oblivious adversary, on the other hand, does not know the random choices of the online algorithm, and that is the adversary we typically use.

As a simple illustration of the difference between an oblivious and nonoblivious adversary, imagine that you are flipping a fair coin n times, and the adversary wants to know how many heads you flipped. A nonoblivious adversary knows, after each flip, whether the coin came up heads or tails, and hence knows how many heads you flipped. An oblivious adversary, on the other hand, knows only that you are flipping a fair coin n times. The oblivious adversary, therefore, can reason that the number of heads follows a binomial distribution, so that the expected number of heads is $n/2$ (by equation (C.41) on page 1199) and the variance is $n/4$ (by equation (C.44) on page 1200). But the oblivious adversary has no way of knowing exactly how many heads you actually flipped.

Let's return to caching. We'll start with a deterministic algorithm and then randomize it. The algorithm we'll use is an approximation of LRU called MARKING. Rather than "least recently used," think of MARKING as simply "recently used." MARKING maintains a 1-bit attribute *mark* for each block in the cache. Initially, all blocks in the cache are unmarked. When a block is requested, if it is already in the cache, it is marked. If the request is a cache miss, MARKING checks to see whether there are any unmarked blocks in the cache. If all blocks are marked, then they are all changed to unmarked. Now, regardless of whether all blocks in the cache were marked when the request occurred, there is at least one unmarked block in the cache, and so an arbitrary unmarked block is evicted, and the requested block is placed into the cache and marked.

How should the block to evict from among the unmarked blocks in the cache be chosen? The procedure RANDOMIZED-MARKING on the next page shows the

process when the block is chosen randomly. The procedure takes as input a block b being requested.

```
RANDOMIZED-MARKING(b)
1  if block b resides in the cache,
2       b.mark = 1
3  else
4       if all blocks b' in the cache have b'.mark = 1
5            unmark all blocks b' in the cache, setting b'.mark = 0
6       select an unmarked block u with u.mark = 0 uniformly at random
7       evict block u
8       place block b into the cache
9       b.mark = 1
```

For the purpose of analysis, we say that a new epoch begins immediately after each time line 5 executes. An epoch starts with no marked blocks in the cache. The first time a block is requested during an epoch, the number of marked blocks increases by 1, and any subsequent requests to that block do not change the number of marked blocks. Therefore, the number of marked blocks monotonically increases within an epoch. Under this view, epochs are the same as in the proof of Theorem 27.3: with a cache that holds k blocks, an epoch comprises requests for k distinct blocks (possibly fewer for the final epoch), and the next epoch begins upon a request for a block not in those k.

Because we are going to analyze a randomized algorithm, we will compute the expected competitive ratio. Recall that for an input I, we denote the solution value of an online algorithm A by $A(I)$ and the solution value of an optimal algorithm F by $F(I)$. Online algorithm A has an ***expected competitive ratio*** c if for all inputs I, we have

$$\mathrm{E}[A(I)] \le cF(I) , \tag{27.12}$$

where the expectation is taken over the random choices made by A.

Although the deterministic MARKING algorithm has a competitive ratio of $\Theta(k)$ (Theorem 27.4 provides the lower bound and see Exercise 27.3-4 for the upper bound), RANDOMIZED-MARKING has a much smaller expected competitive ratio, namely $O(\lg k)$. The key to the improved competitive ratio is that the adversary cannot always make a request for a block that is not in the cache, since an oblivious adversary does not know which blocks are in the cache.

Theorem 27.5
RANDOMIZED-MARKING has an expected competitive ratio of $O(\lg k)$ for the online caching problem with n requests and a cache of size k, against an oblivious adversary.

Before proving Theorem 27.5, we prove a basic probabilistic fact.

Lemma 27.6
Suppose that a bag contains $x + y$ balls: $x - 1$ blue balls, y white balls, and 1 red ball. You repeatedly choose a ball at random and remove it from the bag until you have chosen a total of m balls that are either blue or red, where $m \le x$. You set aside each white ball you choose. Then, one of the balls chosen is the red ball with probability m/x.

Proof Choosing a white ball does not affect how many blue or red balls are chosen in any way. Therefore, we can continue the analysis as if there were no white balls and the bag contains just $x - 1$ blue balls and 1 red ball.

Let A be the event that the red ball is not chosen, and let A_i be the event that the ith draw does not choose the red ball. By equation (C.22) on page 1190, we have

$$\begin{aligned}\Pr\{A\} &= \Pr\{A_1 \cap A_2 \cap \cdots \cap A_m\} \\ &= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \\ &\quad \Pr\{A_m \mid A_1 \cap A_2 \cap \cdots \cap A_{m-1}\} \ . \end{aligned} \tag{27.13}$$

The probability $\Pr\{A_1\}$ that the first ball is blue equals $(x - 1)/x$, since initially there are $x - 1$ blue balls and 1 red ball. More generally, we have

$$\Pr\{A_i \mid A_1 \cap \cdots \cap A_{i-1}\} = \frac{x - i}{x - i + 1} \ , \tag{27.14}$$

since the ith draw is from $x - i$ blue balls and 1 red ball. Equations (27.13) and (27.14) give

$$\Pr\{A\} = \left(\frac{x-1}{x}\right)\left(\frac{x-2}{x-1}\right)\left(\frac{x-3}{x-2}\right)\cdots\left(\frac{x-m+1}{x-m+2}\right)\left(\frac{x-m}{x-m+1}\right) \ . \tag{27.15}$$

The right-hand side of equation (27.15) is a telescoping product, similar to the telescoping series in equation (A.12) on page 1143. The numerator of one term equals the denominator of the next, so that everything except the first denominator and last numerator cancel, and we obtain $\Pr\{A\} = (x - m)/x$. Since we actually want to compute $\Pr\{\bar{A}\} = 1 - \Pr\{A\}$, that is, the probability that the red ball *is* chosen, we get $\Pr\{\bar{A}\} = 1 - (x - m)/x = m/x$. ∎

Now we can prove Theorem 27.5.

Proof We'll analyze RANDOMIZED-MARKING one epoch at a time. Within epoch i, any request for a block b that is not the first request for block b in epoch i must result in a cache hit, since after the first request in epoch i, block b resides in the cache and is marked, so that it cannot be evicted during the epoch. Therefore, since we are counting cache misses, we'll consider only the first request for each block within each epoch, disregarding all other requests.

We can classify the requests in an epoch as either old or new. If block b resides in the cache at the start of epoch i, each request for block b during epoch i is an ***old request***. Old requests in epoch i are for blocks requested in epoch $i-1$. If a request in epoch i is not old, it is a ***new request***, and it is for a block not requested in epoch $i-1$. All requests in epoch 1 are new. For example, let's look again at the request sequence in example (27.11):

$$1,2,1,5 \qquad 4,\ 4,\ 1,\ 2,\ 4,\ 2 \qquad 3,4,5 \qquad 2,\ 2,\ 1,\ 2,\ 2\,.$$

Since we can disregard all requests for a block within an epoch other than the first request, to analyze the cache behavior, we can view this request sequence as just

$$1,2,5 \qquad 4,1,2 \qquad 3,4,5 \qquad 2,\ 1\,.$$

All three requests in epoch 1 are new. In epoch 2, the requests for blocks 1 and 2 are old, but the request for block 4 is new. In epoch 3, the request for block 4 is old, and the requests for blocks 3 and 5 are new. Both requests in epoch 4 are new.

Within an epoch, each new request must cause a cache miss since, by definition, the block is not already in the cache. An old request, on the other hand, may or may not cause a cache miss. The old block is in the cache at the beginning of the epoch, but other requests might cause it to be evicted. Returning to our example, in epoch 2, the request for block 4 must cause a cache miss, as this request is new. The request for block 1, which is old, may or may not cause a cache miss. If block 1 was evicted when block 4 was requested, then a cache miss occurs and block 1 must be brought back into the cache. If instead block 1 was not evicted when block 4 was requested, then the request for block 1 results in a cache hit. The request for block 2 could incur a cache miss under two scenarios. One is if block 2 was evicted when block 4 was requested. The other is if block 1 was evicted when block 4 was requested, and then block 2 was evicted when block 1 was requested. We see that, within an epoch, each ensuing old request has an increasing chance of causing a cache miss.

Because we consider only the first request for each block within an epoch, we assume that each epoch contains exactly k requests, and each request within an epoch is for a unique block. (The last epoch might contain fewer than k requests. If it does, just add dummy requests to fill it out to k requests.) In epoch i, denote the number of new requests by $r_i \geq 1$ (an epoch must contain at least one new

request), so that the number of old requests is $k - r_i$. As mentioned above, a new request always incurs a cache miss.

Let us now focus on an arbitrary epoch i to obtain a bound on the expected number of cache misses within that epoch. In particular, let's think about the jth old request within the epoch, where $1 \le j < k$. Denote by b_{ij} the block requested in the jth old request of epoch i, and denote by n_{ij} and o_{ij} the number of new and old requests, respectively, that occur within epoch i but before the jth old request. Because $j - 1$ old requests occur before the jth old request, we have $o_{ij} = j - 1$. We will show that the probability of a cache miss upon the jth old request is $n_{ij}/(k - o_{ij})$, or $n_{ij}/(k - j + 1)$.

Start by considering the first old request, for block $b_{i,1}$. What is the probability that this request causes a cache miss? It causes a cache miss precisely when one of the $n_{i,1}$ previous requests resulted in $b_{i,1}$ being evicted. We can determine the probability that $b_{i,1}$ was chosen for eviction by using Lemma 27.6: consider the k blocks in the cache to be k balls, with block $b_{i,1}$ as the red ball, the other $k - 1$ blocks as the $k - 1$ blue balls, and no white balls. Each of the $n_{i,1}$ requests chooses a block to evict with equal probability, corresponding to drawing balls $n_{i,1}$ times. Thus, we can apply Lemma 27.6 with $x = k$, $y = 0$, and $m = n_{i,1}$, deriving the probability of a cache miss upon the first old request as $n_{i,1}/k$, which equals $n_{ij}/(k - j + 1)$ since $j = 1$.

In order to determine the probability of a cache miss for subsequent old requests, we'll need an additional observation. Let's consider the second old request, which is for block $b_{i,2}$. This request causes a cache miss precisely when one of the previous requests evicts $b_{i,2}$. Let's consider two cases, based on the request for $b_{i,1}$. In the first case, suppose that the request for $b_{i,1}$ did not cause an eviction, because $b_{i,1}$ was already in the cache. Then, the only way that $b_{i,2}$ could have been evicted is by one of the $n_{i,2}$ new requests that precedes it. What is the probability that this eviction happens? There are $n_{i,2}$ chances for $b_{i,2}$ to be evicted, but we also know that there is one block in the cache, namely $b_{i,1}$, that is not evicted. Thus, we can again apply Lemma 27.6, but with $b_{i,1}$ as the white ball, $b_{i,2}$ as the red ball, the remaining blocks as the blue balls, and drawing balls $n_{i,2}$ times. Applying Lemma 27.6, with $x = k - 1$, $y = 1$, and $m = n_{i,2}$, we find that the probability of a cache miss is $n_{i,2}/(k - 1)$. In the second case, the request for $b_{i,1}$ does cause an eviction, which can happen only if one of the new requests preceding the request for $b_{i,1}$ evicts $b_{i,1}$. Then, the request for $b_{i,1}$ brings $b_{i,1}$ back into the cache and evicts some other block. In this case, we know that of the new requests, one of them did not result in $b_{i,2}$ being evicted, since $b_{i,1}$ was evicted. Therefore, $n_{i,2} - 1$ new requests could evict $b_{i,2}$, as could the request for $b_{i,1}$, so that the number of requests that could evict $b_{i,2}$ is $n_{i,2}$. Each such request evicts a block chosen from among $k - 1$ blocks, since the request that resulted in evicting $b_{i,1}$ did not also cause $b_{i,2}$ to be evicted. Therefore, we can apply Lemma 27.6, with $x = k - 1$,

$y = 1$, and $m = n_{i,2}$, and get that the probability of a miss is $n_{i,2}/(k-1)$. In both cases the probability is the same, and it equals $n_{ij}/(k-j+1)$ since $j = 2$.

More generally, o_{ij} old requests occur before the jth old request. Each of these prior old requests either caused an eviction or did not. For those that caused an eviction, it is because they were evicted by a previous request, and for those that did not cause an eviction, it is because they were not evicted by any previous request. In either case, we can decrease the number of blocks that the random process is choosing from by 1 for each old request, and thus o_{ij} requests cannot cause b_{ij} to be evicted. Therefore, we can use Lemma 27.6 to determine the probability that b_{ij} was evicted by a previous request, with $x = k - o_{ij}$, $y = o_{ij}$ and $m = n_{ij}$. Thus, we have proven our claim that the probability of a cache miss on the jth request for an old block is $n_{ij}/(k-o_{ij})$, or $n_{ij}/(k-j+1)$. Since $n_{ij} \le r_i$ (recall that r_i is the number of new requests during epoch i), we have an upper bound of $r_i/(k-j+1)$ on the probability that the jth old request incurs a cache miss.

We can now compute the expected number of misses during epoch i using indicator random variables, as introduced in Section 5.2. We define indicator random variables

$$\begin{aligned} Y_{ij} &= \mathrm{I}\{\text{the } j\text{th old request in epoch } i \text{ incurs a cache miss}\}\ , \\ Z_{ij} &= \mathrm{I}\{\text{the } j\text{th new request in epoch } i \text{ incurs a cache miss}\}\ . \end{aligned}$$

We have $Z_{ij} = 1$ for $j = 1, 2, \ldots, r_i$, since every new request results in a cache miss. Let X_i be the random variable denoting the number of cache misses during epoch i, so that

$$X_i = \sum_{j=1}^{k-r_i} Y_{ij} + \sum_{j=1}^{r_i} Z_{ij}\ ,$$

and so

$$\begin{aligned} \mathrm{E}[X_i] &= \mathrm{E}\left[\sum_{j=1}^{k-r_i} Y_{ij} + \sum_{j=1}^{r_i} Z_{ij}\right] \\ &= \sum_{j=1}^{k-r_i} \mathrm{E}[Y_{ij}] + \sum_{j=1}^{r_i} \mathrm{E}[Z_{ij}] \qquad \text{(by linearity of expectation)} \\ &\le \sum_{j=1}^{k-r_i} \frac{r_i}{k-j+1} + \sum_{j=1}^{r_i} 1 \qquad \text{(by Lemma 5.1 on page 130)} \\ &= r_i \left(\sum_{j=1}^{k-r_i} \frac{1}{k-j+1} + 1\right) \end{aligned}$$

$$\leq r_i \left(\sum_{j=1}^{k-1} \frac{1}{k-j+1} + 1 \right)$$
$$= r_i H_k \qquad \text{(by equation (A.8) on page 1142)} , \tag{27.16}$$

where H_k is the kth harmonic number.

To compute the expected total number of cache misses, we sum over all epochs. Let p denote the number of epochs and X be the random variable denoting the number of cache misses. Then, we have $X = \sum_{i=1}^{p} X_i$, so that

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{p} X_i \right] \\ &= \sum_{i=1}^{p} \mathrm{E}[X_i] \qquad \text{(by linearity of expectation)} \\ &\leq \sum_{i=1}^{p} r_i H_k \qquad \text{(by inequality (27.16))} \\ &= H_k \sum_{i=1}^{p} r_i \ . \end{aligned} \tag{27.17}$$

To complete the analysis, we need to understand the behavior of the optimal offline algorithm. It could make a completely different set of decisions from those made by RANDOMIZED-MARKING, and at any point its cache may look nothing like the cache of the randomized algorithm. Yet, we want to relate the number of cache misses of the optimal offline algorithm to the value in inequality (27.17), in order to have a competitive ratio that does not depend on $\sum_{i=1}^{p} r_i$. Focusing on individual epochs won't suffice. At the beginning of any epoch, the offline algorithm might have loaded the cache with exactly the blocks that will be requested in that epoch. Therefore, we cannot take any one epoch in isolation and claim that an offline algorithm must suffer any cache misses during that epoch.

If we consider two consecutive epochs, however, we can better analyze the optimal offline algorithm. Consider two consecutive epochs, $i-1$ and i. Each contains k requests for k different blocks. (Recall our assumption that all requests are first requests in an epoch.) Epoch i contains r_i requests for new blocks, that is, blocks that were not requested during epoch $i-1$. Therefore, the number of distinct requests during epochs $i-1$ and i is exactly $k+r_i$. No matter what the cache contents were at the beginning of epoch $i-1$, after $k+r_i$ distinct requests, there must be at least r_i cache misses. There could be more, but there is no way to have fewer. Letting m_i denote the number of cache misses of the offline algorithm during epoch i, we have just argued that

$$m_{i-1} + m_i \geq r_i \ . \tag{27.18}$$

The total number of cache misses of the offline algorithm is

$$\begin{aligned}
\sum_{i=1}^{p} m_i &= \frac{1}{2}\sum_{i=1}^{p} 2m_i \\
&= \frac{1}{2}\left(m_1 + \sum_{i=2}^{p}(m_{i-1} + m_i) + m_p\right) \\
&\geq \frac{1}{2}\left(m_1 + \sum_{i=2}^{p}(m_{i-1} + m_i)\right) \\
&\geq \frac{1}{2}\left(m_1 + \sum_{i=2}^{p} r_i\right) && \text{(by inequality (27.18))} \\
&= \frac{1}{2}\sum_{i=1}^{p} r_i && \text{(because } m_1 = r_1\text{)} .
\end{aligned}$$

The justification $m_1 = r_1$ for the last equality follows because, by our assumptions, the cache starts out empty and every request incurs a cache miss in the first epoch, even for the optimal offline adversary.

To conclude the analysis, because we have an upper bound of $H_k \sum_{i=1}^{p} r_i$ on the expected number of cache misses for RANDOMIZED-MARKING and a lower bound of $\frac{1}{2}\sum_{i=1}^{p} r_i$ on the number of cache misses for the optimal offline algorithm, the expected competitive ratio is at most

$$\begin{aligned}
\frac{H_k \sum_{i=1}^{p} r_i}{\frac{1}{2}\sum_{i=1}^{p} r_i} &= 2H_k \\
&= 2\ln k + O(1) \quad \text{(by equation (A.9) on page 1142)} \\
&= O(\lg k) .
\end{aligned}$$

■

Exercises

27.3-1

For the cache sequence (27.10), show the contents of the cache after each request and count the number of cache misses. How many misses does each epoch incur?

27.3-2

Show that LFU has a competitive ratio of $\Theta(n/k)$ for the online caching problem with n requests and a cache of size k.

27.3-3

Show that FIFO has a competitive ratio of $O(k)$ for the online caching problem with n requests and a cache of size k.

27.3-4

Show that the deterministic MARKING algorithm has a competitive ratio of $O(k)$ for the online caching problem with n requests and a cache of size k.

27.3-5

Theorem 27.4 shows that any deterministic online algorithm for caching has a competitive ratio of $\Omega(k)$, where k is the cache size. One way in which an algorithm might be able to perform better is to have some ability to know what the next few requests will be. We say that an algorithm is ***l-lookahead*** if it has the ability to look ahead at the next l requests. Prove that for every constant $l \geq 0$ and every cache size $k \geq 1$, every deterministic l-lookahead algorithm has competitive ratio $\Omega(k)$.

Problems

27-1 Cow-path problem

The Appalachian Trail (AT) is a marked hiking trail in the eastern United States extending between Springer Mountain in Georgia and Mount Katahdin in Maine. The trail is about 2,190 miles long. You decide that you are going to hike the AT from Georgia to Maine and back. You plan to learn more about algorithms while on the trail, and so you bring along your copy of *Introduction to Algorithms* in your backpack.[2] You have already read through this chapter before starting out. Because the beauty of the trail distracts you, you forget about reading this book until you have reached Maine and hiked halfway back to Georgia. At that point, you decide that you have already seen the trail and want to continue reading the rest of the book, starting with Chapter 28. Unfortunately, you find that the book is no longer in your pack. You must have left it somewhere along the trail, but you don't know where. It could be anywhere between Georgia and Maine. You want to find the book, but now that you have learned something about online algorithms, you want your algorithm for finding it to have a good competitive ratio. That is, no matter where the book is, if its distance from you is x miles away, you would like to be sure that you do not walk more than cx miles to find it, for some constant c. You do not know x, though you may assume that $x \geq 1$.[3]

[2] This book is heavy. We do not recommend that you carry it on a long hike.

[3] In case you're wondering what this problem has to do with cows, some papers about it frame the problem as a cow looking for a field in which to graze.

What algorithm should you use, and what constant c can you prove bounds the total distance cx that you would have to walk? Your algorithm should work for a trail of any length, not just the 2,190-mile-long AT.

27-2 Online scheduling to minimize average completion time

Problem 15-2 discusses scheduling to minimize average completion time on one machine, without release times and preemption and with release times and preemption. Now you will develop an online algorithm for nonpreemptively scheduling a set of tasks with release times. Suppose you are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i has ***release time*** r_i, before which it cannot start, and requires p_i units of processing time to complete once it has started. You have one computer on which to run the tasks. Tasks cannot be ***preempted***, which is to say that once started, a task must run to completion without interruption. (See Problem 15-2 on page 446 for a more detailed description of this problem.) Given a schedule, let C_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to find a schedule that minimizes the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} C_i$.

In the online version of this problem, you learn about task i only when it arrives at its release time r_i, and at that point, you know its processing time p_i. The offline version of this problem is NP-hard (see Chapter 34), but you will develop a 2-competitive online algorithm.

a. Show that, if there are release times, scheduling by shortest processing time (when the machine becomes idle, start the already released task with the smallest processing time that has not yet run) is not d-competitive for any constant d.

In order to develop an online algorithm, consider the preemptive version of this problem, which is discussed in Problem 15-2(b). One way to schedule is to run the tasks according to the shortest remaining processing time (SRPT) order. That is, at any point, the machine is running the available task with the smallest amount of remaining processing time.

b. Explain how to run SRPT as an online algorithm.

c. Suppose that you run SRPT and obtain completion times $C_1^P, \ldots, C_n^P$. Show that

$$\sum_{i=1}^{n} C_i^P \le \sum_{i=1}^{n} C_i^* ,$$

where the C_i^* are the completion times in an optimal nonpreemptive schedule.

Consider the (offline) algorithm COMPLETION-TIME-SCHEDULE.

```
COMPLETION-TIME-SCHEDULE(S)
1  compute an optimal schedule for the preemptive version of the problem
2  renumber the tasks so that the completion times in the optimal
       preemptive schedule are ordered by their completion times
       C_1^P < C_2^P < ··· < C_n^P in SRPT order
3  greedily schedule the tasks nonpreemptively in the renumbered
       order a_1, ..., a_n
4  let C_1, ..., C_n be the completion times of renumbered tasks a_1, ..., a_n
       in this nonpreemptive schedule
5  return C_1, ..., C_n
```

d. Prove that $C_i^P \geq \max\left\{\sum_{j=1}^{i} p_j, \max\{r_j : j \leq i\}\right\}$ for $i = 1, \ldots, n$.

e. Prove that $C_i \leq \max\{r_j : j \leq i\} + \sum_{j=1}^{i} p_j$ for $i = 1, \ldots, n$.

f. Algorithm COMPLETION-TIME-SCHEDULE is an offline algorithm. Explain how to modify it to produce an online algorithm.

g. Combine parts (c)–(f) to show that the online version of COMPLETION-TIME-SCHEDULE is 2-competitive.

Chapter notes

Online algorithms are widely used in many domains. Some good overviews include the textbook by Borodin and El-Yaniv [68], the collection of surveys edited by Fiat and Woeginger [142], and the survey by Albers [14].

The move-to-front heuristic from Section 27.2 was analyzed by Sleator and Tarjan [416, 417] as part of their early work on amortized analysis. This rule works quite well in practice.

Competitive analysis of online caching also originated with Sleator and Tarjan [417]. The randomized marking algorithm was proposed and analyzed by Fiat et al. [141]. Young [464] surveys online caching and paging algorithms, and Buchbinder and Naor [76] survey primal-dual online algorithms.

Specific types of online algorithms are described using other names. ***Dynamic graph algorithms*** are online algorithms on graphs, where at each step a vertex or edge undergoes modification. Typically a vertex or edge is either inserted or

deleted, or some associated property, such as edge weight, changes. Some graph problems need to be solved again after each change to the graph, and a good dynamic graph algorithm will not need to solve from scratch. For example, edges are inserted and deleted, and after each change to the graph, the minimum spanning tree is recomputed. Exercise 21.2-8 asks such a question. Similar questions can be asked for other graph algorithms, such as shortest paths, connectivity, or matching. The first paper in this field is credited to Even and Shiloach [138], who study how to maintain a shortest-path tree as edges are being deleted from a graph. Since then hundreds of papers have been published. Demetrescu et al. [110] survey early developments in dynamic graph algorithms.

For massive data sets, the input data might be too large to store. ***Streaming algorithms*** model this situation by requiring the memory used by an algorithm to be significantly smaller than the input size. For example, you may have a graph with n vertices and m edges with $m \gg n$, but the memory allowed may be only $O(n)$. Or you may have n numbers, but the memory allowed may only be $O(\lg n)$ or $O(\sqrt{n})$. A streaming algorithm is measured by the number of passes made over the data in addition to the running time of the algorithm. McGregor [322] surveys streaming algorithms for graphs and Muthukrishnan [341] surveys general streaming algorithms.

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 to 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***. Because actual computers have limits to how precisely they can represent floating-point numbers, round-off errors in numerical computations may become amplified over the course of a computation, leading to incorrect results. Such computations are called ***numerically unstable***. Although we'll briefly consider numerical stability on occasion, we won't focus on it in this chapter. We refer you to the excellent book by Higham [216] for a thorough discussion of stability issues.

28.1 Solving systems of linear equations

Numerous applications need to solve sets of simultaneous linear equations. A linear system can be cast 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.

The process starts 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.

Next, 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,

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

This section focuses on the case in which A is nonsingular or, equivalently (by Theorem D.1 on page 1220), the rank of A equals 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's return to the problem of solving the system $Ax = b$ of n equations in n unknowns. One option is to 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'll show that every nonsingular matrix A possesses such a decomposition.

Computing an LUP decomposition for the matrix A has the advantage that linear systems can be efficiently solved when they are triangular, as is the case for both matrices L and U. If you have an LUP decomposition for A, you can solve equation (28.2), $Ax = b$, by solving only triangular linear systems, as follows. Multiply both sides of $Ax = b$ by P, yielding the equivalent equation $PAx = Pb$. By Exercise D.1-4 on page 1219, multiplying both sides by a permutation matrix amounts to permuting the equations (28.1). By the decomposition (28.4), substituting LU for PA gives

$$LUx = Pb .$$

You can now solve this equation by solving two triangular linear systems. Define $y = Ux$, where x is the desired solution vector. First, 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, solve the upper-triangular system

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

for the unknown x by a method called "back substitution." Why does this process solve $Ax = b$? Because the permutation matrix P is invertible (see Exercise D.2-3 on page 1223), 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 that satisfies $Ux = y$ is the solution to $Ax = b$:

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

The 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. An array $\pi[1:n]$ provides a more compact format to represent the permutation P than an $n \times n$ matrix that is mostly 0s. 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, the matrix equation $Ly = Pb$ is equivalent to the n equations

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

The first equation gives $y_1 = b_{\pi[1]}$ directly. Knowing the value of y_1, you can substitute it into the second equation, yielding

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

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

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

Once you've solved for y, you can solve for x in equation (28.6) using ***back substitution***, which is similar to forward substitution. This time, you 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, the matrix equation $Ux = y$ is equivalent to the n equations

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

Thus, you 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 on the next page solves for x by combining forward and back substitution. The permutation matrix P is represented by the array π. The procedure first solves for y using forward substitution in lines 2–3, and then it solves for x using backward substitution in lines 4–5. 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 $Ax = b$, where

$$A = \begin{pmatrix} 1 & 2 & 0 \\ 3 & 4 & 4 \\ 5 & 6 & 3 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 3 \\ 7 \\ 8 \end{pmatrix} ,$$

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

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

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 0.2 & 1 & 0 \\ 0.6 & 0.5 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 5 & 6 & 3 \\ 0 & 0.8 & -0.6 \\ 0 & 0 & 2.5 \end{pmatrix}, \quad \text{and} \quad P = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

(You might want to verify that $PA = LU$.) Using forward substitution, 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. Then, using back substitution, 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

Given an LUP decomposition for a nonsingular matrix A, you can use forward and back substitution to solve the system $Ax = b$ of linear equations. Now let's see

how to efficiently compute an LUP decomposition for A. We start with the simpler case in which A is an $n \times n$ nonsingular matrix and P is absent (or, equivalently, $P = I_n$, the $n \times n$ identity matrix), so that $A = LU$. We call the two matrices L and U an ***LU decomposition*** of A.

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

To implement this strategy, let's start with a recursive formulation. The input is an $n \times n$ nonsingular matrix A. If $n = 1$, then nothing needs to be done: just choose $L = I_1$ and $U = A$. For $n > 1$, break A into four parts:

$$\begin{aligned} A &= \left(\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right) \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix}, \end{aligned} \tag{28.8}$$

where $v = (a_{21}, a_{31}, \ldots, a_{n1})$ is a column $(n-1)$-vector, $w^{\mathrm{T}} = (a_{12}, a_{13}, \ldots, a_{1n})^{\mathrm{T}}$ 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), factor A as

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & A' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix}. \end{aligned} \tag{28.9}$$

The 0s in the first and second matrices of equation (28.9) are row and column $(n-1)$-vectors, respectively. The term vw^{T}/a_{11} is an $(n-1) \times (n-1)$ matrix formed by taking the outer product of v and w and dividing each element of the result by a_{11}. Thus it conforms in size to the matrix A' from which it is subtracted. The resulting $(n-1) \times (n-1)$ matrix

$$A' - vw^{\mathrm{T}}/a_{11} \tag{28.10}$$

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 on page 1223 to equation (28.9), 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, it, too, has an LU decomposition, which we can find recursively. Let's say that

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

where L' is unit lower-triangular and U' is upper-triangular. The LU decomposition of A is then $A = LU$, with

$$L = \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \text{ and } U = \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} ,$$

as shown by

$$\begin{aligned} A &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & A' - vw^{\mathrm{T}}/a_{11} \end{pmatrix} \quad \text{(by equation (28.9))} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & L'U' \end{pmatrix} \\ &= \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ v & vw^{\mathrm{T}}/a_{11} + L'U' \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ v/a_{11} & L' \end{pmatrix} \begin{pmatrix} a_{11} & w^{\mathrm{T}} \\ 0 & U' \end{pmatrix} \\ &= LU. \end{aligned}$$

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 the next step of the recursion will divide by it. The denominators in each step of LU decomposition are called ***pivots***, and they occupy the diagonal elements of the matrix U. The permutation matrix P included in LUP decomposition provides a way to avoid dividing by 0, as we'll see below. Using permutations to avoid division by 0 (or by small numbers, which can contribute to numerical instability), is called ***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 to avoid dividing by 0 in the recursive strategy outlined above. We will prove this result, as well as several others, in Section 28.3.

The pseudocode in the procedure LU-DECOMPOSITION 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 7-5 on page 202.) The procedure initializes the matrix U with 0s below the diagonal and matrix L with 1s on its diagonal and 0s above the diagonal. Each iteration works on a square submatrix, using its upper leftmost element as the pivot to compute the v and w vectors and the Schur complement, which becomes the square submatrix worked on by the next iteration.

```
LU-DECOMPOSITION(A, n)
 1  let L and U be new n × n matrices
 2  initialize U with 0s below the diagonal
 3  initialize L with 1s on the diagonal and 0s above the diagonal
 4  for k = 1 to n
 5      u_kk = a_kk
 6      for i = k + 1 to n
 7          l_ik = a_ik / a_kk          // a_ik holds v_i
 8          u_ki = a_ki                 // a_ki holds w_i
 9      for i = k + 1 to n              // compute the Schur complement ...
10          for j = k + 1 to n
11              a_ij = a_ij − l_ik u_kj // ... and store it back into A
12  return L and U
```

Each recursive step in the description above takes place in one iteration of the outer **for** loop of lines 4–11. Within this loop, line 5 determines the pivot to be $u_{kk} = a_{kk}$. The **for** loop in lines 6–8 (which does not execute when $k = n$) uses the v and w vectors to update L and U. Line 7 determines the below-diagonal elements of L, storing v_i/a_{kk} in l_{ik}, and line 8 computes the above-diagonal elements of U, storing w_i in u_{ki}. Finally, lines 9–11 compute the elements of the Schur complement and store them back into the matrix A. (There is no need to divide by a_{kk} in line 11 because that already happened when line 7 computed l_{ik}.) Because line 11 is triply nested, LU-DECOMPOSITION runs in $\Theta(n^3)$ time.

Figure 28.1 illustrates the operation of LU-DECOMPOSITION. It shows a standard optimization of the procedure that stores the significant elements of L and U in place in the matrix A. Each element a_{ij} corresponds to either l_{ij} (if $i > j$) or u_{ij} (if $i \leq j$), so that the matrix A holds both L and U when the procedure terminates. To obtain the pseudocode for this optimization from the pseudocode for the LU-DECOMPOSITION procedure, just replace each reference to l or u by a. You can verify that this transformation preserves correctness.

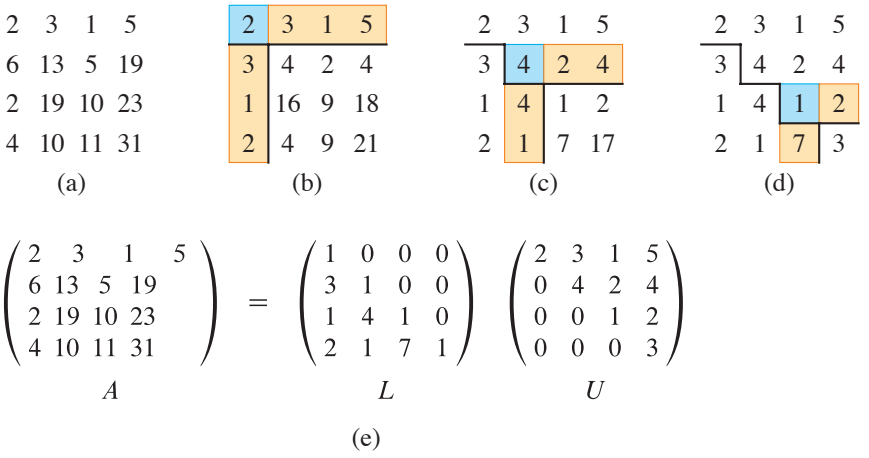


Figure 28.1 The operation of LU-DECOMPOSITION. **(a)** The matrix A. **(b)** The result of the first iteration of the outer **for** loop of lines 4–11. The element $a_{11} = 2$ highlighted in blue is the pivot, the tan column is v/a_{11}, and the tan row is w^{T}. The elements of U computed thus far 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)** The result of the next iteration of the outer **for** loop, on the Schur complement matrix from part (b). The element $a_{22} = 4$ highlighted in blue is the pivot, and the tan 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 iteration, the matrix A is factored. The element 3 in the new Schur complement becomes part of U when the recursion terminates.) **(e)** The factorization $A = LU$.

Computing an LUP decomposition

If the diagonal of the matrix given to LU-DECOMPOSITION contains any 0s, then the procedure will attempt to divide by 0, which would cause disaster. Even if the diagonal contains no 0s, but does have numbers with small absolute values, dividing by such numbers can cause numerical instabilities. Therefore, LUP decomposition pivots on entries with the largest absolute values that it can find.

In LUP decomposition, the input is an $n \times n$ nonsingular matrix A, with a goal of finding a permutation matrix P, a unit lower-triangular matrix L, and an upper-triangular matrix U such that $PA = LU$. Before partitioning the matrix A, as LU decomposition does, LUP decomposition moves a nonzero element, say a_{k1}, from somewhere in the first column to the $(1, 1)$ position of the matrix. For the greatest numerical stability, LUP decomposition chooses the element in the first column with the greatest absolute value as a_{k1}. (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 on page 1221.) In order to preserve the set of equations, LUP decomposition exchanges row 1 with row k, which is equivalent to multiplying A by a

permutation matrix Q on the left (Exercise D.1-4 on page 1219). Thus, the analog to equation (28.8) expresses QA as

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

where $v = (a_{21}, a_{31}, \ldots, a_{n1})$, except that a_{11} replaces a_{k1}; $w^{\mathrm{T}} = (a_{k2}, a_{k3}, \ldots, a_{kn})^{\mathrm{T}}$; and A' is an $(n-1) \times (n-1)$ matrix. Since $a_{k1} \neq 0$, the analog to equation (28.9) guarantees no division by 0:

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

Just as in LU decomposition, if A is nonsingular, then the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ is nonsingular, too. Therefore, you 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 on page 1219). This definition of P gives

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

which yields 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, both the column vector v/a_{k1} and the Schur complement $A' - vw^{\mathrm{T}}/a_{k1}$ are multiplied by the permutation matrix P'. The procedure LUP-DECOMPOSITION gives the pseudocode for LUP decomposition.

```
LUP-DECOMPOSITION(A, n)
 1  let π[1:n] be a new array
 2  for i = 1 to n
 3      π[i] = i                          // initialize π to the identity permutation
 4  for k = 1 to n
 5      p = 0
 6      for i = k to n                    // find largest absolute value in column k
 7          if |a_ik| > p
 8              p = |a_ik|
 9              k' = i                    // row number of the largest found so far
10      if p == 0
11          error "singular matrix"
12      exchange π[k] with π[k']
13      for i = 1 to n
14          exchange a_ki with a_k'i      // exchange rows k and k'
15      for i = k + 1 to n
16          a_ik = a_ik/a_kk
17          for j = k + 1 to n
18              a_ij = a_ij − a_ik a_kj   // compute L and U in place in A
```

Like LU-DECOMPOSITION, the LUP-DECOMPOSITION procedure replaces the recursion with an iteration loop. As an improvement over a direct implementation of the recursion, the procedure dynamically maintains 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. The LUP-DECOMPOSITION procedure also implements the improvement mentioned earlier, computing 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 2–3 initialize the array π to represent the identity permutation. The outer **for** loop of lines 4–18 implements the recursion, finding an LUP decomposition of

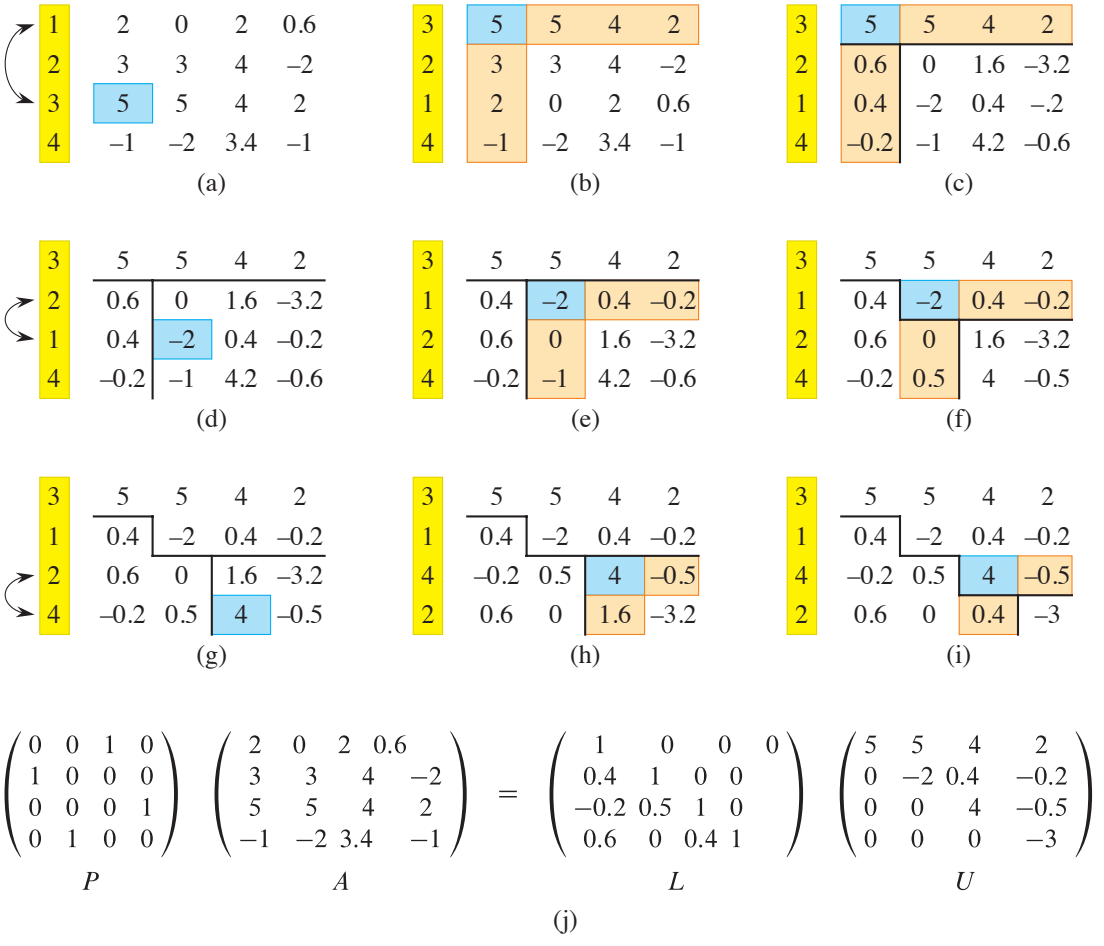


Figure 28.2 The operation of LUP-DECOMPOSITION. **(a)** The input matrix A with the identity permutation of the rows in yellow on the left. The first step of the algorithm determines that the element 5 highlighted in blue in the third row is the pivot for the first column. **(b)** Rows 1 and 3 are swapped and the permutation is updated. The tan 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 $PA = LU$.

the $(n-k+1) \times (n-k+1)$ submatrix whose upper left is in row k and column k. Each time through the outer loop, lines 5–9 determine the element $a_{k'k}$ with the largest absolute value of those in the current first column (column k) of the $(n-k+1) \times (n-k+1)$ submatrix that the procedure is currently working on. If all elements in the current first column are 0, lines 10–11 report that the matrix is singular. To pivot, line 12 exchanges $\pi[k']$ with $\pi[k]$, and lines 13–14 exchange the kth and k'th rows of A, 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 15–18 in much the same way as it is computed by lines 6–11 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 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

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

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, 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 you can use equation (28.3) to solve a system of linear equations by computing a matrix inverse, in practice you are better off using more numerically stable techniques, such as LUP decomposition. Sometimes, however, you really do need to compute a matrix inverse. This section shows how to use LUP decomposition to compute a matrix inverse. It also proves that matrix multiplication and computing the inverse of a matrix are equivalently hard problems, in that (subject to technical conditions) an algorithm for one can solve the other in the same asymptotic running time. Thus, you 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 idea 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 you 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, you can solve an equation of the form $Ax = b$ in $\Theta(n^2)$ time. Since the LUP decomposition depends on A but not b, you can run LUP-SOLVE on a second set of equations of the form $Ax = b'$ in $\Theta(n^2)$ additional time. In general, once you have the LUP decomposition of A, you can solve, in $\Theta(k n^2)$ time, k versions of the equation $Ax = b$ that differ only in the vector b.

Let's think of the equation

$$AX = I_n \,, \tag{28.11}$$

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. You can then solve equation (28.11) for X by using the LUP decomposition for A to solve each equation

$$AX_i = e_i$$

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

Matrix multiplication and matrix inversion

Now let's see how the theoretical speedups obtained for matrix multiplication translate to speedups for matrix inversion. In fact, we'll 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 a nonsingular $n \times n$ matrix can be inverted in $O(M(n))$ time. Moreover, if $I(n)$ denotes the time to invert a nonsingular $n \times n$ matrix, then two $n \times n$ matrices can be multiplied in $O(I(n))$ time. We prove these results as two separate theorems.

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

Proof Let A and B be $n \times n$ matrices. To compute their product $C = AB$, 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 to compute the product AB, just take the upper right $n \times n$ submatrix of D^{-1}.

Constructing matrix D takes $\Theta(n^2)$ time, which is $O(I(n))$ from the assumption that $I(n) = \Omega(n^2)$, and inverting D takes $O(I(3n)) = O(I(n))$ 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 proved in Section 28.3.

Theorem 28.2 (Inversion is no harder than multiplication)
Suppose that two $n \times n$ real matrices can be multiplied in $M(n)$ time, where $M(n) = \Omega(n^2)$ and $M(n)$ satisfies the following two regularity conditions:

1. $M(n+k) = O(M(n))$ for any k in the range $0 \le k < n$, and
2. $M(n/2) \le cM(n)$ for some constant $c < 1/2$.

Then the inverse of any real nonsingular $n \times n$ matrix can be computed in $O(M(n))$ time.

Proof Let A be an $n \times n$ matrix with real-valued entries that is nonsingular. Assume that n is an exact power of 2 (i.e., $n = 2^l$ for some integer l); we'll see at the end of the proof what to do if n is not an exact power of 2.

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

$$A = \begin{pmatrix} B & C^{\mathrm{T}} \\ C & D \end{pmatrix} \text{ and } A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix}. \tag{28.12}$$

Then, if we let

$$S = D - CB^{-1}C^{\mathrm{T}} \tag{28.13}$$

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

$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix} = \begin{pmatrix} B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1} & -B^{-1}C^{\mathrm{T}}S^{-1} \\ -S^{-1}CB^{-1} & S^{-1} \end{pmatrix}, \tag{28.14}$$

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 on page 1223, B^{-1} and S^{-1} are symmetric, so that $(B^{-1})^{\mathrm{T}} = B^{-1}$ and $(S^{-1})^{\mathrm{T}} = S^{-1}$. Therefore, to compute the submatrices

$$\begin{aligned} R &= B^{-1} + B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1}, \\ T &= -B^{-1}C^{\mathrm{T}}S^{-1}, \\ U &= -S^{-1}CB^{-1}, \text{ and} \\ V &= S^{-1} \end{aligned}$$

of A^{-1}, do the following, 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 on page 1219 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.
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 $(S^{-1})^{\mathrm{T}} = S^{-1}$).
7. Compute the matrix product $Z = W^{\mathrm{T}}Y$, which equals $B^{-1}C^{\mathrm{T}}S^{-1}CB^{-1}$.
8. Set $R = B^{-1} + Z$.
9. Set $T = -Y^{\mathrm{T}}$.
10. Set $U = -Y$.
11. Set $V = S^{-1}$.

Thus, to invert an $n \times n$ symmetric positive-definite matrix, invert two $n/2 \times n/2$ matrices in steps 2 and 5; perform four multiplications of $n/2 \times n/2$ matrices in steps 3, 4, 6, and 7; plus incur 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. The running time is given by the recurrence

$$\begin{aligned} I(n) &\le 2I(n/2) + 4M(n/2) + O(n^2) \\ &= 2I(n/2) + \Theta(M(n)) \\ &= O(M(n)) \,. \end{aligned} \tag{28.15}$$

The second line follows from the assumption that $M(n) = \Omega(n^2)$ and from the second regularity condition in the statement of the theorem, which implies that $4M(n/2) < 2M(n)$. Because $M(n) = \Omega(n^2)$, case 3 of the master theorem (Theorem 4.1) applies to the recurrence (28.15), giving the $O(M(n))$ result.

It remains to prove how to 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 on page 1222). 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, to compute A^{-1}, first multiply A^{T} by A to obtain $A^{\mathrm{T}}A$, then invert the symmetric positive-definite matrix $A^{\mathrm{T}}A$ using the above divide-and-conquer algorithm, and finally multiply the result by A^{T}. Each of these three steps takes $O(M(n))$ time, and thus any nonsingular matrix with real entries can be inverted in $O(M(n))$ time.

The above proof assumed that A is an $n \times n$ matrix, where n is an exact power of 2. If n is not an exact power of 2, then let $k < n$ be such that $n + k$ is an exact power of 2, and define the $(n + k) \times (n + k)$ matrix A' as

$$A' = \begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix} .$$

Then the inverse of A' is

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

Apply the method of the proof to A' to compute the inverse of A', and take the first n rows and n columns of the result as the desired answer A^{-1}. The first regularity condition on $M(n)$ ensures that enlarging the matrix in this way increases the running time by at most a constant factor. ■

The proof of Theorem 28.2 suggests how to solve the equation $Ax = b$ by using LU decomposition without pivoting, so long as A is nonsingular. Let $y = A^{\mathrm{T}}b$. Multiply both sides of the equation $Ax = b$ by A^{T}, yielding $(A^{\mathrm{T}}A)x = A^{\mathrm{T}}b = y$. This transformation doesn't affect the solution x, since A^{T} is invertible. Because $A^{\mathrm{T}}A$ is symmetric positive-definite, it can be factored by computing an LU decomposition. Then, use forward and back substitution to solve for x in the equation $(A^{\mathrm{T}}A)x = y$. 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. Show that an $M(n)$-time matrix-multiplication algorithm implies an $O(M(n))$-time LUP-decomposition algorithm. (The LUP decomposition your method produces need not be the same as the result produced by the LUP-DECOMPOSITION procedure.)

28.2-3
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 23.2.) 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.

28.2-4
Does the matrix-inversion algorithm based on Theorem 28.2 work when matrix elements are drawn from the field of integers modulo 2? Explain.

★ ***28.2-5***
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. An $n \times n$ matrix A is ***symmetric positive-definite*** if $A = A^{\mathrm{T}}$ (A is symmetric) and $x^{\mathrm{T}}Ax > 0$ for all n-vectors $x \neq 0$ (A is positive-definite). Symmetric positive-definite matrices are nonsingular, and an LU decomposition on them will not divide by 0. This section proves these and several other important properties of symmetric positive-definite matrices. We'll also see 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 on page 1221, 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 an LU decomposition on a symmetric positive-definite matrix A won't divide 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 Since A is symmetric, each leading submatrix A_k is also symmetric. We'll prove that A_k is positive-definite by 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

$$A = \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \tag{28.16}$$

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 (28.16). Equation (28.10) generalizes to define the ***Schur complement*** S of A with respect to A_k as

$$S = C - BA_k^{-1}B^{\mathrm{T}}\,. \tag{28.17}$$

(By Lemma 28.4, A_k is symmetric and positive-definite, and therefore, A_k^{-1} exists by Lemma 28.3, and S is well defined.) The earlier definition (28.10) of the Schur complement is consistent with equation (28.17) 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 its corollary will help prove that LU decomposition works 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 on page 1223, the product $BA_k^{-1}B^{\mathrm{T}}$ is symmetric. Since C and $BA_k^{-1}B^{\mathrm{T}}$ are symmetric, then by Exercise D.1-1 on page 1219, so is S.

It remains to show that S is positive-definite. Consider the partition of A given in equation (28.16). For any nonzero vector x, we have $x^{\mathrm{T}}Ax > 0$ by the assumption that A is positive-definite. Let the subvectors y and z consist of the first k and last $n - k$ elements in x, respectively, and thus they are compatible with A_k and C, respectively. Because A_k^{-1} exists, we have

$$\begin{aligned}
x^{\mathrm{T}}Ax &= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k & B^{\mathrm{T}} \\ B & C \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \\
&= \begin{pmatrix} y^{\mathrm{T}} & z^{\mathrm{T}} \end{pmatrix} \begin{pmatrix} A_k y + B^{\mathrm{T}} z \\ By + Cz \end{pmatrix} \\
&= y^{\mathrm{T}} A_k y + y^{\mathrm{T}} B^{\mathrm{T}} z + z^{\mathrm{T}} By + z^{\mathrm{T}} Cz \\
&= (y + A_k^{-1}B^{\mathrm{T}}z)^{\mathrm{T}} A_k (y + A_k^{-1}B^{\mathrm{T}}z) + z^{\mathrm{T}}(C - BA_k^{-1}B^{\mathrm{T}})z \,, \qquad (28.18)
\end{aligned}$$

This last equation, which you can verify by multiplying through, 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, pick any nonzero z and then choose $y = -A_k^{-1}B^{\mathrm{T}}z$, which causes the first term in equation (28.18) 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 an $n \times n$ symmetric positive-definite matrix. In fact, we'll prove a stronger result than the statement of the corollary: every pivot is strictly positive. The first pivot is a_{11}. Let e_1 be the length-n unit vector $(1\ 0\ 0\ \cdots\ 0)^{\mathrm{T}}$, so that $a_{11} = e_1^{\mathrm{T}}Ae_1$, which is positive because e_1 is nonzero and A is positive

definite. 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. You are given a set of m data points

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

where you know that the y_i are subject to measurement errors. You wish to determine a function $F(x)$ such that the approximation errors

$$\eta_i = F(x_i) - y_i \tag{28.19}$$

are small for $i = 1, 2, \ldots, m$. The form of the function F depends on the problem at hand. Let's 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 n of summands 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, if you are given m data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, you need 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 $n = m$, you can calculate each y_i *exactly* in equation (28.19). Such a high-degree polynomial 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, you 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 you choose a value of n that is less than m, you end up with an overdetermined set of equations whose solution you wish to approximate. Let's see 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, let's minimize the norm of the error vector η, which gives a ***least-squares solution***, since

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

Because

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

to minimize $\|\eta\|$, differentiate $\|\eta\|^2$ with respect to each c_k and then set the result to 0:

$$\frac{d\,\|\eta\|^2}{dc_k} = \sum_{i=1}^{m} 2 \left(\sum_{j=1}^{n} a_{ij} c_j - y_i \right) a_{ik} = 0 . \tag{28.20}$$

The n equations (28.20) for $k = 1, 2, \dots, n$ are equivalent to the single matrix equation

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

or, equivalently (using Exercise D.1-2 on page 1219), to

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

which implies

$$A^{\mathrm{T}} Ac = A^{\mathrm{T}} y \,. \tag{28.21}$$

In statistics, equation (28.21) 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 on page 1222, $A^{\mathrm{T}}A$ is positive-definite as well. Hence, $(A^{\mathrm{T}}A)^{-1}$ exists, and the solution to equation (28.21) is

$$\begin{aligned} c &= \left((A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\right) y \\ &= A^{+} y \,, \end{aligned} \tag{28.22}$$

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 (28.22) 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 you 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 orange dots in Figure 28.3, and you want to fit these points with a quadratic polynomial

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

Start with the matrix of basis-function values

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

whose pseudoinverse is

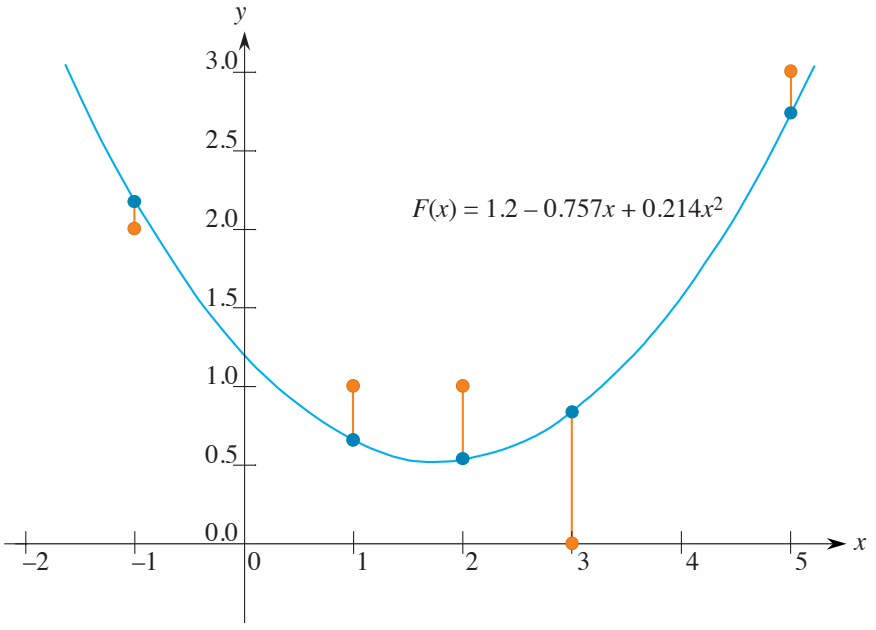


Figure 28.3 The least-squares fit of a quadratic polynomial to the set of five data points $\{(-1, 2), (1, 1), (2, 1), (3, 0), (5, 3)\}$. The orange dots are the data points, and the blue dots are their estimated values predicted by the polynomial $F(x) = 1.2 - 0.757x + 0.214x^2$, the quadratic polynomial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

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

Multiplying y by A^+ gives 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, you would typically solve the normal equation (28.21) 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.)

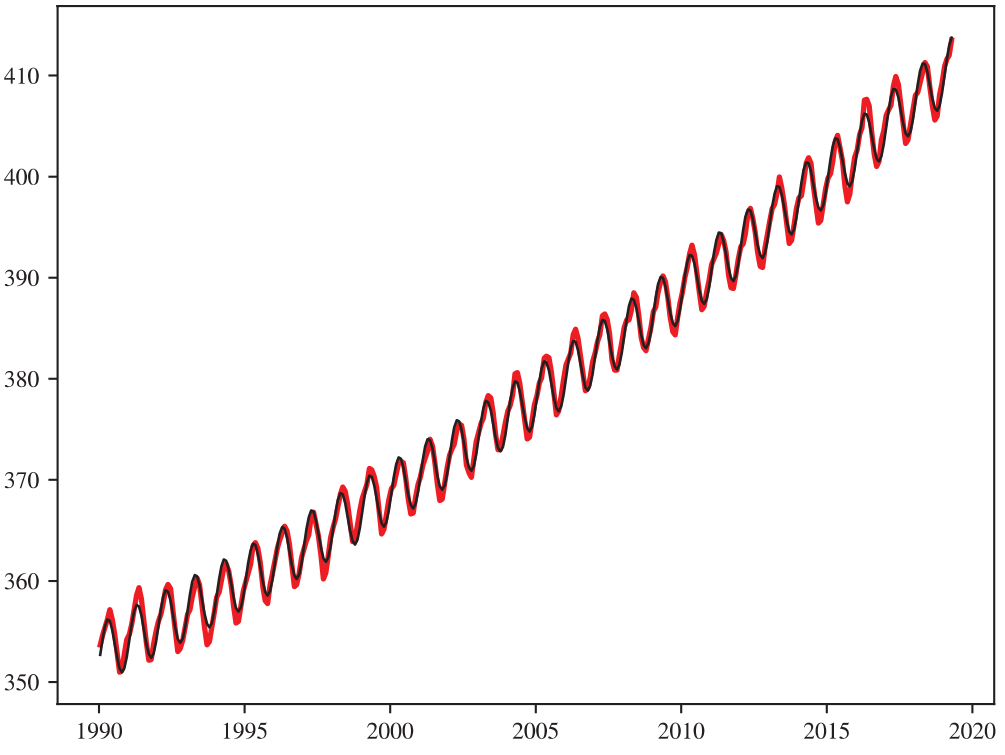


Figure 28.4 A least-squares fit of a curve of the form

$$c_1 + c_2 x + c_3 x^2 + c_4 \sin(2\pi x) + c_5 \cos(2\pi x)$$

for the carbon-dioxide concentrations measured in Mauna Loa, Hawaii from 1990[1] to 2019, where x is the number of years elapsed since 1990. This curve is the famous "Keeling curve," illustrating curve-fitting to nonpolynomial formulas. The sine and cosine terms allow modeling of seasonal variations in CO_2 concentrations. The red curve shows the measured CO_2 concentrations. The best fit, shown in black, has the form

$$352.83 + 1.39x + 0.02x^2 + 2.83 \sin(2\pi x) - 0.94 \cos(2\pi x) \; .$$

We close this section with an example in Figure 28.4, illustrating that a curve can also fit a nonpolynomial function. The curve confirms one aspect of climate change: that carbon dioxide (CO_2) concentrations have steadily increased over a period of 29 years. Linear and quadratic terms model the annual increase, and sine and cosine terms model seasonal variations.

[1] The year in which *Introduction to Algorithms* was first published.

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 $Ax = (1\ 1\ 1\ 1\ 1)^{\mathrm{T}}$ by using forward and back substitution.

c. Find the inverse of A.

d. Show how to solve the equation $Ax = b$ for any $n \times n$ symmetric positive-definite, tridiagonal matrix A and any n-vector 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 to solve the equation $Ax = b$ for any $n \times n$ nonsingular, tridiagonal matrix A and any n-vector 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***. You are given a set $\{(x_i, y_i) : i = 0, 1, \ldots, n\}$ of $n + 1$ point-value pairs, where $x_0 < x_1 < \cdots < x_n$. Your goal is 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, assume that $x_i = i$ for $i = 0, 1, \ldots, n$.

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

$$f(x_i) = f_i(0) = y_i ,$$
$$f(x_{i+1}) = f_i(1) = y_{i+1}$$

for $i = 0, 1, \ldots, n-1$. To ensure that $f(x)$ is sufficiently smooth, also require the first derivative to 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 $i = 0, 1, \ldots, n$, in addition to the point-value pairs $\{(x_i, y_i)\}$, you are also given the first derivative $D_i = f'(x_i)$ 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 you 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, 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 $i = 1, 2, \ldots, n-1$,

$$D_{i-1} + 4D_i + D_{i+1} = 3(y_{i+1} - y_{i-1}) \,. \tag{28.23}$$

c. Show that

$$2D_0 + D_1 = 3(y_1 - y_0) \,, \tag{28.24}$$

$$D_{n-1} + 2D_n = 3(y_n - y_{n-1}) \,. \tag{28.25}$$

d. Rewrite equations (28.23)–(28.25) as a matrix equation involving the vector $D = (\, D_0 \;\; D_1 \;\; D_2 \;\; \cdots \;\; D_n \,)^{\mathrm{T}}$ 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 [180], Golub and Van Loan [192], Press, Teukolsky, Vetterling, and Flannery [365, 366], and Strang [422, 423].

Golub and Van Loan [192] 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 $\|A\|_\infty = \max\left\{\sum_{j=1}^{n} |a_{ij}| : 1 \le i \le n\right\}$. 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 [424], Strassen showed that an $n \times n$ matrix can be inverted in $O(n^{\lg 7})$ time. Winograd [460] 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 [422, 423] and Golub and Van Loan [192] contain good treatments of the SVD.

Strang [423] 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 you can specify the objective as a linear function of certain variables, and if you can specify the constraints on resources as equalities or inequalities on those variables, then you 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 you don't believe in. You realize, however, that certain issues may be more effective in winning votes in certain places. Your primary issues are preparing for a zombie apocalypse, equipping sharks with lasers, building highways for flying cars, and allowing dolphins to vote.

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 preparing for a zombie

policy	urban	suburban	rural
zombie apocalypse	−2	5	3
sharks with lasers	8	2	−5
highways for flying cars	0	0	10
dolphins voting	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.

apocalypse, $0 to equipping sharks with lasers, $4,000 to building highways for flying cars, and $9,000 to allowing dolphins to vote. In this case, you would win $(20 \cdot -2) + (0 \cdot 8) + (4 \cdot 0) + (9 \cdot 10) = 50$ thousand urban votes, $(20 \cdot 5) + (0 \cdot 2) + (4 \cdot 0) + (9 \cdot 0) = 100$ thousand suburban votes, and $(20 \cdot 3) + (0 \cdot -5) + (4 \cdot 10) + (9 \cdot -2) = 82$ 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, according to your model, 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.

It's natural to wonder whether this strategy is the best possible. That is, can you achieve your goals while spending less on advertising? Additional trial and error might help you to answer this question, but a better approach is to formulate (or ***model***) this question mathematically.

The first step is to decide what decisions you have to make and to introduce variables that capture these decisions. Since you have four decisions, you introduce four ***decision variables***:

- x_1 is the number of thousands of dollars spent on advertising on preparing for a zombie apocalypse,
- x_2 is the number of thousands of dollars spent on advertising on equipping sharks with lasers,
- x_3 is the number of thousands of dollars spent on advertising on building highways for flying cars, and
- x_4 is the number of thousands of dollars spent on advertising on allowing dolphins to vote.

You then think about ***constraints***, which are limits, or restrictions, on the values that the decision variables can take. You can write the requirement that you win at least 50,000 urban votes as

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

Similarly, you can write the requirements that you 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.

Finally, you think about your ***objective***, which is the quantity that you wish to either minimize or maximize. 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, you require that

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

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

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

The solution to this linear program yields your optimal strategy.

The remainder of this chapter covers how to formulate linear programs and is an introduction to modeling in general. Modeling refers to the general process of converting a problem into a mathematical form amenable to solution by an algorithm. Section 29.1 discusses briefly the algorithmic aspects of linear programming, although it does not include the details of a linear-programming algorithm. Throughout this book, we have seen ways to model problems, such as by shortest paths and connectivity in a graph. When modeling a problem as a linear program, you go through the steps used in this political example—identifying the decision variables, specifying the constraints, and formulating the objective function. In order to model a problem as a linear program, the constraints and objectives must be

linear. In Section 29.2, we will see several other examples of modeling via linear programs. Section 29.3 discusses duality, an important concept in linear programming and other optimization algorithms.

29.1 Linear programming formulations and algorithms

Linear programs take a particular form, which we will examine in this section. Multiple algorithms have been developed to solve linear programs. Some run in polynomial time, some do not, but they are all too complicated to show here. Instead, we will give an example that demonstrates some ideas behind the simplex algorithm, which is currently the most commonly deployed solution method.

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

$$f(x_1, x_2, \ldots, x_n) \le b \quad \text{and} \quad f(x_1, x_2, \ldots, x_n) \ge b$$

are ***linear inequalities***. We use the general term ***linear constraints*** to denote either linear equalities or linear inequalities. Linear programming does 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 minimizing, we call the linear program a ***minimization linear program***, and if maximizing, we call the linear program a ***maximization linear program***.

In order to discuss linear-programming algorithms and properties, it will be helpful to use a standard notation for the input. By convention, a maximization linear program takes as input 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 $j = 1, 2, \ldots, n$.

The goal is to find n real numbers $x_1, x_2, \ldots, x_n$ that

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.11}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.12}$$

$$x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n \ . \tag{29.13}$$

We call expression (29.11) the ***objective function*** and the $n + m$ inequalities in lines (29.12) and (29.13) the ***constraints***. The n constraints in line (29.13) are the ***nonnegativity constraints***. It can sometimes be more 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 (29.11)–(29.13) as

$$\text{maximize} \quad c^{\mathrm{T}} x \tag{29.14}$$

subject to

$$Ax \le b \tag{29.15}$$

$$x \ge 0 \ . \tag{29.16}$$

In line (29.14), $c^{\mathrm{T}} x$ is the inner product of two n-vectors. In inequality (29.15), Ax is the m-vector that is the product of an $m \times n$ matrix and an n-vector, and in inequality (29.16), $x \ge 0$ means that each entry of the vector x must be nonnegative. We call this representation the ***standard form*** for a linear program, and we adopt the convention that A, b, and c always have the dimensions given above.

The standard form above may not naturally correspond to real-life situations you are trying to model. For example, you might have equality constraints or variables that can take on negative values. Exercises 29.1-6 and 29.1-7 ask you to show how to convert any linear program into this standard form.

We now introduce terminology to describe solutions to linear programs. We denote a particular setting of the values in a variable, say x, by putting a bar over the variable name: $\bar{x}$. If $\bar{x}$ satisfies all the constraints, then it is a ***feasible solution***, but if it fails to satisfy at least one constraint, then it 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***, and otherwise, it is ***feasible***. The set of points that satisfy all the constraints is the ***feasible region***. If a linear program has some feasible solutions but does not have a finite optimal objective value, then the feasible region is ***unbounded*** and so is the linear program. Exercise 29.1-5 asks you to show that a linear program can have a finite optimal objective value even if the feasible region is unbounded.

One of the reasons for the power and popularity of linear programming is that linear programs can, in general, be solved efficiently. There are two classes of algorithms, known as ellipsoid algorithms and interior-point algorithms, that solve linear programs in polynomial time. In addition, the simplex algorithm is widely used. Although it does not run in polynomial time in the worst case, it tends to perform well in practice.

We will not give a detailed algorithm for linear programming, but will discuss a few important ideas. First, we will give an example of using a geometric procedure to solve a two-variable linear program. Although this example does not immediately generalize to an efficient algorithm for larger problems, it introduces some important concepts for linear programming and for optimization in general.

A two-variable linear program

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

$$\begin{aligned}
&\text{maximize} && x_1 + x_2 && && && (29.17)\\
&\text{subject to} && && && && \\
& && 4x_1 - x_2 && \le 8 && && (29.18)\\
& && 2x_1 + x_2 && \le 10 && && (29.19)\\
& && 5x_1 - 2x_2 && \ge -2 && && (29.20)\\
& && x_1, x_2 && \ge 0 && . && (29.21)
\end{aligned}$$

Figure 29.2(a) graphs the constraints in the (x_1, x_2)-Cartesian coordinate system. The feasible region in the two-dimensional space (highlighted in blue in the figure) is convex.[1] Conceptually, you could evaluate the objective function $x_1 + x_2$ at each point in the feasible region, and then identify a point that has the maximum objective value as an optimal solution. For this example (and for most linear programs), however, the feasible region contains an infinite number of points, and so to solve this linear program, you need 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, you 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. Plotting $x_1 + x_2 = 0$ produces 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 value z, the

[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.

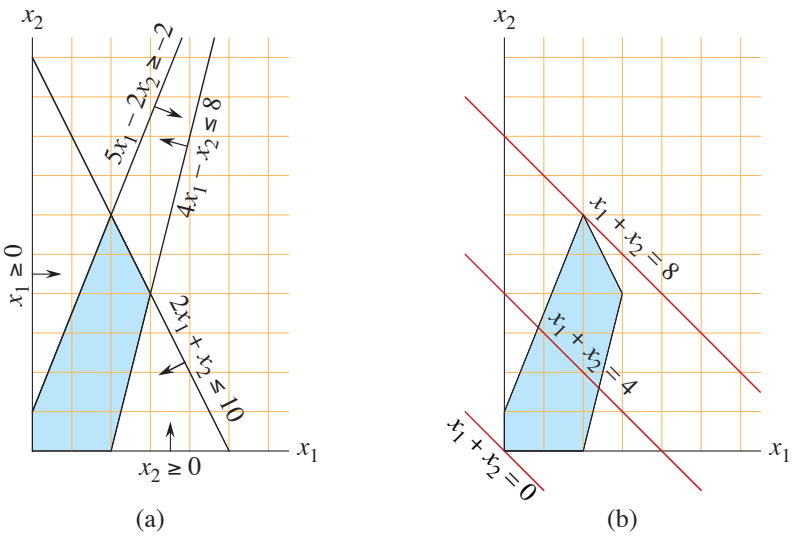


Figure 29.2 **(a)** The linear program given in (29.18)–(29.21). Each constraint is represented by a line and a direction. The intersection of the constraints, which is the feasible region, is highlighted in blue. **(b)** The red lines show, respectively, the points for which the objective value is 0, 4, and 8. The optimal solution to the linear program is $x_1 = 2$ and $x_2 = 6$ with objective value 8.

intersection 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 in the feasible region that maximizes $x_1 + x_2$ is an optimal solution to the linear program, which in this case is the vertex of the feasible region at $x_1 = 2$ 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 you cannot easily graph linear programs with more than two variables, the same intuition holds. If you 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 you move this plane away from the origin, in a direction normal to the objective function, you 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 you 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. Any algorithm for linear programming must also identify linear programs that have no solutions, as well as linear programs that have no finite optimal solution.

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 29.3, we'll see an important concept called "duality," which we'll use to prove that the solution returned by the simplex algorithm is indeed optimal.

The simplex 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 you add to a linear program the additional requirement that all variables take on integer values, you have an ***integer linear program***. Exercise 34.5-3 on page 1098 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 prob-

lems, there is no known polynomial-time algorithm for integer linear programming. In contrast, a general linear-programming problem can be solved in polynomial time.

Exercises

29.1-1

Consider the linear program

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

Give three feasible solutions to this linear program. What is the objective value of each one?

29.1-2

Consider the following linear program, which has a nonpositivity constraint:

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

Give three feasible solutions to this linear program. What is the objective value of each one?

29.1-3

Show that the following linear program is infeasible:

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

29.1-4

Show that the following linear program is unbounded:

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

29.1-5

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

29.1-6

Sometimes, in a linear program, you need to convert constraints from one form to another.

a. Show how to convert an equality constraint into an equivalent set of inequalities. That is, given a constraint $\sum_{j=1}^{n} a_{ij} x_j = b_i$, give a set of inequalities that will be satisfied if and only if $\sum_{j=1}^{n} a_{ij} x_j = b_i$,

b. Show how to convert an inequality constraint $\sum_{j=1}^{n} a_{ij} x_j \le b_i$ into an equality constraint and a nonnegativity constraint. You will need to introduce an additional variable s, and use the constraint that $s \ge 0$.

29.1-7

Explain how to convert a minimization linear program to an equivalent maximization linear program, and argue that your new linear program is equivalent to the original one.

29.1-8

In the political problem at the beginning of this chapter, there are feasible solutions that correspond to winning more voters than there actually are in the district. For example, you can set x_2 to 200, x_3 to 200, and $x_1 = x_4 = 0$. That solution is feasible, yet it seems to say that you will win 400,000 suburban voters, even though there are only 200,000 actual suburban voters. What constraints can you add to the linear program to ensure that you never seem to win more voters than there actually are? Even if you don't add these constraints, argue that the optimal solution to this linear program can never win more voters than there actually are in the district.

29.2 Formulating problems as linear programs

Linear programming has many 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. Here are two more examples:

- An airline wishes to schedule its flight crews. The Federal Aviation Administration imposes several 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.

Linear programs also model and solve graph and combinatorial problems, such as those appearing in this book. We have already seen a special case of linear programming used to solve systems of difference constraints in Section 22.4. In this section, we'll study how to formulate several graph and network-flow problems as linear programs. Section 35.4 uses linear programming as a tool to find an approximate solution to another graph problem.

Perhaps the most important aspect of linear programming is to be able to recognize when you can formulate a problem as a linear program. Once you cast a problem as a polynomial-sized linear program, you 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'll 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 from Chapter 22 and the maximum-flow problem from Chapter 24. 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 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, and 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 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. We'll 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 29.2-2.

In the single-pair shortest-path problem, the input is 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. The goal is 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, you need to determine a set of variables and constraints that define when you have a shortest path from s to t. The triangle inequality (Lemma 22.10 on page 633) gives $d_v \le d_u + w(u, v)$ for each edge $(u, v) \in E$. The source vertex initially receives a value $d_s = 0$, which never changes. Thus the following linear program expresses the shortest-path weight from s to t:

$$\text{maximize} \quad d_t \tag{29.22}$$

subject to

$$d_v \le d_u + w(u, v) \text{ for each edge } (u, v) \in E \tag{29.23}$$

$$d_s = 0 \ . \tag{29.24}$$

You might be surprised that this linear program maximizes an objective function when it is supposed to compute shortest paths. Minimizing the objective function would be a mistake, because when all the edge weights are nonnegative, setting $\bar{d}_v = 0$ for all $v \in V$ (recall that a bar over a variable name denotes a specific setting of the variable's value) would yield an optimal solution to the linear program without solving the shortest-paths problem. Maximizing is the right thing to do because an optimal solution to the shortest-paths problem sets each $\bar{d}_v$ to $\min\{\bar{d}_u + w(u, v) : u \in V \text{ and } (u, v) \in E\}$, so that $\bar{d}_v$ is the largest value that is less than or equal to all of the values in the set $\{\bar{d}_u + w(u, v)\}$. Therefore, it makes sense to maximize d_v for all vertices v on a shortest path from s to t subject to these constraints, 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, let's express the maximum-flow problem as a linear program. Recall that the input is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$, and two distinguished vertices: a source s and a sink t. As defined in Section 24.1, 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, the maximum-flow problem can be expressed as a linear program:

$$\text{maximize} \quad \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \tag{29.25}$$

subject to

$$f_{uv} \leq c(u, v) \quad \text{for each } u, v \in V \tag{29.26}$$

$$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \quad \text{for each } u \in V - \{s, t\} \tag{29.27}$$

$$f_{uv} \geq 0 \qquad \text{for each } u, v \in V \ . \tag{29.28}$$

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 (29.25)–(29.28) has, for ease of notation, a flow and capacity of 0 for each pair of vertices u, v with $(u, v) \notin E$. It is more efficient to rewrite the linear program so that it has $O(V + E)$ constraints. Exercise 29.2-4 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, 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 it can be solved by linear programming. Books abound with such real-world problems that linear programming can solve. Linear programming is also

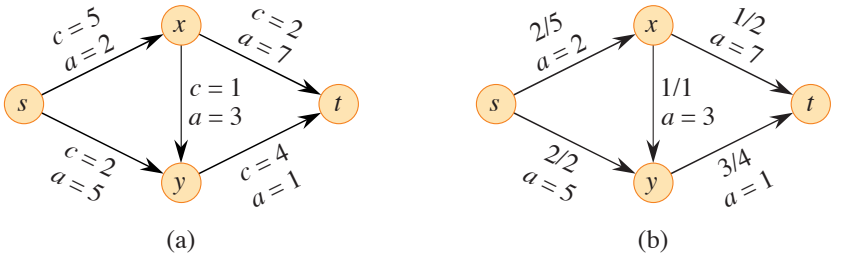


Figure 29.3 **(a)** An example of a minimum-cost-flow problem. Capacities are denoted by c and costs by a. Vertex s is the source, and vertex t is the sink. The goal is to send 4 units of flow from s to t. **(b)** A solution to the minimum-cost flow problem in which 4 units of flow are sent from s to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

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), you are given a real-valued cost $a(u, v)$. As in the maximum-flow problem, assume that $c(u, v) = 0$ if $(u, v) \notin E$ and that there are no antiparallel edges. If you send f_{uv} units of flow over edge (u, v), you incur a cost of $a(u, v) \cdot f_{uv}$. You are also given a flow demand d. You wish to send d units of flow from s to t while minimizing the total cost $\sum_{(u,v)\in E} a(u, v) \cdot f_{uv}$ 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, with a goal of sending 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 (29.26)–(29.28), incurs a total cost of $\sum_{(u,v)\in E} a(u, v) \cdot f_{uv}$. What is the particular 4-unit flow that minimizes this cost? Figure 29.3(b) shows an optimal solution, with total cost $\sum_{(u,v)\in E} a(u, v) \cdot f_{uv} = (2 \cdot 2) + (5 \cdot 2) + (3 \cdot 1) + (7 \cdot 1) + (1 \cdot 3) = 27$.

There are polynomial-time algorithms specifically designed for the minimum-cost-flow problem, but they are beyond the scope of this book. The minimum-cost-flow problem can be expressed as a linear program, however. The linear program looks similar to the one for the maximum-flow problem with the additional constraint that the value of the flow must be exactly d units, and with the new objective function of minimizing the cost:

$$\text{minimize} \quad \sum_{(u,v)\in E} a(u,v) \cdot f_{uv} \tag{29.29}$$

subject to

$$\begin{aligned} f_{uv} &\le c(u,v) && \text{for each } u, v \in V \\ \sum_{v\in V} f_{vu} - \sum_{v\in V} f_{uv} &= 0 && \text{for each } u \in V - \{s,t\} \\ \sum_{v\in V} f_{sv} - \sum_{v\in V} f_{vs} &= d \\ f_{uv} &\ge 0 && \text{for each } u, v \in V \ . \end{aligned} \tag{29.30}$$

Multicommodity flow

As a final example, let's consider another flow problem. Suppose that the Lucky Puck company from Section 24.1 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 are needed in 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***. The input to this problem is once again 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, implicitly assume that $c(u, v) = 0$ for $(u, v) \notin E$ and that the graph has no antiparallel edges. In addition, there are k different commodities, $K_1, K_2, \ldots, K_k$, with commodity i specified 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 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). This problem has no objective function: the question is to determine whether such a flow exists. Thus the linear program for this problem has a "null" objective function:

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

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
Write out explicitly the linear program corresponding to finding the shortest path from vertex s to vertex x in Figure 22.2(a) on page 609.

29.2-2
Given a graph G, write a linear program for the single-source shortest-paths problem. The solution should have the property that d_v is the shortest-path weight from the source vertex s to v for each vertex $v \in V$.

29.2-3
Write out explicitly the linear program corresponding to finding the maximum flow in Figure 24.1(a).

29.2-4
Rewrite the linear program for maximum flow (29.25)–(29.28) so that it uses only $O(V + E)$ constraints.

29.2-5
Write a linear program that, given a bipartite graph $G = (V, E)$, solves the maximum-bipartite-matching problem.

29.2-6
There can be more than one way to model a particular problem as a linear program. This exercise gives an alternative formulation for the maximum-flow problem. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ be the set of *all* possible directed simple paths from source s

to sink t. Using decision variables $x_1, \ldots, x_p$, where x_i is the amount of flow on path i, formulate a linear program for the maximum-flow problem. What is an upper bound on p, the number of directed simple paths from s to t?

29.2-7

In the ***minimum-cost multicommodity-flow problem***, the input is a 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, there are k different commodities, $K_1, K_2, \ldots, K_k$, with commodity i specified 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 $\sum_{u,v \in V} a(u, v) \cdot f_{uv}$, and the goal is to find the feasible flow of minimum cost. Express this problem as a linear program.

29.3 Duality

We will now introduce a powerful concept called ***linear-programming duality***. In general, given a maximization problem, duality allows you to formulate a related minimization problem that has the same objective value. The idea of duality is actually more general than linear programming, but we restrict our attention to linear programming in this section.

Duality enables us to prove that a solution is indeed optimal. We saw an example of duality in Chapter 24 with Theorem 24.6, the max-flow min-cut theorem. Suppose that, given an instance of a maximum-flow problem, you find a flow f with value $|f|$. How do you know whether f is a maximum flow? By the max-flow min-cut theorem, if you can find a cut whose value is also $|f|$, then you have verified that f is indeed a maximum flow. This relationship provides an example of duality: given a maximization problem, define a related minimization problem such that the two problems have the same optimal objective values.

Given a linear program in standard form in which the objective is to maximize, let's see 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 the primal linear program

$$\text{maximize} \quad \sum_{j=1}^{n} c_j x_j \tag{29.31}$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \text{for } i = 1, 2, \ldots, m \tag{29.32}$$

$$x_j \ge 0 \quad \text{for } j = 1, 2, \ldots, n \ , \tag{29.33}$$

its dual is

$$\text{minimize} \quad \sum_{i=1}^{m} b_i y_i \tag{29.34}$$

subject to

$$\sum_{i=1}^{m} a_{ij} y_i \ge c_j \quad \text{for } j = 1, 2, \ldots, n \tag{29.35}$$

$$y_i \ge 0 \quad \text{for } i = 1, 2, \ldots, m \ . \tag{29.36}$$

Mechanically, to form the dual, change the maximization to a minimization, exchange the roles of coefficients on the right-hand sides and in the objective function, and replace each $\le$ by $\ge$. Each of the m constraints in the primal corresponds to a variable y_i in the dual. Likewise, each of the n constraints in the dual corresponds to a variable x_j in the primal. For example, consider the following primal linear program:

$$\text{maximize} \quad 3x_1 + x_2 + 4x_3 \tag{29.37}$$

subject to

$$x_1 + x_2 + 3x_3 \le 30 \tag{29.38}$$

$$2x_1 + 2x_2 + 5x_3 \le 24 \tag{29.39}$$

$$4x_1 + x_2 + 2x_3 \le 36 \tag{29.40}$$

$$x_1, x_2, x_3 \ge 0 \ . \tag{29.41}$$

Its dual is

$$\text{minimize} \quad 30y_1 + 24y_2 + 36y_3 \tag{29.42}$$

subject to

$$y_1 + 2y_2 + 4y_3 \ge 3 \tag{29.43}$$

$$y_1 + 2y_2 + y_3 \ge 1 \tag{29.44}$$

$$3y_1 + 5y_2 + 2y_3 \ge 4 \tag{29.45}$$

$$y_1, y_2, y_3 \ge 0 \ . \tag{29.46}$$

Although forming the dual can be considered a mechanical operation, there is an intuitive explanation. Consider the primal maximization problem (29.37)–(29.41). Each constraint gives an upper bound on the objective function. In addition, if you

take one or more constraints and add together nonnegative multiples of them, you get a valid constraint. For example, you can add constraints (29.38) and (29.39) to obtain the constraint $3x_1 + 3x_2 + 8x_3 \leq 54$. Any feasible solution to the primal must satisfy this new constraint, but there is something else interesting about it. Comparing this new constraint to the objective function (29.37), you can see that for each variable, the corresponding coefficient is at least as large as the coefficient in the objective function. Thus, since the variables x_1, x_2 and x_3 are nonnegative, we have that

$$3x_1 + x_2 + 4x_3 \leq 3x_1 + 3x_2 + 8x_3 \leq 54 ,$$

and so the solution value to the primal is at most 54. In other words, adding these two constraints together has generated an upper bound on the objective value.

In general, for any nonnegative multipliers y_1, y_2, and y_3, you can generate a constraint

$$y_1(x_1+x_2+3x_3)+y_2(2x_1+2x_2+5x_3)+y_3(4x_1+x_2+2x_3) \leq 30y_1+24y_2+36y_3$$

from the primal constraints or, by distributing and regrouping,

$$(y_1+2y_2+4y_3)x_1+(y_1+2y_2+y_3)x_2+(3y_1+5y_2+2y_3)x_3 \leq 30y_1+24y_2+36y_3 .$$

Now, as long as this constraint has coefficients of x_1, x_2, and x_3 that are at least their objective-function coefficients, it is a valid upper bound. That is, as long as

$$\begin{aligned} y_1 + 2y_2 + 4y_3 &\geq 3 , \\ y_1 + 2y_2 + y_3 &\geq 1 , \\ 3y_1 + 5y_2 + 2y_3 &\geq 4 , \end{aligned}$$

you have a valid upper bound of $30y_1+24y_2+36y_3$. The multipliers y_1, y_2, and y_3 must be nonnegative, because otherwise you cannot combine the inequalities. Of course, you would like the upper bound to be as small as possible, and so you want to choose y to minimize $30y_1 + 24y_2 + 36y_3$. Observe that we have just described the dual linear program as the problem of finding the smallest possible upper bound on the primal.

We'll formalize this idea and show in Theorem 29.4 that, if the linear program and its dual are feasible and bounded, then the optimal value of the dual linear program is always equal to the optimal value of the primal linear program. 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 29.1 (Weak linear-programming duality)
Let $\bar{x}$ be any feasible solution to the primal linear program in (29.31)–(29.33), and let $\bar{y}$ be any feasible solution to its dual linear program in (29.34)–(29.36). Then

$$\sum_{j=1}^{n} c_j \bar{x}_j \le \sum_{i=1}^{m} b_i \bar{y}_i \ .$$

Proof We have

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\le \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \quad \text{(by inequalities (29.35))} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\
&\le \sum_{i=1}^{m} b_i \bar{y}_i \qquad \text{(by inequalities (29.32))} \ .
\end{aligned}$$

■

Corollary 29.2
Let $\bar{x}$ be a feasible solution to the primal linear program in (29.31)–(29.33), and let $\bar{y}$ be a feasible solution to its dual linear program in (29.34)–(29.36). 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 29.1, 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. ■

We now show that, at optimality, the primal and dual objective values are indeed equal. To prove linear programming duality, we will require one lemma from linear algebra, known as Farkas's lemma, the proof of which Problem 29-4 asks you to provide. Farkas's lemma can take several forms, each of which is about when a set of linear equalities has a solution. In stating the lemma, we use $m + 1$ as a dimension because it matches our use below.

Lemma 29.3 (Farkas's lemma)
Given $M \in \mathbb{R}^{(m+1)\times n}$ and $g \in \mathbb{R}^{m+1}$, exactly one of the following statements is true:

1. There exists $v \in \mathbb{R}^n$ such that $Mv \le g$,
2. There exists $w \in \mathbb{R}^{m+1}$ such that $w \ge 0$, $w^{\mathrm{T}}M = 0$ (an n-vector of all zeros), and $w^{\mathrm{T}}g < 0$.

■

Theorem 29.4 (Linear-programming duality)
Given the primal linear program in (29.31)–(29.33) and its corresponding dual in (29.34)–(29.36), if both are feasible and bounded, then for optimal solutions x^* and y^*, we have $c^{\mathrm{T}}x^* = b^{\mathrm{T}}y^*$.

Proof Let $\mu = b^{\mathrm{T}}y^*$ be the optimal value of the dual linear program given in (29.34)–(29.36). Consider an augmented set of primal constraints in which we add a constraint to (29.31)–(29.33) that the objective value is at least μ. We write out this ***augmented primal*** as

$$Ax \le b\,, \tag{29.47}$$
$$c^{\mathrm{T}}x \ge \mu\,. \tag{29.48}$$

We can multiply (29.48) through by -1 and rewrite (29.47)–(29.48) as

$$\begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} x \le \begin{pmatrix} b \\ -\mu \end{pmatrix}. \tag{29.49}$$

Here, $\begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix}$ denotes an $(m+1)\times n$ matrix, x is an n-vector, and $\begin{pmatrix} b \\ -\mu \end{pmatrix}$ denotes an $(m+1)$-vector.

We claim that if there is a feasible solution $\bar{x}$ to the augmented primal, then the theorem is proved. To establish this claim, observe that $\bar{x}$ is also a feasible solution to the original primal and that it has objective value at least μ. We can then apply Lemma 29.1, which states that the objective value of the primal is at most μ, to complete the proof of the theorem.

It therefore remains to show that the augmented primal has a feasible solution. Suppose, for the purpose of contradiction, that the augmented primal is infeasible, which means that there is no $v \in \mathbb{R}^n$ such that $\begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} v \le \begin{pmatrix} b \\ -\mu \end{pmatrix}$. We can apply Farkas's lemma, Lemma 29.3, to inequalty (29.49) with

$$M = \begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} \text{ and } g = \begin{pmatrix} b \\ -\mu \end{pmatrix}.$$

Because the augmented primal is infeasible, condition 1 of Farkas's lemma does not hold. Therefore, condition 2 must apply, so that there must exist a $w \in \mathbb{R}^{m+1}$ such that $w \ge 0$, $w^{\mathrm{T}}M = 0$, and $w^{\mathrm{T}}g < 0$. Let's write w as $w = \begin{pmatrix} \bar{y} \\ \lambda \end{pmatrix}$ for some $\bar{y} \in \mathbb{R}^m$ and $\lambda \in \mathbb{R}$, where $\bar{y} \ge 0$ and $\lambda \ge 0$. Substituting for w, M, and g in condition 2 gives

$$\begin{pmatrix} \bar{y} \\ \lambda \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} A \\ -c^{\mathrm{T}} \end{pmatrix} = 0 \text{ and } \begin{pmatrix} \bar{y} \\ \lambda \end{pmatrix}^{\mathrm{T}} \begin{pmatrix} b \\ -\mu \end{pmatrix} < 0\,.$$

Unpacking the matrix notation gives

$$\bar{y}^{\mathrm{T}}A - \lambda c^{\mathrm{T}} = 0 \text{ and } \bar{y}^{\mathrm{T}}b - \lambda\mu < 0 \,. \tag{29.50}$$

We now show that the requirements in (29.50) contradict the assumption that μ is the optimal solution value for the dual linear program. We consider two cases.

The first case is when $\lambda = 0$. In this case, (29.50) simplifies to

$$\bar{y}^{\mathrm{T}}A = 0 \text{ and } \bar{y}^{\mathrm{T}}b < 0 \,. \tag{29.51}$$

We'll now construct a dual feasible solution y' with an objective value smaller than $b^{\mathrm{T}}y^*$. Set $y' = y^* + \epsilon\bar{y}$, for any $\epsilon > 0$. Since

$$\begin{aligned} y'^{\mathrm{T}}A &= (y^* + \epsilon\bar{y})^{\mathrm{T}}A \\ &= y^{*\mathrm{T}}A + \epsilon\bar{y}^{\mathrm{T}}A \\ &= y^{*\mathrm{T}}A && \text{(by (29.51))} \\ &\geq c^{\mathrm{T}} && \text{(because } y^* \text{ is feasible)} \,, \end{aligned}$$

y' is feasible. Now consider the objective value

$$\begin{aligned} b^{\mathrm{T}}y' &= b^{\mathrm{T}}(y^* + \epsilon\bar{y}) \\ &= b^{\mathrm{T}}y^* + \epsilon b^{\mathrm{T}}\bar{y} \\ &< b^{\mathrm{T}}y^* \,, \end{aligned}$$

where the last inequality follows because $\epsilon > 0$ and, by (29.51), $\bar{y}^{\mathrm{T}}b = b^{\mathrm{T}}\bar{y} < 0$ (since both $\bar{y}^{\mathrm{T}}b$ and $b^{\mathrm{T}}\bar{y}$ are the inner product of b and $\bar{y}$), and so their product is negative. Thus we have a feasible dual solution of value less than μ, which contradicts μ being the optimal objective value.

We now consider the second case, where $\lambda > 0$. In this case, we can take (29.50) and divide through by λ to obtain

$$(\bar{y}^{\mathrm{T}}/\lambda)A - (\lambda/\lambda)c^{\mathrm{T}} = 0 \text{ and } (\bar{y}^{\mathrm{T}}/\lambda)b - (\lambda/\lambda)\mu < 0 \,. \tag{29.52}$$

Now set $y' = \bar{y}/\lambda$ in (29.52), giving

$$y'^{\mathrm{T}}A = c^{\mathrm{T}} \text{ and } y'^{\mathrm{T}}b < \mu \,.$$

Thus, y' is a feasible dual solution with objective value strictly less than μ, a contradiction. We conclude that the augmented primal has a feasible solution, and the theorem is proved. ■

Fundamental theorem of linear programming

We conclude this chapter by stating the fundamental theorem of linear programming, which extends Theorem 29.4 to the cases when the linear program may be either feasible or unbounded. Exercise 29.3-8 asks you to provide the proof.

Theorem 29.5 (Fundamental theorem of linear programming)
Any linear program, given in standard form, either

1. has an optimal solution with a finite objective value,
2. is infeasible, or
3. is unbounded. ■

Exercises

29.3-1
Formulate the dual of the linear program given in lines (29.6)–(29.10) on page 852.

29.3-2
You have a linear program that is not in standard form. You could produce the dual by first converting it to standard form, and then taking the dual. It would be more convenient, however, to produce the dual directly. Explain how to directly take the dual of an arbitrary linear program.

29.3-3
Write down the dual of the maximum-flow linear program, as given in lines (29.25)–(29.28) on page 862. Explain how to interpret this formulation as a minimum-cut problem.

29.3-4
Write down the dual of the minimum-cost-flow linear program, as given in lines (29.29)–(29.30) on page 864. Explain how to interpret this problem in terms of graphs and flows.

29.3-5
Show that the dual of the dual of a linear program is the primal linear program.

29.3-6
Which result from Chapter 24 can be interpreted as weak duality for the maximum-flow problem?

29.3-7
Consider the following 1-variable primal linear program:

$$\begin{aligned} &\text{maximize} && tx \\ &\text{subject to} \\ & && rx \le s \\ & && x \ge 0 \ , \end{aligned}$$

where r, s, and t are arbitrary real numbers. State for which values of r, s, and t you can assert that

1. Both the primal linear program and its dual have optimal solutions with finite objective values.
2. The primal is feasible, but the dual is infeasible.
3. The dual is feasible, but the primal is infeasible.
4. Neither the primal nor the dual is feasible.

29.3-8
Prove the fundamental theorem of linear programming, Theorem 29.5.

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. Given an algorithm for the linear-programming problem, show how to 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. Given an algorithm for the linear-inequality feasibility problem, show how to 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 (29.31)–(29.33), and let $\bar{y}$ be a feasible solution to the dual linear program given in (29.34)–(29.36). Complementary slackness states that the following conditions are necessary and sufficient for $\bar{x}$ and $\bar{y}$ to be optimal:

$$\sum_{i=1}^{m} a_{ij}\bar{y}_i = c_j \text{ or } \bar{x}_j = 0 \text{ for } j = 1, 2, \ldots, n$$

and

$$\sum_{j=1}^{n} a_{ij}\bar{x}_j = b_i \text{ or } \bar{y}_i = 0 \text{ for } i = 1, 2, \ldots, m .$$

a. Verify that complementary slackness holds for the linear program in lines (29.37)–(29.41).

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 (29.31)–(29.33) 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 (29.34)–(29.36),
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 integer values. Exercise 34.5-3 on page 1098 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 29.1) holds for an integer linear program.

b. Show that duality (Theorem 29.4) does not always hold for an integer linear program.

c. Given a primal linear program in standard form, let P be the optimal objective value for the primal linear program, D be the optimal objective value for its dual, IP 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 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

Prove Farkas's lemma, Lemma 29.3.

29-5 Minimum-cost circulation

This problem considers a variant of the minimum-cost-flow problem from Section 29.2 in which there is no demand, source, or sink. Instead, the input, as before, contains a flow network, capacity constraints $c(u, v)$, 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 $a(u, v) > 0$. What does an optimal solution to the minimum-cost-circulation problem look like?

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 derive 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 [94], Gass [178], Karloff [246], Schrijver [398], and Vanderbei [444]. Many other books give a good coverage of linear programming, including those by Papadimitriou and Steiglitz [353] 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 [94] and [246].

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 [201] 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 [247] 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 [189] and the book by Ye [463].

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 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 [421] 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 [349] 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. This chapter will 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 expresses 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. Many fine books delve into the rich area of signal processing, and 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:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \ .$$

The values $a_0, a_1, \ldots, a_{n-1}$ are the ***coefficients*** of the polynomial. The coefficients and x 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, in which case we say that $\text{degree}(A) = k$. 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.

A variety of operations extend to polynomials. For ***polynomial addition***, if $A(x)$ and $B(x)$ are polynomials of degree-bound n, their ***sum*** is a polynomial $C(x)$, also

of degree-bound n, such that $C(x) = A(x) + B(x)$ for all x in the underlying field. That is, if

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \quad \text{and} \quad B(x) = \sum_{j=0}^{n-1} b_j x^j ,$$

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, given the polynomials $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$, their sum is $C(x) = 4x^3 + 7x^2 - 6x + 4$.

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, you can multiply $A(x) = 6x^3 + 7x^2 - 10x + 9$ and $B(x) = -2x^3 + 4x - 5$ as follows:

$$\begin{array}{rrrrrrr}
 & & & 6x^3 & + 7x^2 & - 10x & + 9 \\
 & & & - 2x^3 & & + 4x & - 5 \\
\hline
 & & & - 30x^3 & - 35x^2 & + 50x & - 45 \quad \text{(multiply } A(x) \text{ by } -5) \\
 & & 24x^4 & + 28x^3 & - 40x^2 & + 36x & \quad \text{(multiply } A(x) \text{ by } 4x) \\
- 12x^6 & - 14x^5 & + 20x^4 & - 18x^3 & & & \quad \text{(multiply } A(x) \text{ by } -2x^3) \\
\hline
- 12x^6 & - 14x^5 & + 44x^4 & - 20x^3 & - 75x^2 & + 86x & - 45
\end{array}$$

Another way to express the product $C(x)$ is

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j , \tag{30.1}$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} , \tag{30.2}$$

(By the definition of degree, $a_k = 0$ for all $k >$ degree(A) and $b_k = 0$ for all $k >$ degree(B).) If A is a polynomial of degree-bound n_a and B is a polynomial of degree-bound n_b, then C must be a polynomial of degree-bound $n_a + n_b - 1$, because degree(C) = degree(A) + degree(B). Since a polynomial of degree-bound k is also a polynomial of degree-bound $k+1$, we normally make the somewhat simpler statement 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 method for multiplying polynomials of degree n—equations (30.1) and (30.2)—takes $\Theta(n^2)$ time with polynomials represented in coefficient form, but only $\Theta(n)$ time with point-value form. Converting between the two representations, however, reduces the time to multiply polynomials to just $\Theta(n \lg n)$. To see why this approach works, you must first understand complex roots of unity, which Section 30.2 covers. Section 30.2 then uses the FFT and its inverse to perform the conversions. Because the FFT is used so often in signal processing, it is often implemented as a circuit in hardware, and Section 30.3 illustrates the structure of such circuits.

This chapter relies on complex numbers, and within this chapter the symbol i denotes $\sqrt{-1}$ exclusively.

30.1 Representing polynomials

The coefficient and point-value representations of polynomials are in a sense equivalent: a polynomial in point-value form has a unique counterpart in coefficient form. This section introduces the two representations and shows how to combine them in order to 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 $a = (a_0, a_1, \ldots, a_{n-1})$. Matrix equations in this chapter 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 $A(x_0)$. To evaluate a polynomial in $\Theta(n)$ time, use ***Horner's rule***:

$$A(x_0) = a_0 + x_0\left(a_1 + x_0\left(a_2 + \cdots + x_0\left(a_{n-2} + x_0(a_{n-1})\right)\cdots\right)\right) .$$

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: 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. The method described by equations (30.1) and (30.2) takes $\Theta(n^2)$ time, since it multiplies each coefficient in the vector a by each co-

efficient 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 any set of n distinct points $x_0, x_1, \ldots, x_{n-1}$ can serve as a basis for the representation.

Computing a point-value representation for a polynomial given in coefficient form is in principle straightforward, since all you 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 $\Theta(n^2)$ time. We'll see later that if you choose the points x_k cleverly, you can accelerate this computation to run in $\Theta(n \lg n)$ time.

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 on page 1223, this matrix has determinant

$$\prod_{0 \le j < k \le n-1} (x_k - x_j) ,$$

and therefore, by Theorem D.5 on page 1221, it is invertible (that is, nonsingular) if the x_k are distinct. To solve for the coefficients a_j uniquely given the point-value representation, use the inverse of the Vandermonde matrix:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y . \qquad \blacksquare$$

The proof of Theorem 30.1 describes an algorithm for interpolation based on solving the set (30.4) of linear equations. Section 28.1 shows how to solve these equations in $O(n^3)$ time.

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 might want 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 $\Theta(n^2)$ time.

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 $\Theta(n^2)$ time.

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, given point-value representations for A,

$$\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} ,$$

[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.

and for B,

$$\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{n-1}, y'_{n-1})\} ,$$

where 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 to obtain a point-value representation for C, just pointwise multiply a point-value representation for A by a point-value representation for B. Polynomial multiplication differs from polynomial addition in one key aspect, however: degree(C) = degree(A) + degree(B), so that if A and B have degree-bound n, then C has degree-bound $2n$. A standard point-value representation for A and B consists of n point-value pairs for each polynomial. Multiplying these together gives n point-value pairs, but $2n$ pairs are necessary to interpolate a unique polynomial C of degree-bound $2n$. (See Exercise 30.1-4.) Instead, 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, multiplying them to obtain the point-value form of the result takes just $\Theta(n)$ time, much less than the $\Theta(n^2)$ time required to multiply polynomials in coefficient form.

Finally, let's consider how to evaluate a polynomial given in point-value form at a new point. For this problem, the simplest approach known is to first convert the polynomial to coefficient form and then evaluate it at the new point.

Fast multiplication of polynomials in coefficient form

Can the linear-time multiplication method for polynomials in point-value form expedite polynomial multiplication in coefficient form? The answer hinges on

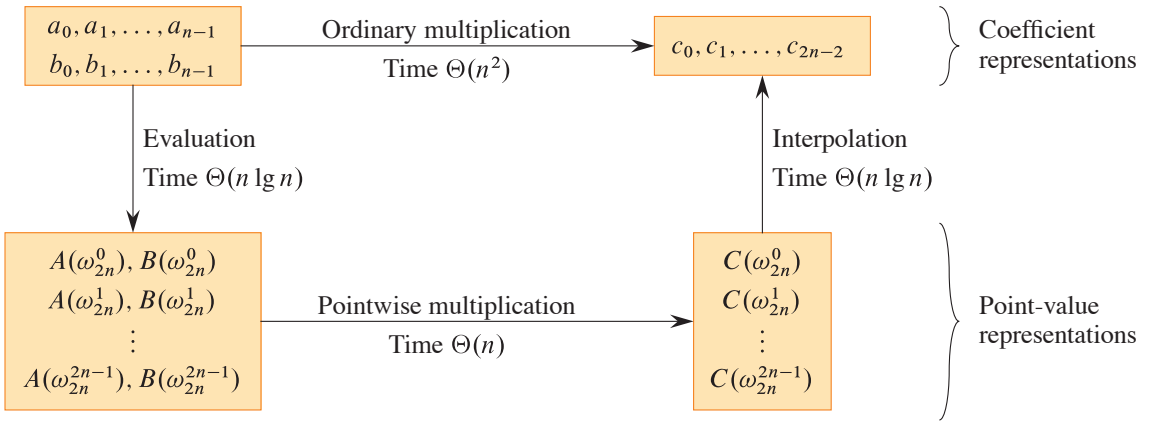


Figure 30.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, and those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots of unity.

whether it is possible convert a polynomial quickly from coefficient form to point-value form (evaluate) and vice versa (interpolate).

Any points can serve as evaluation points, but certain evaluation points allow conversion between representations in only $\Theta(n \lg n)$ time. As we'll see in Section 30.2, if "complex roots of unity" are the evaluation points, then the discrete Fourier transform (or DFT) evaluates and the inverse DFT interpolates. Section 30.2 shows 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, first double their degree-bounds to $2n$ by adding n high-order coefficients of 0. Because the vectors have $2n$ elements, use "complex $(2n)$th roots of unity," which are denoted by the ω_{2n} terms in Figure 30.1.

The following procedure takes advantage of the FFT to multiply two polynomials $A(x)$ and $B(x)$ of degree-bound n in $\Theta(n \lg n)$-time, where the input and output representations are in coefficient form. The procedure assumes that n is an exact power of 2, so if it isn't, just add 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 $\Theta(n)$ time, and steps (2) and (4) take $\Theta(n \lg n)$ time. Thus, once we show how to use the FFT, we will have proven the following.

Theorem 30.2
Two polynomials of degree-bound n with both the input and output representations in coefficient form can be multiplied in $\Theta(n \lg n)$ time. ■

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

$$A(x) = q(x)(x - x_0) + r \ .$$

Then we have $A(x_0) = r$. Show how to compute the remainder r and the coefficients of $q(x)$ from x_0 and the coefficients of A in $\Theta(n)$ time.

30.1-3
Given a polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$, define $A^{\text{rev}}(x) = \sum_{j=0}^{n-1} a_{n-1-j} x^j$. Show how to derive a point-value representation for $A^{\text{rev}}(x)$ from a point-value representation for $A(x)$, 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 $\Theta(n^2)$ time. (*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 $O(n)$ time.)

30.1-6

Explain what is wrong with the "obvious" approach to polynomial division using a point-value representation: 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 $10n$. The ***Cartesian sum*** of A and B is defined by

$$C = \{x + y : x \in A \text{ and } y \in B\} .$$

The integers in C lie in the range from 0 to $20n$. Show how, in $O(n \lg n)$ time, 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. (*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 by computing the DFT and its inverse by using the FFT, it is possible to evaluate and interpolate a degree n polynomial at the complex roots of unity in $\Theta(n \lg n)$ time. This section defines complex roots of unity, studies their properties, defines the DFT, and then shows 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: $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$. To interpret this formula, 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

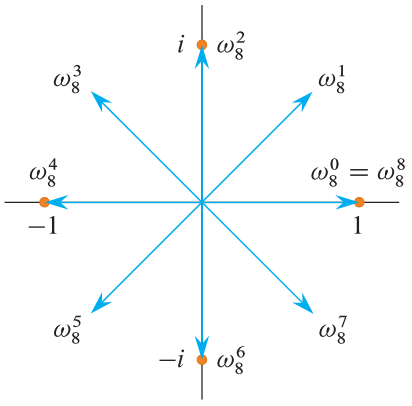


Figure 30.2 The values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$ in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.

$$\omega_n = e^{2\pi i/n} \tag{30.6}$$

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 $n > 0$, $k \geq 0$, 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, $(\omega_n^k)^2 = \omega_{n/2}^k$ for any nonnegative integer k. Squaring all of the complex nth roots of unity produces 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 $\omega_n^{k+n/2} = \omega_n^k\omega_n^{n/2} = -\omega_n^k$, and thus $(\omega_n^{k+n/2})^2 = (-\omega_n^k)^2 = (\omega_n^k)^2$. ■

As we'll see, the halving lemma is essential to the 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.6) on page 1142 applies to complex values as well as to reals, giving

$$\begin{aligned}\sum_{j=0}^{n-1}\left(\omega_n^k\right)^j &= \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1}\\ &= \frac{(\omega_n^n)^k - 1}{\omega_n^k - 1}\\ &= \frac{(1)^k - 1}{\omega_n^k - 1}\\ &= 0\,.\end{aligned}$$

To see that the denominator is not 0, note that $\omega_n^k = 1$ only when k is divisible by n, which the lemma statement prohibits. ■

The DFT

Recall the goal of evaluating 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] The polynomial 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

The ***fast Fourier transform (FFT)*** takes advantage of the special properties of the complex roots of unity to compute $\mathrm{DFT}_n(a)$ in $\Theta(n \lg n)$ time, as opposed to the $\Theta(n^2)$ time of the straightforward method. Assume throughout that n is an exact power of 2. Although strategies for dealing with sizes that are not exact powers of 2 are known, they are beyond the scope of this book.

---

[3] The length n is actually what Section 30.1 referred to as $2n$, since the degree-bound of the given polynomials doubles prior to evaluation. In the context of polynomial multiplication, therefore, we are actually working with complex $(2n)$th roots of unity.

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 $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ of degree-bound $n/2$:

$$\begin{aligned} A^{\text{even}}(x) &= a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{n/2-1} , \\ A^{\text{odd}}(x) &= a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{n/2-1} . \end{aligned}$$

Note that A^{even} contains all the even-indexed coefficients of A (the binary representation of the index ends in 0) and A^{odd} contains all the odd-indexed coefficients (the binary representation of the index ends in 1). It follows that

$$A(x) = A^{\text{even}}(x^2) + x A^{\text{odd}}(x^2) , \tag{30.9}$$

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 $A^{\text{even}}(x)$ and $A^{\text{odd}}(x)$ 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, the FFT recursively evaluates the polynomials A^{even} and A^{odd} 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, dividing an n-element DFT_n computation into two $n/2$-element $\text{DFT}_{n/2}$ computations. This decomposition is the basis for the FFT procedure on the next page, which computes the DFT of an n-element vector $a = (a_0, a_1, \ldots, a_{n-1})$, where n is an exact power of 2.

The FFT procedure works as follows. Lines 1–2 represent the base case of the recursion. The DFT of 1 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 5–6 define the coefficient vectors for the polynomials A^{even} and A^{odd}. Lines 3, 4, and 12 guarantee that ω is updated properly so that whenever lines 10–11 are executed, $\omega = \omega_n^k$. (Keeping a running value of ω from iteration to iteration saves

```
FFT(a, n)
 1  if n == 1
 2      return a                          // DFT of 1 element is the element itself
 3  ω_n = e^{2πi/n}
 4  ω = 1
 5  a^even = (a_0, a_2, ..., a_{n-2})
 6  a^odd = (a_1, a_3, ..., a_{n-1})
 7  y^even = FFT(a^even, n/2)
 8  y^odd = FFT(a^odd, n/2)
 9  for k = 0 to n/2 − 1                  // at this point, ω = ω_n^k
10      y_k = y_k^even + ω y_k^odd
11      y_{k+(n/2)} = y_k^even − ω y_k^odd
12      ω = ω ω_n
13  return y
```

time over computing ω_n^k from scratch each time through the **for** loop.[4]) Lines 7–8 perform the recursive $\mathrm{DFT}_{n/2}$ computations, setting, for $k = 0, 1, \ldots, n/2 - 1$,

$$y_k^{\text{even}} = A^{\text{even}}(\omega_{n/2}^k) ,$$
$$y_k^{\text{odd}} = A^{\text{odd}}(\omega_{n/2}^k) ,$$

or, since $\omega_{n/2}^k = \omega_n^{2k}$ by the cancellation lemma,

$$y_k^{\text{even}} = A^{\text{even}}(\omega_n^{2k}) ,$$
$$y_k^{\text{odd}} = A^{\text{odd}}(\omega_n^{2k}) .$$

Lines 10–11 combine the results of the recursive $\mathrm{DFT}_{n/2}$ calculations. For the first $n/2$ results $y_0, y_1, \ldots, y_{n/2-1}$, line 10 yields

$$\begin{aligned} y_k &= y_k^{\text{even}} + \omega_n^k y_k^{\text{odd}} \\ &= A^{\text{even}}(\omega_n^{2k}) + \omega_n^k A^{\text{odd}}(\omega_n^{2k}) \\ &= A(\omega_n^k) \qquad \text{(by equation (30.9))} . \end{aligned}$$

For $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$, letting $k = 0, 1, \ldots, n/2 - 1$, line 11 yields

[4] The downside of iteratively updating ω is that round-off errors can accumulate, especially for larger input sizes. Several techniques to limit the magnitude of FFT round-off errors have been proposed, but are beyond the scope of this book. If several FFTs are going to be run on inputs of the same size, then it might be worthwhile to directly precompute a table of all $n/2$ values of ω_n^k.

$$\begin{aligned}
y_{k+(n/2)} &= y_k^{\text{even}} - \omega_n^k y_k^{\text{odd}} \\
&= y_k^{\text{even}} + \omega_n^{k+(n/2)} y_k^{\text{odd}} && \text{(since } \omega_n^{k+(n/2)} = -\omega_n^k\text{)} \\
&= A^{\text{even}}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{\text{odd}}(\omega_n^{2k}) \\
&= A^{\text{even}}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{\text{odd}}(\omega_n^{2k+n}) && \text{(since } \omega_n^{2k+n} = \omega_n^{2k}\text{)} \\
&= A(\omega_n^{k+(n/2)}) && \text{(by equation (30.9)) .}
\end{aligned}$$

Thus the vector y returned by FFT is indeed the DFT of the input vector a.

Lines 10 and 11 multiply each value y_k^{odd} by ω_n^k, for $k = 0, 1, \ldots, n/2 - 1$. Line 10 adds this product to y_k^{even}, and line 11 subtracts it. Because each factor ω_n^k appears in both its positive and negative forms, we call the factors ω_n^k ***twiddle factors***.

To determine the running time of the procedure FFT, note that exclusive of the recursive calls, each invocation takes $\Theta(n)$ time, where n is the length of the input vector. The recurrence for the running time is therefore

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

by case 2 of the master theorem (Theorem 4.1). Thus the FFT can evaluate a polynomial of degree-bound n at the complex nth roots of unity in $\Theta(n \lg n)$ time.

Interpolation at the complex roots of unity

The polynomial multiplication scheme entails converting from coefficient form to point-value form by evaluating the polynomial at the complex roots of unity, pointwise multiplying, and finally converting from point-value form back to coefficient form by interpolating. We've just seen how to evaluate, so now we'll see how to interpolate the complex roots of unity by a polynomial. To interpolate, we'll write the DFT as a matrix equation and then look 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 factors 0 to $n - 1$.

For the inverse operation, which we write as $a = \mathrm{DFT}_n^{-1}(y)$, multiply 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^{-jk}/n.

Proof We show that $V_n^{-1}V_n = I_n$, the $n \times n$ identity matrix. Consider the (k,k') entry of $V_n^{-1}V_n$:

$$\begin{aligned}[V_n^{-1}V_n]_{kk'} &= \sum_{j=0}^{n-1} (\omega_n^{-jk}/n)(\omega_n^{jk'}) \\ &= \sum_{j=0}^{n-1} \omega_n^{j(k'-k)}/n \ .\end{aligned}$$

This summation equals 1 if $k' = k$, and it is 0 otherwise by the summation lemma (Lemma 30.6). Note that in order for the summation lemma to apply, $k' - k$ must not be divisible by n. Indeed, it is not, since $-(n-1) \le k' - k \le n-1$. ■

With the inverse matrix V_n^{-1} defined, $\mathrm{DFT}_n^{-1}(y)$ is given by

$$\begin{aligned}a_j &= \sum_{k=0}^{n-1} y_k \frac{\omega^{-jk}}{n} \\ &= \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega_n^{-kj} \qquad (30.11)\end{aligned}$$

for $j = 0, 1, \ldots, n-1$. By comparing equations (30.8) and (30.11), you can see that if you modify 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, you get the inverse DFT (see Exercise 30.2-4). Thus, DFT_n^{-1} is computable in $\Theta(n \lg n)$ time as well.

Thus, the FFT and the inverse FFT provide a way to transform a polynomial of degree-bound n back and forth between its coefficient representation and a point-value representation in only $\Theta(n \lg n)$ time. In the context of polynomial multiplication, we have shown the following about the convolution $a \otimes b$ of vectors a and b:

Theorem 30.8 (Convolution theorem)
For any two vectors a and b of length n, where n is an exact 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 an exact power of 3. Give a recurrence for the running time, and solve the recurrence.

★ ***30.2-6***
Instead of performing an n-element FFT over the field of complex numbers (where n is an exact power of 2), let's use the ring $\mathbb{Z}_m$ of integers modulo m, where $m = 2^{tn/2} + 1$ and t is an arbitrary positive integer. We can 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 $O(n \lg^2 n)$ time. (*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 for any complex number z in $O(n \lg n)$ time. (*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 FFT circuits

Many of the FFT's applications in signal processing require the utmost speed, and so the FFT is often implemented as a circuit in hardware. The FFT's divide-and-conquer structure enables the circuit to have a parallel structure so that the ***depth*** of the circuit—the maximum number of computational elements between any output and any input that can reach it—is $\Theta(\lg n)$. Moreover, the structure of the FFT circuit has several interesting mathematical properties, which we won't go into here.

Butterfly operations

Notice that the **for** loop of lines 9–12 of the FFT procedure computes the value $\omega_n^k\, y_k^{\text{odd}}$ twice per iteration: once in line 10 and once in line 11. A good optimizing compiler produces code that evaluates this ***common subexpression*** just once, storing its value into a temporary variable, so that lines 10–11 are treated like the three lines

$$\begin{aligned} t &= \omega\, y_k^{\text{odd}} \\ y_k &= y_k^{\text{even}} + t \\ y_{k+(n/2)} &= y_k^{\text{even}} - t \end{aligned}$$

This operation, multiplying the twiddle factor $\omega = \omega_n^k$ by y_k^{odd}, storing the product into the temporary variable t, and adding and subtracting t from y_k^{even}, is known as a ***butterfly operation***. Figure 30.3 shows it as a circuit, and you can see how it vaguely resembles the shape of a butterfly. (Although less colorfully, it could have been called a "bowtie" operation.)

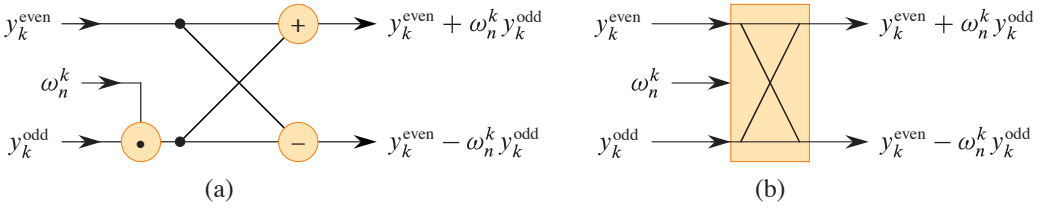


Figure 30.3 A circuit for a butterfly operation. **(a)** The two input values enter from the left, the twiddle factor ω_n^k is multiplied by y_k^{odd}, and the sum and difference are output on the right. **(b)** A simplified drawing of a butterfly operation, which we'll use when drawing the parallel FFT circuit.

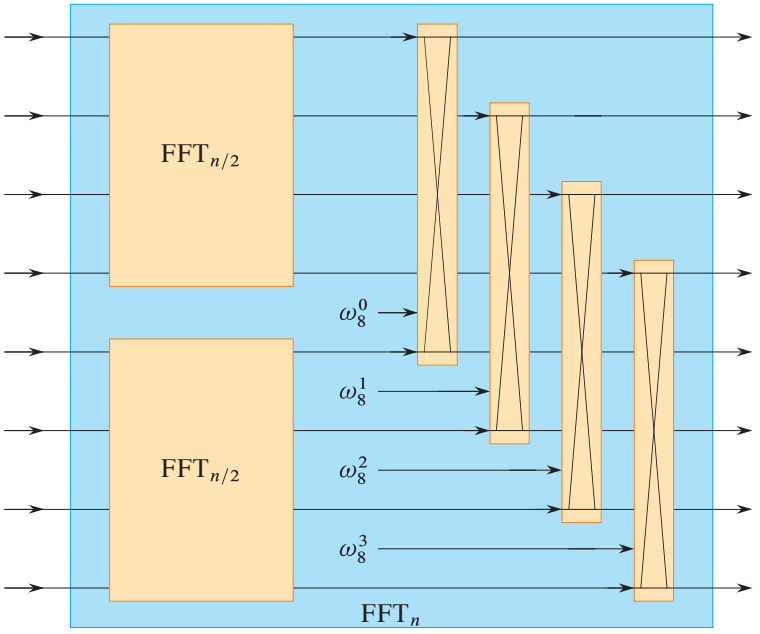


Figure 30.4 The schema for the conquer and combine steps of an n-input, n-output FFT circuit, FFT_n, shown for $n = 8$. Inputs enter from the left, and outputs exit from the right. The input values first go through two $\text{FFT}_{n/2}$ circuits, and then $n/2$ butterfly circuits combine the results. Only the top and bottom wires entering 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.

Recursive circuit structure

The FFT procedure follows the divide-and-conquer strategy that we first saw in Section 2.3.1:

Divide the n-element input vector into its $n/2$ even-indexed and $n/2$ odd-indexed elements.

Conquer by recursively computing the DFTs of the two subproblems, each of size $n/2$.

Combine by performing $n/2$ butterfly operations. These butterfly operations work with twiddle factors $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2-1}$.

The circuit schema in Figure 30.4 follows the conquer and combine steps of this pattern for an FFT circuit with n inputs and n outputs, denoted by FFT_n. Each line is a wire that carries a value. Inputs enter from the left, one per wire, and outputs exit from the right. The conquer step runs the inputs through two $\text{FFT}_{n/2}$ circuits, which are also constructed recursively. The values produced by the two $\text{FFT}_{n/2}$ circuits feed into $n/2$ butterfly circuits, with twiddle factors $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n/2} - 1$,

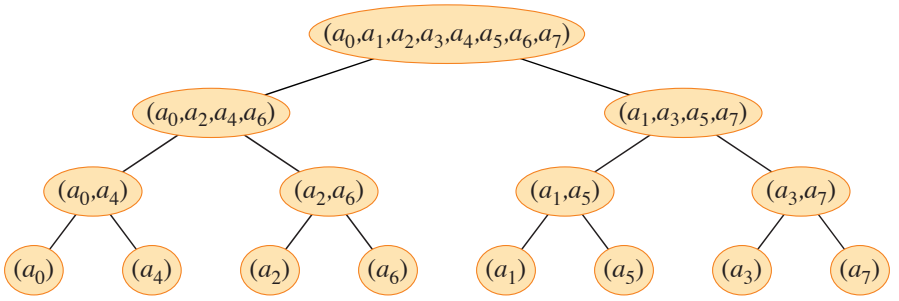


Figure 30.5 The tree of input vectors to the recursive calls of the FFT procedure. The initial invocation is for $n = 8$.

to combine the results. The base case of the recursion occurs when $n = 1$, where the sole output value equals the sole input value. An FFT_1 circuit, therefore, does nothing, and so the smallest nontrivial FFT circuit is FFT_2, a single butterfly operation whose twiddle factor is $\omega_2^0 = 1$.

Permuting the inputs

How does the divide step enter into the circuit design? Let's examine how input vectors to the various recursive calls of the FFT procedure relate to the original input vector, so that the circuit can emulate the divide step at the start for all levels of recursion. Figure 30.5 arranges the input vectors to the recursive calls in an invocation of 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 by the elements of the initial call as they appear in the corresponding input vector. Each 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, observe that if you arrange the elements of the initial vector a into the order in which they appear in the leaves, you can trace the execution of the FFT procedure, but bottom up instead of top down. First, 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 $n/2$ two-element DFTs. Next, 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 two-element DFTs with one four-element DFT. The vector then holds $n/4$ four-element DFTs. Continue in this manner until the vector holds two $(n/2)$-element DFTs, which $n/2$ butterfly operations combine into the final n-element DFT. In other words, you can start with the elements of the initial vector a, but rearranged as in the leaves of Figure 30.5, and then feed them directly into a circuit that follows the schema in Figure 30.4.

Let's think about the permutation that rearranges the input vector. The order in which the leaves appear in Figure 30.5 is a ***bit-reversal permutation***. That is, letting $\text{rev}(k)$ be the $\lg n$-bit integer formed by reversing the bits of the binary representation of k, then vector element a_k moves to position $\text{rev}(k)$. In Figure 30.5, 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 you can obtain it by reversing the bits of each number in the sequence $0, 1, 2, 3, 4, 6, 7$ or, in binary, $000, 001, 010, 011, 100, 101, 110, 111$. To see in general that the input vector should be rearranged by a bit-reversal permutation, 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, continue this process down the tree, until you get the order given by the bit-reversal permutation at the leaves.

The full FFT circuit

Figure 30.6 depicts the entire circuit for $n = 8$. The circuit begins with a bit-reversal permutation of the inputs, followed by $\lg n$ stages, each stage consisting of $n/2$ butterflies executed in parallel. Assuming that each butterfly circuit has constant depth, the full circuit has depth $\Theta(\lg n)$. The butterfly operations at each level of recursion in the FFT procedure are independent, and so the circuit performs them in parallel. The figure shows wires running from left to right, carrying values through the $\lg n$ stages. For $s = 1, 2, \ldots, \lg n$, stage s consists of $n/2^s$ groups of butterflies, with 2^{s-1} butterflies per group. The twiddle factors in stage s are $\omega_m^0, \omega_m^1, \ldots, \omega_m^{m/2-1}$, where $m = 2^s$.

Exercises

30.3-1

Show the values on the wires for each butterfly input and output in the FFT circuit of Figure 30.6, given the input vector $(0, 2, 3, -1, 4, 5, 7, 9)$.

30.3-2

Consider an FFT_n circuit, such as in Figure 30.6, with wires $0, 1, \ldots, n-1$ (wire j has output y_j) and stages numbered as in the figure. Stage s, for $s = 1, 2 \ldots, \lg n$, consists of $n/2^s$ groups of butterflies. Which two wires are inputs and outputs for the jth butterfly circuit in the gth group in stage s?

30.3-3

Consider a b-bit integer k in the range $0 \le k < 2^b$. Treating k as a b-element vector over $\{0, 1\}$, describe a $b \times b$ matrix M such that the matrix-vector product Mk is the binary representation of $\text{rev}(k)$.

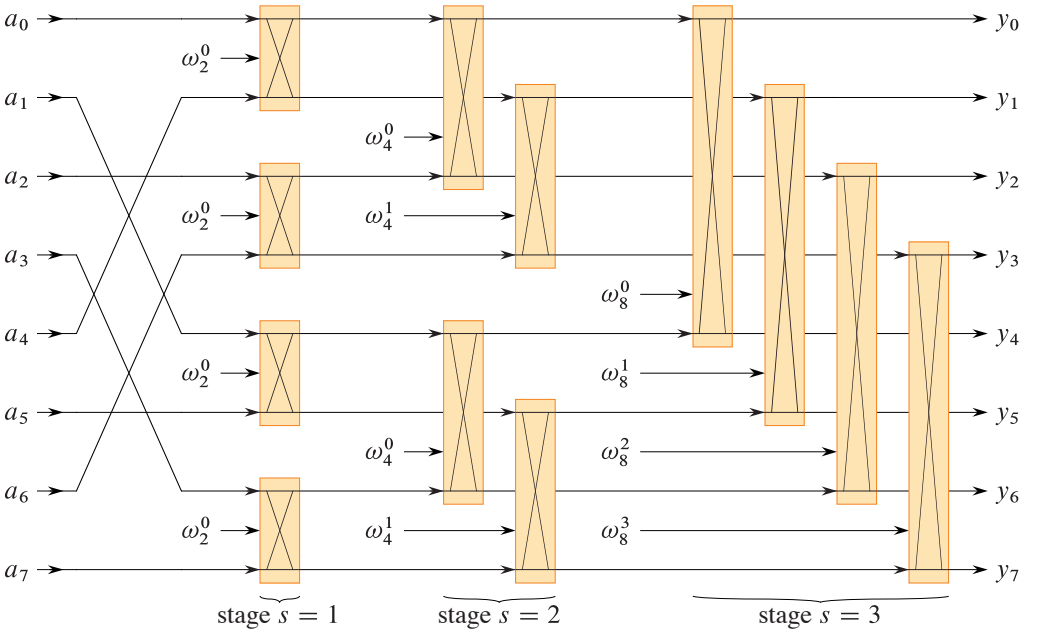


Figure 30.6 A full circuit that computes the FFT in parallel, here shown for $n = 8$ inputs. It has $\lg n$ stages, and each stage comprises $n/2$ butterflies that can operate in parallel. As in Figure 30.4, only the top and bottom wires entering a butterfly interact with it. For example, the top butterfly in stage 2 has inputs and outputs only on wires 0 and 2 (the wires with outputs y_0 and y_2, respectively). This circuit has depth $\Theta(\lg n)$ and performs $\Theta(n \lg n)$ butterfly operations altogether.

30.3-4

Write pseudocode for the procedure BIT-REVERSE-PERMUTATION(a, n), which performs the bit-reversal permutation on a vector a of length n in-place. Assume that you may call the procedure BIT-REVERSE-OF(k, b), which returns an integer that is the b-bit reversal of the nonnegative integer k, where $0 \le k < 2^b$.

★ *30.3-5*

Suppose that the adders within the butterfly operations of a given FFT circuit sometimes fail in such a manner that they always produce a 0 output, independent of their inputs. In addition, suppose that exactly one adder has failed, but 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 Multidimensional fast Fourier transform

The 1-dimensional discrete Fourier transform defined by equation (30.8) generalizes 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 $n_1 n_2 \cdots n_d = n$. The d-dimensional discrete Fourier transform is defined 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 how to produce a d-dimensional DFT by computing 1-dimensional DFTs on each dimension in turn. That is, 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, compute n/n_2 separate 1-dimensional DFTs along dimension 2. Using this result as the input, 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 if you compute the 1-dimensional DFTs in any order of the d dimensions, you compute the d-dimensional DFT.

c. Show that if you 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.

30-3 Evaluating all derivatives of a polynomial at a point

Given a polynomial $A(x)$ of degree-bound n, we define its tth derivative by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0 , \\ \frac{d}{dx} A^{(t-1)}(x) & \text{if } 1 \le t \le n-1 , \\ 0 & \text{if } t \ge n . \end{cases}$$

In this problem, you will show how to determine $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n-1$, given the coefficient representation $(a_0, a_1, \ldots, a_{n-1})$ of $A(x)$ and a point x_0.

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 you can evaluate all nontrivial derivatives of $A(x)$ at x_0 in $O(n \lg n)$ time.

30-4 Polynomial evaluation at multiple points

Problem 2-3 showed how to evaluate a polynomial of degree-bound n at a single point in $O(n)$ time using Horner's rule. This chapter described how to evaluate such a polynomial at all n complex roots of unity in $O(n \lg n)$ time using the FFT. Now, you will show how to evaluate a polynomial of degree-bound n at n arbitrary points in $O(n \lg^2 n)$ time.

To do so, assume that you can compute the polynomial remainder when one such polynomial is divided by another in $O(n \lg n)$ time. 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 $x_0, x_1, \ldots, x_{n-1}$, your goal is 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 both $Q_{ik}(x) = Q_{ij}(x) \bmod P_{ik}(x)$ and $Q_{kj}(x) = Q_{ij}(x) \bmod P_{kj}(x)$.

d. Give an $O(n \lg^2 n)$-time algorithm to evaluate $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

30-5 FFT using modular arithmetic

As defined, the discrete Fourier transform requires computation 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 a variant of the FFT based on modular arithmetic 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 explores another approach that uses a modulus of the more reasonable length $O(\lg n)$, but it requires that you understand the material of Chapter 31. Let n be an exact power of 2.

a. You wish to search for the smallest k such that $p = kn + 1$ is prime. Give a simple heuristic argument why you might expect k to be approximately $\ln n$. (The value of k might be much larger or smaller, but you 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 $O(n \lg n)$ time, 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 $(0, 5, 3, 7, 7, 2, 1, 6)$. (*Hint:* Verify and use the fact that $g = 3$ is a generator of $\mathbb{Z}_{17}^*$.)

Chapter notes

Van Loan's book [442] provides an outstanding treatment of the fast Fourier transform. Press, Teukolsky, Vetterling, and Flannery [365, 366] offer 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 [347] and Oppenheim and Willsky [348]. The Oppenheim and Schafer book also shows how to handle cases in which n is not an exact power of 2.

Fourier analysis is not limited to 1-dimensional data. It is widely used in image processing to analyze data in two or more dimensions. The books by Gonzalez and Woods [194] and Pratt [363] discuss multidimensional Fourier transforms and their use in image processing, and books by Tolimieri, An, and Lu [439] and Van Loan [442] discuss the mathematics of multidimensional fast Fourier transforms.

Cooley and Tukey [101] 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 [211] traces the history of the FFT as far back as C. F. Gauss in 1805.

Frigo and Johnson [161] 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. Moreover, FFTW has the 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 [449] 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 quickly, 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.

We start in Section 31.1 by introducing basic concepts of number theory, such as divisibility, modular equivalence, and unique prime factorization. Section 31.2 studies one of the world's oldest algorithms: Euclid's algorithm for computing the greatest common divisor of two integers, and Section 31.3 reviews concepts of modular arithmetic. Section 31.4 then explores the set of multiples of a given number a, modulo n, and shows how to find all solutions to the equation $ax = b \pmod{n}$ 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, such as the RSA public-key cryptosystem described in Section 31.7. We wrap up in Section 31.8, which examines a randomized primality test. This test finds large primes efficiently, an essential step in creating keys for the RSA cryptosystem.

Size of inputs and cost of arithmetic computations

Because we'll be working with large integers, we need to adjust how to 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, the size of an input depends on the *number of bits* required to represent that input, not just the number of integers in the input. An algorithm with integer in-

puts $a_1, a_2, \ldots, a_k$ 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.

Most of this book considers the elementary arithmetic operations (multiplications, divisions, or computing remainders) as primitive operations that take one unit of time. Counting the number of such arithmetic operations that an algorithm performs provides 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 appropriate 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, dividing a β-bit integer by a shorter integer or taking the remainder of a β-bit integer when divided by a shorter integer requires $\Theta(\beta^2)$ time 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 $O(\beta \lg \beta \lg\lg \beta)$ time is possible. For practical purposes, however, the $\Theta(\beta^2)$ algorithm is often best, and we use this bound as a basis for our analyses. In this chapter, we'll usually analyze algorithms 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 $d \ge 0$, then d is a ***divisor*** of a. Since $d \mid a$ if and only if $-d \mid a$, without loss of generality, we define the divisors of a 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 integers that are not multiples 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 [345]).

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, so that $n \mid a$ if and only if $a \bmod n = 0$.

The integers partition 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, $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$, and $[-4]_7$ and $[10]_7$ also denote this set. With the notation defined on page 64, writing $a \in [b]_n$ is the same as writing $a = b \pmod{n}$. 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 $-1 = n - 1 \pmod{n}$.

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. Any pair of integers has a common divisor of 1.

An important property of common divisors is that

$$\text{if } d \mid a \text{ and } d \mid b, \text{ then } d \mid (a+b) \text{ and } d \mid (a-b)\ . \tag{31.3}$$

More generally, for any integers x and y,

$$\text{if } d \mid a \text{ and } d \mid b, \text{ then } d \mid (ax+by)\ . \tag{31.4}$$

Also, if $a \mid b$, then either $|a| \le |b|$ or $b = 0$, which implies that

$$\text{if } a \mid b \text{ and } b \mid a, \text{ then } a = \pm b\ . \tag{31.5}$$

The ***greatest common divisor*** of two integers a and b which are not both 0, denoted by $\gcd(a,b)$, is the largest of the common divisors of a and 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, so that standard properties of the gcd function (such as equation (31.9) below) hold universally.

Exercise 31.1-9 asks you to prove the following 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 way to characterize $\gcd(a,b)$.

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 (3.11) on page 64 then implies

$$\begin{aligned} a \bmod s &= a - qs \\ &= a - q(ax + by) \\ &= a\,(1 - qx) + b\,(-qy), \end{aligned}$$

so that $a \bmod s$ is a linear combination of a and b as well. Because s is the smallest *positive* such linear combination and $0 \le a \bmod s < s$ (inequality (3.12) on page 64), $a \bmod s$ cannot be positive. Hence, $a \bmod s = 0$. Therefore, we have that $s \mid a$ and, by analogous reasoning, $s \mid b$. Thus, s is a common divisor of a and b, so that $\gcd(a, b) \ge s$. By definition, $\gcd(a, b)$ divides both a and b, and s is defined as a linear combination of a and b. Equation (31.4) therefore implies that $\gcd(a, b) \mid s$. 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, the smallest positive linear combination of a and b, is also their greatest common divisor. ■

Theorem 31.2 engenders three useful corollaries.

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) and Theorem 31.2, because $\gcd(a, b)$ is a linear combination of a and b, ■

Corollary 31.4
For all integers a and b and any nonnegative integer n, we have

$$\gcd(an, bn) = n \gcd(a, b) .$$

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 in turn 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 Exercise 31.1-5 asks you to provide the proof. ■

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, we have $\gcd(ab, p) = 1$ if and only if $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$ both hold.

Proof If $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, then it follows from Theorem 31.2 that there exist integers x, y, x', and y' such that

$$ax + py = 1 ,$$
$$bx' + py' = 1 .$$

Multiplying these equations and rearranging gives

$$ab(xx') + p(ybx' + y'ax + pyy') = 1 .$$

Since 1 is thus a positive linear combination of ab and p, it is the smallest positive linear combination. Applying Theorem 31.2 implies $\gcd(ab, p) = 1$, completing the proof in this direction.

Conversely, if $\gcd(ab, p) = 1$, then Theorem 31.2 implies that there exist integers x and y such that

$$abx + py = 1 .$$

Writing abx as $a(bx)$ and applying Theorem 31.2 again proves that $\gcd(a, p) = 1$. Proving that $\gcd(b, p) = 1$ is similar. ■

Integers $n_1, n_2, \ldots, n_k$ are ***pairwise relatively prime*** if $\gcd(n_i, n_j) = 1$ for $1 \le i < j \le k$.

Unique prime 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 $p \nmid b$. Because $p > 1$ and $ab = kp$ for some $k \in \mathbb{Z}$, equation (31.10) gives

that $\gcd(ab, p) = p$. We also have that $\gcd(a, p) = 1$ and $\gcd(b, p) = 1$, since the only divisors of p are 1 and p, and we assumed that p divides neither a nor b. Theorem 31.6 then implies that $\gcd(ab, p) = 1$, contradicting $\gcd(ab, p) = p$. This contradiction completes the proof. ■

A consequence of Theorem 31.7 is that any composite integer can be uniquely factored into a product of primes. Exercise 31.1-11 asks you to provide a proof.

Theorem 31.8 (Unique prime 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. ■

As an example, the unique prime factorization of the number 6000 is $2^4 \cdot 3^1 \cdot 5^3$.

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,

$$(a + b)^p = a^p + b^p \pmod{p} .$$

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

$x = y \pmod b$ implies $x = y \pmod a$

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, we have

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c) .$$

★ ***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 $\Theta(\beta^2)$ time.

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 $M(\beta)$ time, where $M(\beta) = \Omega(\beta)$, then you can convert binary to decimal in $O(M(\beta) \lg \beta)$ time. (*Hint:* Use a divide-and-conquer approach, obtaining the top and bottom halves of the result with separate recursions.)

31.1-14

Professor Marshall sets up n lightbulbs in a row. The lightbulbs all have switches, so that if he presses a bulb, it toggles on if it was off and off if it was on. The lightbulbs all start off. For $i = 1, 2, 3, \ldots, n$, the professor presses bulb $i, 2i, 3i, \ldots$. After the last press, which lightbulbs are on? Prove your answer.

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'll 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, for positive integers a and b, their prime factorizations suffice to compute $\gcd(a, b)$. 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 0 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,

$$\gcd(a, b) = p_1^{\min\{e_1, f_1\}} p_2^{\min\{e_2, f_2\}} \cdots p_r^{\min\{e_r, f_r\}} . \tag{31.13}$$

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 will show that $\gcd(a, b)$ and $\gcd(b, a \bmod b)$ divide each other. Since they are both nonnegative, equation (31.5) then implies that they must be equal.

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 (3.11) on page 64, $a \bmod b = a - qb$, 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 $d \mid \gcd(b, a \bmod b)$, that is,

$$\gcd(a, b) \mid \gcd(b, a \bmod b) . \tag{31.14}$$

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, so 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

Euclid's *Elements* (circa 300 B.C.E.) describes the following gcd algorithm, although its origin might be even earlier. The recursive procedure EUCLID implements Euclid's algorithm, 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)
```

For example, here is how the procedure computes $\gcd(30, 21)$:

$$\begin{aligned} \text{EUCLID}(30, 21) &= \text{EUCLID}(21, 9) \\ &= \text{EUCLID}(9, 3) \\ &= \text{EUCLID}(3, 0) \\ &= 3 \; . \end{aligned}$$

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 by equation (31.9), $\gcd(a, b) = \gcd(a, 0) = a$. 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

Let's analyze the worst-case running time of EUCLID as a function of the size of a and b. The overall running time of EUCLID is proportional to the number of recursive calls it makes. The analysis assumes that $a > b \geq 0$, that is, the first argument is greater than the second argument. Why? If $b = a > 0$, then $a \bmod b = 0$ and the procedure terminates after one recursive call. If $b > a \geq 0$,

then the procedure makes just one more recursive call than when $a > b$, because in this case EUCLID(a, b) immediately makes the recursive call EUCLID(b, a), and now the first argument is greater than the second.

Our analysis relies on the Fibonacci numbers F_k, defined by the recurrence equation (3.31) on page 69.

Lemma 31.10
If $a > b \geq 1$ and the call 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 base case 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.

Assuming inductively that the lemma holds if the procedure makes $k - 1$ recursive calls, we shall 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

$$\begin{aligned} b + (a \bmod b) &= b + (a - b \lfloor a/b \rfloor) \qquad \text{(by equation (3.11))} \\ &\leq a \ , \end{aligned}$$

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. ■

To show that the upper bound of Theorem 31.11 is the best possible, we'll show 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 $F_{k+1} \bmod F_k = F_{k-1}$. Because EUCLID$(a, b)$ calls EUCLID$(b, a \bmod b)$ when $b > 0$, 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, which meets the upper bound given by 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 (3.32) on page 69, the number of recursive calls in EUCLID is $O(\lg b)$. (See Exercise 31.2-5 for a tighter bound.) Therefore, a call of EUCLID on two β-bit numbers 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 prove an $O(\beta^2)$ bound on the number of bit operations.

The extended form of Euclid's algorithm

By rewriting Euclid's algorithm, we can gain additional useful information. Specifically, let's extend the algorithm to compute the integer coefficients x and y such that

$$d = \gcd(a, b) = ax + by \,, \tag{31.16}$$

where either or both of x and y may be zero or negative. These coefficients will prove 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). As an example, Figure 31.1 traces out the call EXTENDED-EUCLID$(99, 78)$.

```
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)
```

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 in the EUCLID procedure, we have $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$. To obtain x and y such that $d = ax + by$, let's rewrite equation (31.17), setting

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 $\gcd(99, 78) = 3 = 99 \cdot (-11) + 78 \cdot 14$.

$d = d'$ and using equation (3.11):

$$\begin{aligned} d &= bx' + (a - b\lfloor a/b \rfloor)y' \\ &= ay' + b(x' - \lfloor a/b \rfloor y') \, . \end{aligned}$$

Thus, choosing $x = y'$ and $y = x' - \lfloor a/b \rfloor y'$ satisfies the equation $d = ax + by$, thereby 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,

$$\gcd(a, n) = \gcd(a + kn, n) \, . \tag{31.18}$$

Use equation (31.18) to show that $a = 1 \pmod{n}$ implies $\gcd(a, n) = 1$.

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
The ***least common multiple*** $\text{lcm}(a_1, a_2, \ldots, a_n)$ of integers $a_1, a_2, \ldots, a_n$ is the smallest nonnegative integer that is a multiple of each a_i. Show how to compute $\text{lcm}(a_1, a_2, \ldots, a_n)$ 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, you can think of modular arithmetic as arithmetic as usual over the integers, except that when 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 (so that x is replaced by $x \bmod n$). This informal model suffices if you stick to the operations of addition, subtraction, and multiplication. A more formal model for modular arithmetic, which follows, 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 $-a$. An ***abelian group*** $(S, \oplus)$ satisfies the ***commutative law*** $a \oplus b = b \oplus a$ for all $a, b \in S$. The ***size*** of group $(S, \oplus)$ is $|S|$, and if $|S| < \infty$, then $(S, \oplus)$ 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 suitable binary operations, which we obtain by redefining the ordinary operations of addition and multiplication. We can 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 $a = a' \pmod{n}$ and $b = b' \pmod{n}$, then

$$\begin{aligned} a + b &= a' + b' \pmod{n}, \\ ab &= a'b' \pmod{n}. \end{aligned}$$

Thus, we define addition and multiplication modulo n, denoted $+_n$ and $\cdot_n$, by

$$\begin{aligned} [a]_n +_n [b]_n &= [a + b]_n, \\ [a]_n \cdot_n [b]_n &= [ab]_n. \end{aligned} \tag{31.19}$$

(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'll 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 $x \bmod n$.

$+_6$	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 $(\mathbb{Z}_{15}^*, \cdot_{15})$.

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 (31.19) shows that $(\mathbb{Z}_n, +_n)$ is closed. Associativity and commutativity of $+_n$ follow from the associativity and commutativity of $+$:

$$\begin{aligned}
([a]_n +_n [b]_n) +_n [c]_n &= [a + b]_n +_n [c]_n \\
&= [(a + b) + c]_n \\
&= [a + (b + c)]_n \\
&= [a]_n +_n [b + c]_n \\
&= [a]_n +_n ([b]_n +_n [c]_n) ,
\end{aligned}$$

$$\begin{aligned}
[a]_n +_n [b]_n &= [a + b]_n \\
&= [b + a]_n \\
&= [b]_n +_n [a]_n .
\end{aligned}$$

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 $a = (a + kn) \pmod{n}$ 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. (We have denoted an element $[a]_{15}$ as a, and thus, for example, we denote $[7]_{15}$ as 7.) Figure 31.2(b) shows the group $(\mathbb{Z}_{15}^*, \cdot_{15})$. For example, $8 \cdot 11 = 13 \pmod{15}$, 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 we have $d = 1$, since $a \in \mathbb{Z}_n^*$, and

$$ax + ny = 1 \ , \tag{31.20}$$

or equivalently,

$$ax = 1 \pmod{n} \ .$$

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 (31.20) 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 in Section 31.4. ■

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. Furthermore, equivalences modulo n may also be interpreted as equations in $\mathbb{Z}_n$. For example, the following two statements are equivalent:

$$ax = b \pmod{n}$$

and

$$[a]_n \cdot_n [x]_n = [b]_n \ .$$

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 just $\mathbb{Z}_n$ 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 $a/b = ab^{-1} \pmod{n}$. For example, in $\mathbb{Z}_{15}^*$ we have that $7^{-1} = 13 \pmod{15}$, since $7 \cdot 13 = 91 = 1 \pmod{15}$, so that $2/7 = 2 \cdot 13 = 11 \pmod{15}$.

The size of $\mathbb{Z}_n^*$ is denoted $\phi(n)$. This function, known as ***Euler's phi function***, satisfies the equation

$$\phi(n) = n \prod_{p \text{ prime such that } p \mid n} \left(1 - \frac{1}{p}\right) \ , \tag{31.21}$$

so that p runs over all the primes dividing n (including n itself, if n is prime). We won't prove this formula here. Intuitively, 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

$$\begin{aligned} \phi(p) &= p\left(1 - \frac{1}{p}\right) \\ &= p - 1 \ . \end{aligned} \tag{31.22}$$

If n is composite, then $\phi(n) < n - 1$, although it can be shown that

$$\phi(n) > \frac{n}{e^\gamma \ln\ln n + 3/\ln\ln n} \tag{31.23}$$

for $n \geq 3$, where $\gamma = 0.5772156649\ldots$ is ***Euler's constant***. A somewhat simpler (but looser) lower bound for $n > 5$ is

$$\phi(n) > \frac{n}{6 \ln \ln n} . \tag{31.24}$$

The lower bound (31.23) is essentially the best possible, since

$$\liminf_{n \to \infty} \frac{\phi(n)}{n / \ln \ln n} = e^{-\gamma} . \tag{31.25}$$

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, whose proof we leave as Exercise 31.3-3, 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 $(S, \oplus)$. ■

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, whose proof is omitted, provides an extremely useful constraint on the size of a subgroup.

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'll use the following corollary in the 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 affords us a straightforward 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, taking $a = 2$ in the group $\mathbb{Z}_6$ yields the sequence

$$a^{(1)}, a^{(2)}, a^{(3)}, \ldots = 2, 4, 0, 2, 4, 0, 2, 4, 0, \ldots .$$

We have $a^{(k)} = ka \bmod n$ in the group $\mathbb{Z}_n$, and $a^{(k)} = a^k \bmod n$ in the group $\mathbb{Z}_n^*$. We define the ***subgroup generated by a***, denoted $\langle a \rangle$ or $(\langle a \rangle, \oplus)$, by

$$\langle a \rangle = \left\{ a^{(k)} : k \geq 1 \right\} .$$

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 $\mathrm{ord}(a)$, is defined as the smallest positive integer t such that $a^{(t)} = e$. (Recall that $e \in S$ is the group identity.)

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 $\mathrm{ord}(a) = |\langle a \rangle|$.

Proof Let $t = \mathrm{ord}(a)$. 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 $j < i$. Therefore, as we generate elements by a, we see no new elements after $a^{(t)}$. Thus, $\langle a \rangle = \left\{ a^{(1)}, a^{(2)}, \ldots, a^{(t)} \right\}$, 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 equation 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 $i = j \pmod{t}$. ■

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 $\text{ord}(a) \mid |S|$, and so $|S| = 0 \pmod{t}$, where $t = \text{ord}(a)$. 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 f between $\mathbb{Z}_4$ and $\mathbb{Z}_5^*$ such that $a + b = c \pmod{4}$ if and only if $f(a) \cdot f(b) = f(c) \pmod{5}$.

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

$$ax = b \pmod{n}, \tag{31.26}$$

where $a > 0$ and $n > 0$. This problem has several applications. For example, we'll use it in Section 31.7 as part of the procedure to find keys in the RSA public-key cryptosystem. We assume that a, b, and n are given, and we wish to find all values of x, modulo n, that satisfy equation (31.26). 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 (31.26) 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 we have

$$\begin{aligned}\langle a \rangle &= \langle d \rangle \\ &= \{0, d, 2d, \ldots, ((n/d) - 1)d\}\end{aligned}$$

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) returns a triple (d, x, y) such that $ax + ny = d$. Thus, $ax = d \pmod{n}$, 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. Because $d = \gcd(a, n)$, we know that $d \mid a$ 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 $ax = b \pmod{n}$ is solvable for the unknown x if and only if $d \mid b$, where $d = \gcd(a, n)$.

Proof The equation $ax = b \pmod{n}$ 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 $ax = b \pmod{n}$ either has d distinct solutions modulo n, where $d = \gcd(a, n)$, or it has no solutions.

Proof If $ax = b \pmod{n}$ 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 $ax = b \pmod{n}$. ■

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 $ax = b \pmod{n}$ has as one of its solutions the value x_0, where

$$x_0 = x'(b/d) \bmod n .$$

Proof We have

$$\begin{aligned} ax_0 &= ax'(b/d) \pmod{n} \\ &= d(b/d) \pmod{n} \qquad (\text{because } ax' = d \pmod{n}) \\ &= b \pmod{n} , \end{aligned}$$

and thus x_0 is a solution to $ax = b \pmod{n}$. ■

Theorem 31.24
Suppose that the equation $ax = b \pmod{n}$ 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 $ax = b \pmod{n}$, we have $ax_0 \bmod n = b \pmod{n}$. Thus, for $i = 0, 1, \ldots, d-1$, we have

$$\begin{aligned} ax_i \bmod n &= a(x_0 + in/d) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \quad \text{(because } d \mid a \text{ implies that } ain/d \text{ is a multiple of } n\text{)} \\ &= b \pmod{n}, \end{aligned}$$

and hence $ax_i = b \pmod{n}$, making x_i a solution, too. By Corollary 31.22, the equation $ax = b \pmod{n}$ 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 $ax = b \pmod{n}$. The procedure MODULAR-LINEAR-EQUATION-SOLVER 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 MODULAR-LINEAR-EQUATION-SOLVER, consider the equation $14x = 30 \pmod{100}$ (and thus $a = 14$, $b = 30$, and $n = 100$). Calling EXTENDED-EUCLID in line 1 gives $(d, x', y') = (2, -7, 1)$. Since $2 \mid 30$, lines 3–5 execute. Line 3 computes $x_0 = (-7)(15) \bmod 100 = 95$. The **for** loop of lines 4–5 prints the two solutions, 95 and 45.

The procedure MODULAR-LINEAR-EQUATION-SOLVER works as follows. The call to EXTENDED-EUCLID in line 1 returns a triple (d, x', y') such that $d = \gcd(a, n)$ and $d = ax' + ny'$. Therefore, x' is a solution to the equation $ax' = d \pmod{n}$. If d does not divide b, then the equation $ax = b \pmod{n}$ has no solution, by Corollary 31.21. Line 2 checks to see whether $d \mid b$, and if not, line 6 reports that there are no solutions. Otherwise, line 3 computes a solution x_0 to $ax = b \pmod{n}$, as Theorem 31.23 suggests. 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 $ax = b \pmod{n}$ has a unique solution, modulo n. ■

If $b = 1$, a common case of considerable interest, the x that solves the equation is a ***multiplicative inverse*** of a, modulo n.

Corollary 31.26

For any $n > 1$, if $\gcd(a, n) = 1$, then the equation $ax = 1 \pmod{n}$ has a unique solution, modulo n. Otherwise, it has no solution. ■

Thanks to Corollary 31.26, the notation $a^{-1} \bmod n$ refers 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 $ax = 1 \pmod{n}$ is the integer x returned by EXTENDED-EUCLID, since the equation

$$\gcd(a, n) = 1 = ax + ny$$

implies $ax = 1 \pmod{n}$. Thus, EXTENDED-EUCLID can compute $a^{-1} \bmod n$ efficiently.

Exercises

31.4-1

Find all solutions to the equation $35x = 10 \pmod{50}$.

31.4-2

Prove that the equation $ax = ay \pmod{n}$ implies $x = y \pmod{n}$ 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 $x_0 = x'(b/d) \bmod (n/d)$

With this change, will the procedure still work? Explain why or why not.

★ ***31.4-4***
Let p be prime and $f(x) = (f_0 + f_1 x + \cdots + f_t x^t) \pmod p$ 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 $f(a) = 0 \pmod p$. Prove that if a is a zero of f, then $f(x) = (x-a)g(x)(\bmod p)$ 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 100 C.E., 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 $x = 23$, and 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 in designing 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 mapping (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

$$a \leftrightarrow (a_1, a_2, \ldots, a_k) ,$$
$$b \leftrightarrow (b_1, b_2, \ldots, b_k) ,$$

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 Let's see how to translate between the two representations. Going from a to $(a_1, a_2, \ldots, a_k)$ requires only k "mod" operations. The reverse—computing a from inputs $(a_1, a_2, \ldots, a_k)$—is only slightly 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 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. Here is how to compute a as a function of the a_i and c_i:

$$a = (a_1 c_1 + a_2 c_2 + \cdots + a_k c_k) \pmod{n} . \tag{31.32}$$

We now show that equation (31.32) ensures that $a = a_i \pmod{n_i}$ for $i = 1, 2, \ldots, k$. If $j \neq i$, then $m_j = 0 \pmod{n_i}$, which implies that $c_j = m_j = 0 \pmod{n_i}$. Note also that $c_i = 1 \pmod{n_i}$, 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

$$\begin{aligned} a &= a_i c_i && \pmod{n_i} \\ &= a_i m_i (m_i^{-1} \bmod n_i) && \pmod{n_i} \\ &= a_i && \pmod{n_i} , \end{aligned}$$

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 $a = a_i \pmod{n_i}$ 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'll use the following corollaries later in this chapter.

Corollary 31.28
If $n_1, n_2, \ldots, n_k$ are pairwise relatively prime and $n = n_1 n_2 \cdots n_k$, then for any integers $a_1, a_2, \ldots, a_k$, the set of simultaneous equations

$$x = a_i \pmod{n_i} ,$$

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 $n = n_1 n_2 \cdots n_k$, then for all integers x and a,

$$x = a \pmod{n_i}$$

for $i = 1, 2, \ldots, k$ if and only if

$$x = a \pmod{n} .$$ ■

As an example of the application of the Chinese remainder theorem, suppose that you are given the two equations

$$a = 2 \pmod{5} ,$$
$$a = 3 \pmod{13} ,$$

so that $a_1 = 2$, $n_1 = m_2 = 5$, $a_2 = 3$, and $n_2 = m_1 = 13$, and you wish to compute $a \bmod 65$, since $n = n_1 n_2 = 65$. Because $13^{-1} = 2 \pmod{5}$ and $5^{-1} = 8 \pmod{13}$, you compute

$$c_1 = 13 \cdot (2 \bmod 5) = 26 ,$$
$$c_2 = 5 \cdot (8 \bmod 13) = 40 ,$$

and

$$\begin{aligned} a &= 2 \cdot 26 + 3 \cdot 40 \pmod{65} \\ &= 52 + 120 \pmod{65} \\ &= 42 \pmod{65} . \end{aligned}$$

	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.

See Figure 31.3 for an illustration of the Chinese remainder theorem, modulo 65.

Thus, you 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 $x = 4 \pmod 5$ and $x = 5 \pmod{11}$.

31.5-2

Find all integers x that leave remainders 1, 2, and 3 when divided by 9, 8, and 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 $f(x) = 0 \pmod n$ equals the product of the number of roots of each of the equations $f(x) = 0 \pmod{n_1}$, $f(x) = 0 \pmod{n_2}$, $\ldots$, $f(x) = 0 \pmod{n_k}$.

31.6 Powers of an element

Along with considering the multiples of a given element a, modulo n, we often consider the sequence of powers of a, modulo n, where $a \in \mathbb{Z}_n^*$:

$$a^0, a^1, a^2, a^3, \ldots,$$

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	$\cdots$

and 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 through 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 $n > 1$,

$$a^{\phi(n)} = 1 \pmod{n} \text{ for all } a \in \mathbb{Z}_n^* .$$

■

Theorem 31.31 (Fermat's theorem)
If p is prime, then

$$a^{p-1} = 1 \pmod{p} \text{ for all } a \in \mathbb{Z}_p^* .$$

Proof By equation (31.22), $\phi(p) = p - 1$ 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 $a^p = a \pmod{p}$ if p is prime.

If $\mathrm{ord}_n(g) = |\mathbb{Z}_n^*|$, 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 [345].

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 $g^z = a \pmod{n}$. 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 $g^x = g^y \pmod{n}$ holds if and only if the equation $x = y \pmod{\phi(n)}$ holds.

Proof Suppose first that $x = y \pmod{\phi(n)}$. Then, we have $x = y + k\phi(n)$ for some integer k, and thus

$$\begin{aligned} g^x &= g^{y+k\phi(n)} && \pmod{n} \\ &= g^y \cdot (g^{\phi(n)})^k && \pmod{n} \\ &= g^y \cdot 1^k && \pmod{n} \qquad \text{(by Euler's theorem)} \\ &= g^y && \pmod{n} . \end{aligned}$$

Conversely, suppose that $g^x = g^y \pmod{n}$. 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 $g^x = g^y \pmod{n}$, we must have $x = y \pmod{\phi(n)}$. ■

Let's now turn our attention to the square roots of 1, modulo a prime power. The following properties will be useful to justify the primality-testing algorithm in Section 31.8.

Theorem 31.34
If p is an odd prime and $e \geq 1$, then the equation

$$x^2 = 1 \pmod{p^e} \tag{31.33}$$

has only two solutions, namely $x = 1$ and $x = -1$.

Proof By Exercise 31.6-2, equation (31.33) 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, $x = -1 \pmod{p^e}$. 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 $x = 1 \pmod{p^e}$. Therefore, either $x = -1 \pmod{p^e}$ or $x = 1 \pmod{p^e}$. ■

A number x is a ***nontrivial square root of 1, modulo n***, if it satisfies the equation $x^2 = 1 \pmod{n}$ 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'll use the following corollary to Theorem 31.34 in Section 31.8 to prove the Miller-Rabin primality-testing procedure correct.

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. Nor can n be 2, because if $x^2 = 1 \pmod 2$, then $x = 1 \pmod 2$, and therefore, 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.

Repeated squaring is based on the following formula to compute a^b for nonnegative integers a and b:

$$a^b = \begin{cases} 1 & \text{if } b = 0\,, \\ (a^{b/2})^2 & \text{if } b > 0 \text{ and } b \text{ is even}\,, \\ a \cdot a^{b-1} & \text{if } b > 0 \text{ and } b \text{ is odd}\,. \end{cases} \tag{31.34}$$

The last case, where b is odd, reduces to the one of the first two cases, since if b is odd, then $b-1$ is even. The recursive procedure MODULAR-EXPONENTIATION on the next page computes $a^b \bmod n$ using equation (31.34), but performing all computations modulo n. The term "repeated squaring" comes from squaring the intermediate result $d = a^{b/2}$ in line 5. Figure 31.4 shows the values of the parameter b, the local variable d, and the value returned at each level of the recursion for the call MODULAR-EXPONENTIATION$(7, 560, 561)$, which returns the result 1.

b	560	280	140	70	35	34	17	16			8		4	2	1	0
d	67	166	298	241	355	160	103	526	157	49	7	1				–
returned value	1	67	166	298	241	355	160	103	526	157	49	7	1			

Figure 31.4 The values of the parameter b, the local variable d, and the value returned for recursive calls of MODULAR-EXPONENTIATION with parameter values $a = 7$, $b = 560$, and $n = 561$. The value returned by each recursive call is assigned directly to d. The result of the call with $a = 7$, $b = 560$, and $n = 561$ is 1.

```
MODULAR-EXPONENTIATION(a, b, n)
1  if b == 0
2      return 1
3  elseif b mod 2 == 0
4      d = MODULAR-EXPONENTIATION(a, b/2, n)     // b is even
5      return (d · d) mod n
6  else d = MODULAR-EXPONENTIATION(a, b − 1, n)  // b is odd
7      return (a · d) mod n
```

The total number of recursive calls depends on the number of bits of b and the values of these bits. Assume that $b > 0$ and that the most significant bit of b is a 1. Each 0 generates one recursive call (in line 4), and each 1 generates two recursive calls (one in line 6 followed by one in line 4 because if b is odd, then $b - 1$ is even). If the inputs a, b, and n are β-bit numbers, then there are between β and $2\beta - 1$ recursive calls altogether, 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

Show that $x^2 = 1 \pmod{p^e}$ is equivalent to $p^e \mid (x-1)(x+1)$.

31.6-3

Rewrite the third case of MODULAR-EXPONENTIATION, where b is odd, so that if b has β bits and the most significant bit is 1, then there are always exactly β recursive calls.

31.6-4
Give a nonrecursive (i.e., iterative) version of MODULAR-EXPONENTIATION.

31.6-5
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, you can ***encrypt*** messages sent between two communicating parties so that an eavesdropper who overhears the encrypted messages will not be able to decode, or ***decrypt***, 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.

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 the public keys for Alice and Bob as P_A and P_B, respectively, and likewise the secret keys are S_A for Alice and 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.

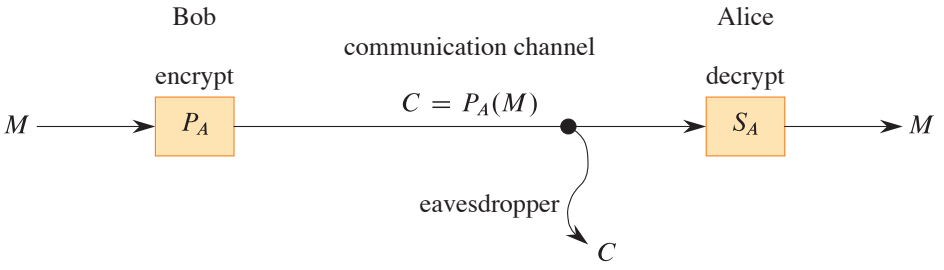


Figure 31.5 Encryption in a public key system. Bob encrypts the message M using Alice's public key P_A and transmits the resulting ciphertext $C = P_A(M)$ over a communication channel to Alice. An eavesdropper who captures the transmitted ciphertext gains no information about M. Alice receives C and decrypts it using her secret key to obtain the original message $M = S_A(C)$.

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. The simplest, and original, formulation of public-key cryptography requires one-to-one functions from $\mathcal{D}$ to itself, based on the public and secret keys. We denote the function based on Alice's public key P_A by $P_A()$ and the function based on 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 keys P_A and 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 back the original message M.

A public-key cryptosystem requires that Alice, and only Alice, be able to compute the function $S_A()$ in any practical amount of time. This assumption is crucial to keeping encrypted messages sent to Alice private and to knowing that Alice's digital signatures are authentic. Alice must keep her key S_A secret. If she does not, whoever else has access to S_A can decrypt messages intended only for Alice and can also forge her digital signature. The assumption that only Alice can reasonably compute $S_A()$ must hold even though everyone knows P_A and can compute $P_A()$, the inverse function to $S_A()$, efficiently. These requirements appear formidable, but we'll see how to satisfy them.

In a public-key cryptosystem, encryption works as shown in Figure 31.5. Suppose that Bob wishes to send Alice a message M encrypted so that it looks like

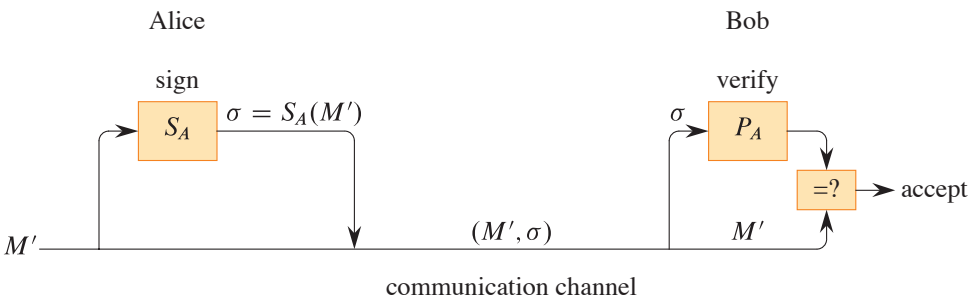


Figure 31.6 Digital signatures in a public-key system. Alice signs the message M' by appending her digital signature $\sigma = S_A(M')$ to it. She transmits the message/signature pair (M', σ) to Bob, who verifies it by checking the equation $M' = P_A(\sigma)$. If the equation holds, he accepts (M', σ) as a message that Alice has signed.

unintelligible gibberish to an eavesdropper. The scenario for sending the message goes as follows.

- Bob obtains Alice's public key P_A, perhaps from a public directory or perhaps 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 $S_A()$, only Alice can compute M from C. Because Bob encrypts M using $P_A()$, only Alice can understand the transmitted message.

Digital signatures can be implemented within this formulation of a public-key cryptosystem. (There are other ways to construct digital signatures, but we won't 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 $\sigma = S_A(M')$.
- 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 that 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 information he received 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 may or may not be encrypted. The message can be "in the clear" and not protected from disclosure. By composing the above protocols for encryption and for signatures, Alice can create a message to Bob that is both signed and encrypted. Alice first appends her digital signature to the message and then encrypts the resulting message/signature pair with Bob's public key. Bob decrypts the received message with his secret key to obtain both the original message and its digital signature. Bob can then verify the signature using Alice's public key. 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 a public key and a secret key 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 (31.21), 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. You 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's secret key is applied to the message to be signed, rather than to a ciphertext. To verify a signature, the public key of the signer is applied to the signature rather than to a message to be encrypted.

To implement the public-key and secret-key operations (31.37) and (31.38), you can use 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

$$\begin{aligned} M^{ed} &= M(M^{p-1})^{k(q-1)} && \pmod{p} \\ &= M((M \bmod p)^{p-1})^{k(q-1)} && \pmod{p} \\ &= M(1)^{k(q-1)} && \pmod{p} \qquad \text{(by Theorem 31.31)} \\ &= M && \pmod{p} \ . \end{aligned}$$

Also, $M^{ed} = M \pmod{p}$ if $M = 0 \pmod{p}$. Thus,

$$M^{ed} = M \pmod{p}$$

for all M. Similarly,

$$M^{ed} = M \pmod{q}$$

for all M. Thus, by Corollary 31.29 to the Chinese remainder theorem,

$$M^{ed} = M \pmod{n}$$

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 factoring large integers is surprisingly difficult. By randomly selecting and multiplying together two 1024-bit primes, you 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, you should use integers that are quite long—more than 1000 bits—to resist possible advances in the art of factoring. In 2021, RSA moduli are commonly in the range of 2048 to 4096 bits. To create moduli of such sizes, you must 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 cryptosystems that are not public-key cryptosystems. With such a ***symmetric-key*** 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 symmetric-key cryptosystem and encrypts M using K, obtaining ciphertext C, where 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.

A similar hybrid approach creates 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.

One way to distribute public keys uses ***certificates***. For example, assume that 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 might be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [327].)

★ ***31.7-3***

Prove that RSA is multiplicative in the sense that

$$P_A(M_1)P_A(M_2) = P_A(M_1M_2) \pmod{n} .$$

Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1% of messages from $\mathbb{Z}_n$ encrypted with P_A, then the adversary could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

★ 31.8 Primality testing

This section shows how to find 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

Many applications, such as cryptography, call for finding large "random" primes. Fortunately, large primes are not too rare, so that it is feasible to test random integers of the appropriate size until you find one that is 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.)

The process of randomly selecting an integer n and determining whether it is prime is really just 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 says how many trials must occur to obtain a success, and by equation (C.36) on page 1197, the expected number of trials is approximately $\ln n$. Thus, in order to find a prime that has the same length as n by testing integers chosen randomly near n, the expected number examined would be approximately $\ln n$. For example, the expectation is that finding a 1024-bit prime would require testing approximately $\ln 2^{1024} \approx 710$ randomly chosen 1024-bit numbers for primality. (Of course, to cut this figure in half, choose only odd integers.)

The remainder of this section shows how to determine whether a large odd integer n is prime. For notational convenience, we assume that n has the prime factorization

$$n = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r} \,,$$

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***: try dividing n by each integer $2, 3, 5, 7, 9, \ldots, \lfloor \sqrt{n} \rfloor$, skipping even integers greater

than 2. We can conclude 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, it also determines one of n's prime factors if n is composite.

This section focuses on finding out whether a given number n is prime. If n is composite, we won't worry about finding its prime factorization. Computing the prime factorization of a number is computationally expensive. You might be surprised that it turns out to be much easier to ascertain whether a given number is prime than it is to determine the prime factorization of the number if it is not prime.

Pseudoprimality testing

We'll start with a method for primality testing that "almost works" and, in fact, is good enough for many practical applications. Later on, we'll refine this method to remove 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

$$a^{n-1} = 1 \pmod{n} . \tag{31.39}$$

Fermat's theorem (Theorem 31.31 on page 932) implies that if n is prime, then n satisfies equation (31.39) for every a in $\mathbb{Z}_n^+$. Thus, if there is any $a \in \mathbb{Z}_n^+$ such that n does *not* satisfy equation (31.39), then n is certainly composite. Surprisingly, the converse *almost* holds, so that this criterion forms an almost perfect test for primality. Instead of trying every value of $a \in \mathbb{Z}_n^+$, test to see whether n satisfies equation (31.39) for just $a = 2$. If not, then declare n to be composite by returning COMPOSITE. Otherwise, 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 procedure PSEUDOPRIME on the next page pretends in this manner to check whether n is prime. It uses the procedure MODULAR-EXPONENTIATION from Section 31.6. It assumes that the input n is an odd integer greater than 2. 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 PSEUDOPRIME err? Surprisingly rarely. There are only 22 values of n less than 10,000 for which it errs, the first four of which are 341, 561, 645,

```
PSEUDOPRIME(n)
1  if MODULAR-EXPONENTIATION(2, n − 1, n) ≠ 1 (mod n)
2       return COMPOSITE        // definitely
3  else return PRIME            // we hope!
```

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 0 as β approaches ∞. Using more precise estimates due to Pomerance [361] of the number of base-2 pseudoprimes of a given size, a randomly chosen 512-bit number that is called prime by PSEUDOPRIME 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. Thus, 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, you might need a better approach for testing primality. As we'll see, a little more cleverness, and some randomization, will yield a primality-testing method that works well on all inputs.

Since PSEUDOPRIME checks equation (31.39) for only $a = 2$, you might think that you could eliminate all the errors by simply checking equation (31.39) for a second base number, say $a = 3$. Better yet, you could check equation (31.39) for even more values of a. Unfortunately, even checking for several values of a does not eliminate all errors, because there exist composite integers n, known as ***Carmichael numbers***, that satisfy equation (31.39) for *all* $a \in \mathbb{Z}_n^*$. (The equation does fail when $\gcd(a, n) > 1$—that is, when $a \notin \mathbb{Z}_n^*$—but demonstrating 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. For example, only 255 of them are less than 100,000,000. Exercise 31.8-2 helps explain why they are so rare.

Let's see how to improve the primality test so that Carmichael numbers won't fool it.

The Miller-Rabin randomized primality test

The Miller-Rabin primality test overcomes the problems of the simple procedure 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 appears in the procedures MILLER-RABIN and WITNESS. The input $n > 2$ to MILLER-RABIN 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 129: RANDOM$(2, n-2)$ returns a randomly chosen integer a satisfying $2 \le a \le n-2$. (This range of values avoids having $a = \pm 1 \pmod{n}$.) The call of the auxiliary procedure WITNESS(a, n) returns 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 will see) that n is composite. The test WITNESS(a, n) is an extension of, but more effective than, the test in equation (31.39) that formed the basis for PSEUDOPRIME, using $a = 2$.

Let's first understand how WITNESS works, and then we'll see how the Miller-Rabin primality test uses it. Let $n - 1 = 2^t u$ where $t \ge 1$ and u is odd. That is, the binary representation of $n-1$ is the binary representation of the odd integer u followed by exactly t zeros. Therefore, $a^{n-1} = (a^u)^{2^t} \pmod{n}$, so that one way to compute $a^{n-1} \bmod n$ is to first compute $a^u \bmod n$ and then square the result t times successively.

```
MILLER-RABIN(n, s)                    // n > 2 is odd
1  for j = 1 to s
2      a = RANDOM(2, n − 2)
3      if WITNESS(a, n)
4          return COMPOSITE           // definitely
5  return PRIME                       // almost surely

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                // found a nontrivial square root of 1
7  if x_t ≠ 1
8      return TRUE                    // composite, as in PSEUDOPRIME
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 repeatedly squaring the result t times 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 $x_i = a^{2^i u} \pmod n$ for $i = 0, 1, \ldots, t$, so that in particular $x_t = a^{n-1} \pmod n$. After line 4 performs a squaring step, however, the loop will terminate early if lines 5–6 detect that a nontrivial square root of 1 has just been discovered. (We'll explain these tests shortly.) If so, the procedure stops and returns TRUE. Lines 7–8 return TRUE if the value computed for $x_t = a^{n-1} \pmod n$ is not equal to 1, just as the PSEUDOPRIME procedure returns COMPOSITE in this case. Line 9 returns FALSE if lines 6 or 8 have not returned TRUE.

The following lemma proves the correctness of WITNESS.

Lemma 31.38
If WITNESS(a, n) returns TRUE, then a proof that n is composite can be constructed using a as a witness.

Proof If WITNESS returns TRUE from line 8, it's because line 7 determined that $x_t = a^{n-1} \bmod n \neq 1$. If n is prime, however, Fermat's theorem (Theorem 31.31) says that $a^{n-1} = 1 \pmod n$ for all $a \in \mathbb{Z}_n^*$. Since $\mathbb{Z}_n^+ = \mathbb{Z}_n^*$ if n is prime, Fermat's theorem also says that $a^{n-1} = 1 \pmod n$ 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 $x_{i-1} \neq \pm 1 \pmod n$ yet $x_i = x_{i-1}^2 = 1 \pmod n$. Corollary 31.35 on page 934 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. ■

Thus, if 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.

Let's explore an alternative view of the behavior of WITNESS as a function of the sequence $X = \langle x_0, x_1, \ldots, x_t \rangle$. We'll find this view useful later on, when we analyze the error rate of the Miller-Rabin primality test. Note that if $x_i = 1$ for some $0 \leq i < t$, 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, so we can consider these positions in the sequence X as being all 1s. There are 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, since 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, since 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, since 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.

Now, let's examine the Miller-Rabin primality test based on how it uses the WITNESS procedure. As before, assume that n is an odd integer greater than 2.

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 $\mathbb{Z}_n^+$, except for 1 and $n-1$ (line 2). If it picks a value of a that 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 it found no witness because no witnesses exist, and therefore it assumes that n is prime. We'll see that this result is likely to be correct if s is large enough, but there is still a tiny chance that the procedure could be unlucky in its choice of s random values of a, so that even though the procedure failed to find a witness, at least one witness exists.

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, the column for $b = 35$ in Figure 31.4 (Section 31.6) shows that WITNESS computes $x_0 = a^{35} = 241 \pmod{561}$. Because of how the MODULAR-EXPONENTIATION procedure operates recursively on its parameter b, the first four columns in Figure 31.4 represent the factor 2^4 of 560—the rightmost four zeros in the binary representation of 560—reading these four zeros from right to left in the binary representation. Thus WITNESS computes the sequence $X = \langle 241, 298, 166, 67, 1 \rangle$. Then, in the last squaring step, WITNESS discovers that a^{280} is a nontrivial square root of 1 since $a^{280} = 67 \pmod{n}$ and $(a^{280})^2 = a^{560} = 1 \pmod{n}$. 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 (31.39), 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 31.39
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 $a^{n-1} = 1 \pmod{n}$ or, equivalently, $a \cdot a^{n-2} = 1 \pmod{n}$. Thus the equation $ax = 1 \pmod{n}$ has a solution, namely a^{n-2}. By Corollary 31.21 on page 924, $\gcd(a,n) \mid 1$, which in turn implies that $\gcd(a,n) = 1$. Therefore, a is a member of $\mathbb{Z}_n^*$, and 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 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 on page 921, we then have $|B| \le |\mathbb{Z}_n^*|/2$. Since $|\mathbb{Z}_n^*| \le n-1$, we obtain $|B| \le (n-1)/2$. Therefore, if all nonwitnesses are contained in a proper subgroup of $\mathbb{Z}_n^*$, then the number of nonwitnesses is at most $(n-1)/2$, so that the number of witnesses must be at least $(n-1)/2$.

To find a proper subgroup B of $\mathbb{Z}_n^*$ containing all of the nonwitnesses, we consider two cases.

Case 1: There exists an $x \in \mathbb{Z}_n^*$ such that

$$x^{n-1} \neq 1 \pmod{n} .$$

In other words, n is not a Carmichael number. Since, as noted earlier, Carmichael numbers are extremely rare, case 1 is the more typical case (e.g., when n has been chosen randomly and is being tested for primality).

Let $B = \{b \in \mathbb{Z}_n^* : b^{n-1} = 1 \pmod{n}\}$. The set B must be nonempty, since $1 \in B$. The set B is closed under multiplication modulo n, and so B is a subgroup of $\mathbb{Z}_n^*$ by Theorem 31.14. Every nonwitness belongs to B, since a nonwitness a satisfies $a^{n-1} = 1 \pmod{n}$. Since $x \in \mathbb{Z}_n^* - B$, we have that B is a proper subgroup of $\mathbb{Z}_n^*$.

Case 2: For all $x \in \mathbb{Z}_n^*$,

$$x^{n-1} = 1 \pmod{n} . \tag{31.40}$$

In other words, n is a Carmichael number. This case is extremely rare in practice. Unlike a pseudoprimality test, however, the Miller-Rabin test can efficiently determine that Carmichael numbers are composite, as we're about to see.

In this case, n cannot be a prime power. To see why, 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 on page 933 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 (31.21) on page 920.) By equation (31.40), we have $g^{n-1} = 1 \pmod{n}$. Then the discrete logarithm theorem (Theorem 31.33 on page 933, taking $y = 0$) implies that $n - 1 = 0 \pmod{\phi(n)}$, or

$$(p-1)p^{e-1} \mid p^e - 1 .$$

This statement 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 t and u are such 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

$$X = \langle a^u, a^{2u}, a^{2^2 u}, \ldots, a^{2^t u} \rangle$$

where 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

$$v^{2^j u} = -1 \pmod{n} .$$

Acceptable pairs certainly exist, since u is odd. Choose $v = n - 1$ and $j = 0$, and let $u = 2k + 1$, so that $v^{2^j u} = (n-1)^u = (n-1)^{2k+1}$. Taking this number modulo n gives $(n-1)^{2k+1} = (n-1)^{2k} \cdot (n-1) = (-1)^{2k} \cdot -1 = -1 \pmod{n}$. Thus, $(n-1, 0)$ 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

$$B = \{x \in \mathbb{Z}_n^* : x^{2^j u} = \pm 1 \pmod{n}\} .$$

Since B is closed under multiplication modulo n, it is a subgroup of $\mathbb{Z}_n^*$. By Theorem 31.15 on page 921, 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 $v^{2^j u} = -1 \pmod{n}$, we also have $v^{2^j u} = -1 \pmod{n_1}$ by Corollary 31.29 to the Chinese remainder theorem. By Corollary 31.28, there exists a w simultaneously satisfying the equations

$$w = v \pmod{n_1},$$
$$w = 1 \pmod{n_2}.$$

Therefore,

$$w^{2^j u} = -1 \pmod{n_1},$$
$$w^{2^j u} = 1 \pmod{n_2}.$$

Corollary 31.29 gives that $w^{2^j u} \not\equiv 1 \pmod{n_1}$ implies $w^{2^j u} \not\equiv 1 \pmod{n}$ and also that $w^{2^j u} \not\equiv -1 \pmod{n_2}$ 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 $w \in \mathbb{Z}_n^*$. We start by working separately modulo n_1 and modulo n_2. Working modulo n_1, since $v \in \mathbb{Z}_n^*$, we have that $\gcd(v, n) = 1$. Also, we have $\gcd(v, n_1) = 1$, since if v does not have any common divisors with n, then it certainly does not have any common divisors with n_1. Since $w = v \pmod{n_1}$, we see that $\gcd(w, n_1) = 1$. Working modulo n_2, we have $w = 1 \pmod{n_2}$ implies $\gcd(w, n_2) = 1$ by Exercise 31.2-3. Since $\gcd(w, n_1) = 1$ and $\gcd(w, n_2) = 1$, Theorem 31.6 on page 908 yields $\gcd(w, n_1 n_2) = \gcd(w, n) = 1$. That is, $w \in \mathbb{Z}_n^*$.

Therefore, we have $w \in \mathbb{Z}_n^* - B$, and we can conclude in case 2 that B, which includes all nonwitnesses, is a proper subgroup of $\mathbb{Z}_n^*$ and therefore has size at most $(n-1)/2$.

In either case, the number of witnesses to the compositeness of n is at least $(n-1)/2$. ■

Theorem 31.40

For any odd integer $n > 2$ and positive integer s, the probability that MILLER-RABIN(n, s) errs is at most 2^{-s}.

Proof By Theorem 31.39, if n is composite, then each execution of the **for** loop of lines 1–4 of MILLER-RABIN has a probability of at least $1/2$ of discovering a witness 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, so that $\beta \approx \lg n \approx 1.443 \ln n$. 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.20) on page 1189) and approximating $\Pr\left\{B \mid \overline{A}\right\}$ by 2^{-s}, we have

$$\begin{aligned} \Pr\{A \mid B\} &= \frac{\Pr\{A\}\Pr\{B \mid A\}}{\Pr\{A\}\Pr\{B \mid A\} + \Pr\left\{\overline{A}\right\}\Pr\left\{B \mid \overline{A}\right\}} \\ &\approx \frac{(1/\ln n) \cdot 1}{(1/\ln n) \cdot 1 + (1 - 1/\ln n) \cdot 2^{-s}} \\ &\approx \frac{1}{1 + 2^{-s}(\ln n - 1)} . \end{aligned}$$

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 you 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 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 considerably 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 31.39. 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 (Theorem 31.30) slightly to the form

$$a^{\lambda(n)} = 1 \pmod{n} \quad \text{for all } a \in \mathbb{Z}_n^* ,$$

where $n = p_1^{e_1} \cdots p_r^{e_r}$ and $\lambda(n)$ is defined by

$$\lambda(n) = \text{lcm}(\phi(p_1^{e_1}), \ldots, \phi(p_r^{e_r})) .$$

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 $561 = 3 \cdot 11 \cdot 17$, for which $\lambda(n) = \text{lcm}(2, 10, 16) = 80$, 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 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.

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 $\mu(ab) = (1 + \lg a)(1 + \lg b)$. 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 (3.31) on page 69 is exponential in n. (See, for example, the FIB procedure on page 751.)

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*** modulo p, 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

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \pmod{p} .$$

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

Knuth [260] contains a good discussion of algorithms for finding the greatest common divisor, as well as other basic number-theoretic algorithms. Dixon [121] gives an overview of factorization and primality testing. Bach [33], Riesel [378], and Bach and Shallit [34] provide overviews of the basics of computational number theory; Shoup [411] provides a more recent survey. The conference proceedings edited by Pomerance [362] contains several excellent survey articles.

Knuth [260] 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.E. Euclid's description may have been derived from an algorithm due to Eudoxus around 375 B.C.E. Euclid's algorithm may hold the honor of being the oldest nontrivial algorithm, rivaled only by an algorithm for multiplication known to the ancient Egyptians. Shallit [407] 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.E. and 200 C.E.—the date is quite uncertain. The same special case was given by the Greek mathematician Nichomachus around 100 C.E. It was generalized by Qin Jiushao 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 [327] and Rabin [373] and is the fastest randomized primality-testing algorithm known,

to within constant factors. The proof of Theorem 31.40 is a slight adaptation of one suggested by Bach [32]. A proof of a stronger result for MILLER-RABIN was given by Monier [332, 333]. 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 Saxena [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 [97], ran in $(\lg n)^{O(\lg\lg\lg n)}$ time on input n, which is just slightly superpolynomial. Nonetheless, for practical purposes, randomized primality-testing algorithms remain more efficient and are generally preferred.

Beauchemin, Brassard, Crépeau, Goutier, and Pomerance [40] nicely discuss the problem of finding large "random" primes.

The concept of a public-key cryptosystem is due to Diffie and Hellman [115]. The RSA cryptosystem was proposed in 1977 by Rivest, Shamir, and Adleman [380]. 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 [190] show that randomization can be an effective tool in the design of secure public-key encryption schemes. For signature schemes, Goldwasser, Micali, and Rivest [191] present a digital-signature scheme for which every conceivable type of forgery is provably as difficult as factoring. Katz and Lindell [253] provide an overview of modern cryptography.

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 [77] as an extension of the ideas in the number-field sieve factoring algorithm by Pollard [360] and Lenstra et al. [295] and refined by Coppersmith [102] 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 [296] may be more effective for some inputs than the number-field sieve method, since 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.

The string-matching problem can be stated formally as follows. The text is given as an array $T[1:n]$ of length n, and the pattern is an array $P[1:m]$ of length $m \leq n$. The elements of P and T are characters drawn from an alphabet Σ, which is a finite set of characters. For example, Σ could be the set $\{0, 1\}$, or it could be the set $\{a, b, \ldots, z\}$. The character arrays P and T are often called ***strings*** of characters.

As Figure 32.1 shows, 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 s is a ***valid shift***, and otherwise, s is 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.

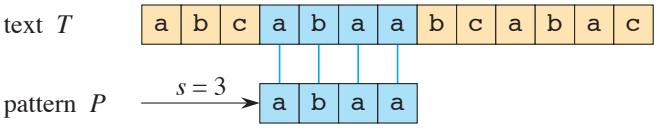


Figure 32.1 An example of the string-matching problem to find all occurrences of the pattern $P = \texttt{abaa}$ in the text $T = \texttt{abcabaabcabac}$. The pattern occurs only once in the text, at shift $s = 3$, which is a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded blue.

Except for the naive brute-force algorithm 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." Here are the preprocessing and matching times for each of the string-matching algorithms in this chapter. The total running time of each algorithm is the sum of the preprocessing and matching times:

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Suffix array[1]	$O(n \lg n)$	$O(m \lg n + km)$

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, which has the same $\Theta(n)$ matching time, but it reduces the preprocessing time to only $\Theta(m)$.

A completely different approach appears in Section 32.5, which examines suffix arrays and the longest common prefix array. You can use these arrays not only to find a pattern in a text, but also to answer other questions, such as what is the longest repeated substring in the text and what is the longest common substring between two texts. The algorithm to form the suffix array in Section 32.5 takes $O(n \lg n)$ time and, given the suffix array, the section shows how to compute the longest common prefix array in $O(n)$ time.

Notation and terminology

We denote by Σ^* (read "sigma-star") the set of all finite-length strings formed using characters from the alphabet Σ. This chapter considers only finite-length

[1] For suffix arrays, the preprocessing time of $O(n \lg n)$ comes from the algorithm presented in Section 32.5. It can be reduced to $\Theta(n)$ by using the algorithm in Problem 32-2. The factor k in the matching time denotes the number of occurrences of the pattern in the text.

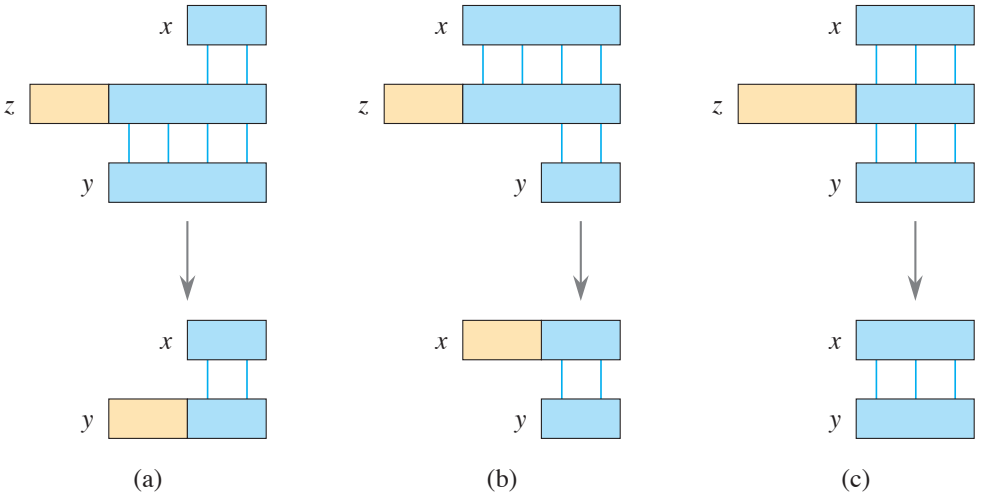


Figure 32.2 A graphical proof of Lemma 32.1. Suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown in blue) of the strings. **(a)** If $|x| \le |y|$, then $x \sqsupset y$. **(b)** If $|x| \ge |y|$, then $y \sqsupset x$. **(c)** If $|x| = |y|$, then $x = y$.

strings. The 0-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.

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 $|w| \le |x|$. Similarly, 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 $|w| \le |x|$. For example, $\texttt{ab} \sqsubset \texttt{abcca}$ and $\texttt{cca} \sqsupset \texttt{abcca}$. A string w is a ***proper prefix*** of x if $w \sqsubset x$ and $|w| < |x|$, and likewise for a ***proper suffix***. 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 $xa \sqsupset ya$. The $\sqsubset$ and $\sqsupset$ relations are transitive. 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 $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof See Figure 32.2 for a graphical proof. ■

For convenience, denote the k-character prefix $P[1:k]$ of the pattern $P[1:m]$ by $P[:k]$. Thus, we can write $P[:0] = \varepsilon$ and $P[:m] = P = P[1:m]$. Similarly, 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 $0 \le s \le n - m$ such that $P \sqsupset T[:s+m]$.

Our pseudocode allows 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 $\Theta(t)$ time, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$.

32.1 The naive string-matching algorithm

The NAIVE-STRING-MATCHER procedure 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, n, m)
1  for s = 0 to n − m
2      if P[1 : m] == T[s + 1 : s + m]
3          print "Pattern occurs with shift" s
```

Figure 32.3 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 1–3 considers each possible shift explicitly. The test in line 2 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 3 prints out each valid shift s.

Procedure NAIVE-STRING-MATCHER takes $O((n-m+1)m)$ time, 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 2 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.

NAIVE-STRING-MATCHER is far from an optimal procedure for this problem. Indeed, this chapter will show 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 $P = \mathtt{aaab}$

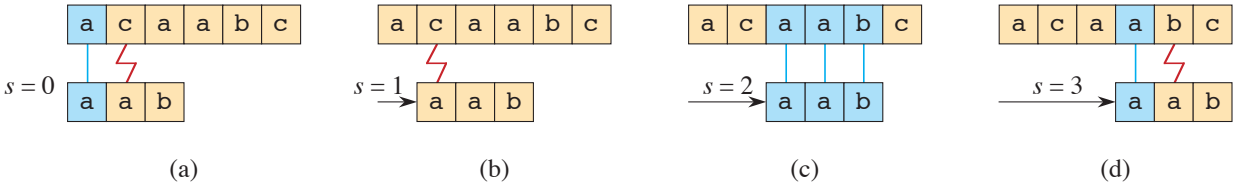


Figure 32.3 The operation of the NAIVE-STRING-MATCHER procedure for the pattern $P = \texttt{aab}$ and the text $T = \texttt{acaabc}$. Imagine the pattern P as a template that slides next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown in blue), and a red 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).

and $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $T[4] = \texttt{b}$. The following sections 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 $O(n)$ time 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 2 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 that the pattern P may contain occurrences of a ***gap character*** $\diamond$ that can match an *arbitrary* string of characters (even one of 0 length). For example, the pattern $\texttt{ab}\diamond\texttt{ba}\diamond\texttt{c}$ occurs in the text $\texttt{cabccbacbacab}$ as

$$\texttt{c}\underbrace{\texttt{ab}}\ \underbrace{\texttt{cc}}\ \underbrace{\texttt{ba}}\ \underbrace{\texttt{cba}}\ \underbrace{\texttt{c}}\ \texttt{ab}$$
$$\texttt{ab}\ \diamond\ \texttt{ba}\ \diamond\ \texttt{c}$$

and as

$$\texttt{c}\underbrace{\texttt{ab}}\ \underbrace{\texttt{ccbac}}\ \underbrace{\texttt{ba}}\ \underbrace{\ }\ \underbrace{\texttt{c}}\ \texttt{ab}\,.$$
$$\texttt{ab}\ \diamond\ \texttt{ba}\ \diamond\ \texttt{c}$$

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's assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit. (In the general case, you can assume that each character is a digit in radix-d notation, so that it has a numerical value in the range 0 to $d-1$, where $d = |\Sigma|$.) You can then view a string of k consecutive characters as representing a length-k decimal number. For example, the character string $\texttt{31415}$ corresponds to the decimal number 31,415. Because we interpret the input characters as both graphical symbols and digits, it will be convenient in this section to denote them as digits in 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 $T[s+1:s+m] = P[1:m]$, and thus, s is a valid shift if and only if $t_s = p$. If you could compute p in $\Theta(m)$ time and all the t_s values in a total of $\Theta(n-m+1)$ time,[2] then you could determine all valid shifts s in $\Theta(m) + \Theta(n-m+1) = \Theta(n)$

[2] 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.

time 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.)

Indeed, you can compute p in $\Theta(m)$ time using Horner's rule (see Problem 2-3):

$$p = P[m] + 10\Big(P[m-1] + 10\big(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots\big)\Big) .$$

Similarly, you can compute t_0 from $T[1:m]$ in $\Theta(m)$ time.

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in $\Theta(n-m)$ time, observe that you 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, suppose that $m = 5$, $t_s = 31415$, and the new low-order digit is $T[s+5+1] = 2$. The high-order digit to remove is $T[s+1] = 3$, and so

$$\begin{aligned} t_{s+1} &= 10\,(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \end{aligned}$$

If you precompute the constant 10^{m-1} (which you can do in $O(\lg m)$ time 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, you can compute p in $\Theta(m)$ time, and you can compute all of $t_0, t_1, \ldots, t_{n-m}$ in $\Theta(n-m+1)$ time. Therefore, you 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.

This scheme works well if P is short enough and the alphabet Σ is small enough that arithmetic operations on p and t_s take constant time. But what if P is long, or if the size of Σ means that instead of powers of 10 in equation (32.1) you have to use powers of a larger number (such as powers of 256 for the extended ASCII character set)? Then the values of p and t_s might be too large to work with in constant time. Fortunately, this problem can be solved, as Figure 32.4 shows: compute p and the t_s values modulo a suitable modulus q. You can compute p modulo q in $\Theta(m)$ time and all the t_s values modulo q in $\Theta(n-m+1)$ time. With $|\Sigma| = 10$, if you choose the modulus q as a prime such that $10q$ just fits within one computer word, then you can perform all the necessary computations with single-precision arithmetic. In general, with a d-ary alphabet $\{0, 1, \ldots, d-1\}$, 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

$$t_{s+1} = \big(d(t_s - T[s+1]h) + T[s+m+1]\big) \bmod q , \tag{32.2}$$

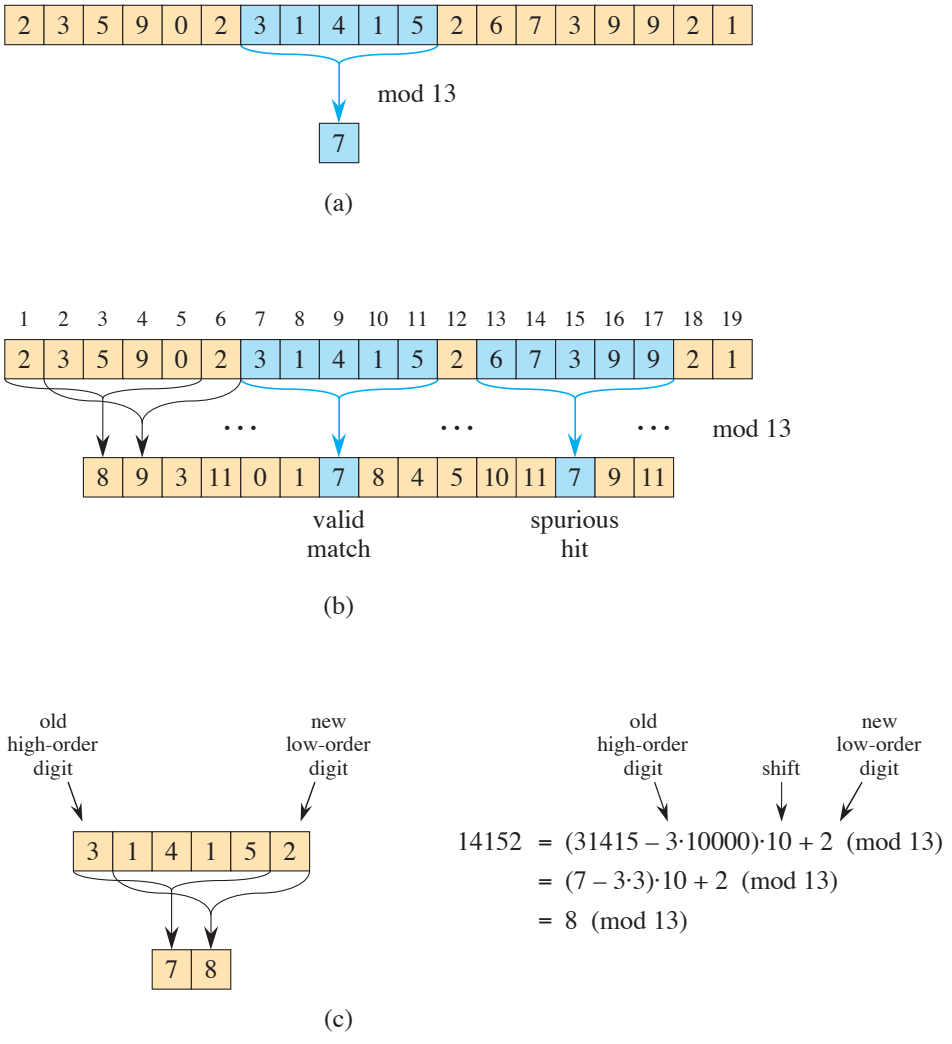


Figure 32.4 The Rabin-Karp algorithm. Each character is a decimal digit. Values are computed modulo 13. **(a)** A text string. A window of length 5 is shaded blue. The numerical value of the blue number, computed modulo 13, yields the value 7. **(b)** The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, look for windows whose value modulo 13 is 7, since $31415 = 7 \pmod{13}$. The algorithm finds two such windows, shaded blue in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern. The second window, 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 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.

where $h = d^{m-1} \bmod q$ 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: $t_s = p \pmod q$ does not automatically mean that $t_s = p$. On the other hand, if $t_s \neq p \pmod q$, then you definitely know that $t_s \neq p$, so that shift s is invalid. Thus you can use the test $t_s = p \pmod q$ as a fast heuristic test to rule out invalid shifts. If $t_s = p \pmod q$—a ***hit***—then you need to test further to see whether s is really valid or you just have a ***spurious hit***. This additional test explicitly checks the condition $P[1:m] = T[s+1:s+m]$. If q is large enough, then you would hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The procedure RABIN-KARP-MATCHER on the next page makes these ideas precise. The inputs to the procedure are the text T, the pattern P, their lengths n and m, the radix d to use (which is typically taken to be $|\Sigma|$), and the prime q to use. The procedure works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity: the procedure works correctly if all the subscripts are dropped. Line 1 initializes h to the value of the high-order digit position of an m-digit window. Lines 2–6 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 7–12 iterates through all possible shifts s, maintaining the following invariant:

Whenever line 8 is executed, $t_s = T[s+1:s+m] \bmod q$.

If a hit occurs because $p = t_s$ in line 8, then line 9 determines whether s is a valid shift or the hit was spurious via the test $P[1:m] == T[s+1:s+m]$. Line 10 prints out any valid shifts that are found. If $s < n - m$ (checked in line 11), then the **for** loop will iterate at least one more time, and so line 12 first executes to ensure that the loop invariant holds upon the next iteration. Line 12 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 $\Theta((n-m+1)m)$ time, since each of the $n-m+1$ possible shifts is valid.

In many applications, you 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 $\mathbb{Z}_q$. The expected number of spurious hits is then $O(n/q)$, because 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

```
RABIN-KARP-MATCHER(T, P, n, m, d, q)
 1  h = d^(m−1) mod q
 2  p = 0
 3  t_0 = 0
 4  for i = 1 to m                          // preprocessing
 5      p = (dp + P[i]) mod q
 6      t_0 = (dt_0 + T[i]) mod q
 7  for s = 0 to n − m                      // matching—try all possible shifts
 8      if p == t_s                         // a hit?
 9          if P[1 : m] == T[s + 1 : s + m] // valid shift?
10              print "Pattern occurs with shift" s
11      if s < n − m
12          t_{s+1} = (d(t_s − T[s + 1]h) + T[s + m + 1]) mod q
```

test of line 8 fails (actually, at most $n - m + 1$ positions) and checking each hit takes $O(m)$ time in line 9, 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 you choose $q \geq m$. That is, if the expected number of valid shifts is small ($O(1)$) and you choose the prime q to be larger than the length of the pattern, then you 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

Describe how to 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 $\{0, 1, \ldots, q-1\}$. Letting

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q \quad \text{and} \quad B(x) = \left(\sum_{i=0}^{n-1} b_i x^i \right) \bmod q \ ,$$

Alice evaluates $A(x)$ and Bob 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 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'll see how to construct the string-matching automaton for a given input pattern.

Finite automata

A ***finite automaton*** M, illustrated in Figure 32.5, 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.

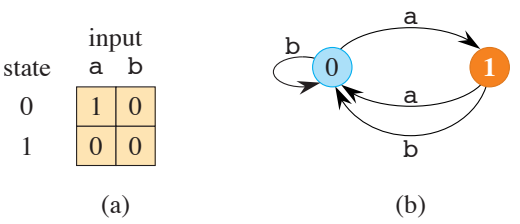


Figure 32.5 A simple two-state finite automaton with state set $Q = \{0, 1\}$, start state $q_0 = 0$, and input alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$. **(a)** A tabular representation of the transition function δ. **(b)** An equivalent state-transition diagram. State 1, in orange, is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that $\delta(1, \texttt{b}) = 0$. 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 $z = \texttt{a}^k$, 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 reading 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:

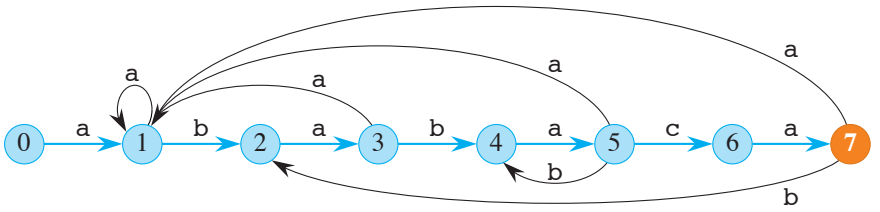
$$\begin{aligned} \phi(\varepsilon) &= q_0 \,, \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma \,. \end{aligned}$$

String-matching automata

For a given pattern P, a preprocessing step constructs a string-matching automaton specific to P. The automaton then searches the text string for occurrences of P. Figure 32.6 illustrates the automaton for the pattern $P = \texttt{ababaca}$. From now on, let's assume that P is fixed, and for brevity, we won't bother to 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 the pattern 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 32.6 **(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 (in orange) is the only accepting state. The transition function δ is defined by equation (32.4), and 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 in blue, correspond to successful matches between pattern and input characters. Except for the edges from state 7 to states 1 and 2, the left-going edges correspond to mismatches. Some edges corresponding to mismatches 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 in blue. **(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 substring of the pattern that occurs in the text is highlighted in blue. The automaton finds this 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 $\sigma(x) \le \sigma(y)$ (see Exercise 32.3-4).

We are now ready to define the string-matching automaton that corresponds to a given pattern $P[1:m]$:

- 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, for any state q and character a, by

$$\delta(q, a) = \sigma(P[:q]a) \; . \tag{32.4}$$

As the automaton consumes characters of the text T, it is trying to build a match of the pattern P against the most recently seen characters of T. At any time, the state number q gives the length of the longest prefix of P that matches the most recently seen text characters. Whenever the automaton reaches state m, the m most recently seen text characters match the first m characters of P. Since P has length m, reaching state m means that the m most recently seen text characters match the entire pattern, so that the automaton has found a match.

With this intuition behind the design of the automaton, here is the reasoning behind defining $\delta(q, a) = \sigma(P[:q]a)$. Suppose that the automaton is in state q after reading the first i characters of the text, that is, $q = \phi(T[:i])$. The intuitive idea then says that q also equals the length of the longest prefix of P that matches a suffix of $T[:i]$ or, equivalently, that $q = \sigma(T[:i])$. Thus, since $\phi(T[:i])$ and $\sigma(T[:i])$ both equal q, we will see (in Theorem 32.4 on page 973) 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 the transition should lead to the state corresponding to the longest prefix of P that is a suffix of $T[:i]a$. That state is $\sigma(T[:i]a)$, and equation (32.5) gives $\phi(T[:i]a) = \sigma(T[:i]a)$. 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 $T[:i]a$ has length not only $\sigma(T[:i]a)$, but also $\sigma(P[:q]a)$, and so $\phi(T[:i]a) = \sigma(P[:q]a)$. (Lemma 32.3 on page 972 will prove that $\sigma(T[:i]a) = \sigma(P[:q]a)$.) Thus, when the automaton is in state q, the transition function δ on character a should take the automaton to state $\delta(q, a) = \delta(\phi(T[:i]), a) = \phi(T[:i]a) = \sigma(P[:q]a)$ (with the last equality following from equation (32.5)).

There are two cases to consider, depending on whether the next character continues to match the pattern. 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 blue edges in Figure 32.6(a)). In the second case, $a \neq P[q+1]$, so that a does not extend the match being built. In this case, we need to find the longest prefix of P that is also a suffix of $T[:i]a$, which will have length at most q. The preprocessing step matches the pattern against itself when creating the string-matching automaton, so that the transition function can quickly identify the longest such smaller prefix of P.

Let's look at an example. Consider state 5 in the string-matching automaton of Figure 32.6. In state 5, the five most recently read characters of T are `ababa`, the characters along the spine of the automaton that reach state 5. If the next character of T is `c`, then the most recently read characters of T are `ababac`, which is the prefix of P with length 6. The automaton should continue along the spine to state 6. This is the first case, in which the match continues, and $\delta(5, \texttt{c}) = 6$. To illustrate the second case, suppose that in state 5, the next character of T is `b`, so the most recently read characters of T are `ababab`. Here, the longest prefix of P that matches the most recently read characters of T—that is, a suffix of the portion of T read so far—is `abab`, with length 4, so $\delta(5, \texttt{b}) = 4$.

To clarify the operation of a string-matching automaton, the simple and efficient procedure FINITE-AUTOMATON-MATCHER simulates 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. From the simple loop structure of FINITE-AUTOMATON-MATCHER, you can see that its matching time on a text string of length n is $\Theta(n)$, assuming that each lookup of the transition function δ takes constant time. This matching time, however, does not include the preprocessing time required to compute the transition function. We address this problem later, after first proving that the procedure FINITE-AUTOMATON-MATCHER operates correctly.

```
FINITE-AUTOMATON-MATCHER(T, δ, n, m)
1  q = 0
2  for i = 1 to n
3       q = δ(q, T[i])
4       if q == m
5            print "Pattern occurs with shift" i − m
```

Let's examine how the automaton operates on an input text $T[1:n]$. We will prove that the automaton is in state $\sigma(T[:i])$ after reading 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 read the pattern P. We start with two lemmas about the suffix function σ.

Lemma 32.2 (Suffix-function inequality)
For any string x and character a, we have $\sigma(xa) \le \sigma(x) + 1$.

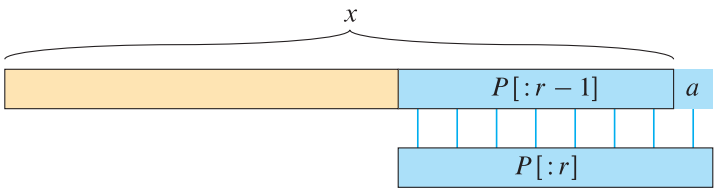


Figure 32.7 An illustration for the proof of Lemma 32.2. The figure shows that $r \le \sigma(x) + 1$, where $r = \sigma(xa)$.

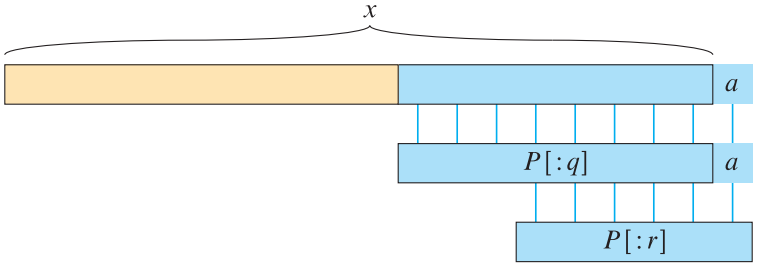


Figure 32.8 An illustration for the proof of Lemma 32.3. The figure shows that $r = \sigma(P[:q]a)$, where $q = \sigma(x)$ and $r = \sigma(xa)$.

Proof Referring to Figure 32.7, let $r = \sigma(xa)$. If $r = 0$, then the conclusion $\sigma(xa) = r \le \sigma(x)+1$ is trivially satisfied since $\sigma(x)$ is nonnegative. Now assume that $r > 0$. Then, $P[:r] \sqsupset xa$, by the definition of σ. Thus, $P[:r-1] \sqsupset x$, by dropping the a from both the end of $P[:r]$ and 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 The definition of σ gives that $P[:q] \sqsupset x$. As Figure 32.8 shows, we also have $P[:q]a \sqsupset xa$. 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 on page 959 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 the 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 assume that $\phi(T[:i]) = \sigma(T[:i])$. We will prove that $\phi(T[:i+1]) = \sigma(T[:i+1])$. Let q denote $\phi(T[:i])$, so that $q = \sigma(T[:i])$, and let a denote $T[i+1]$. Then,

$$\begin{aligned}
\phi(T[:i+1]) &= \phi(T[:i]a) && \text{(by the definitions of } T[:i+1] \text{ and } a) \\
&= \delta(\phi(T[:i]), a) && \text{(by the definition of } \phi) \\
&= \delta(q, a) && \text{(by the definition of } q) \\
&= \sigma(P[:q]a) && \text{(by the definition (32.4) of } \delta) \\
&= \sigma(T[:i]a) && \text{(by Lemma 32.3)} \\
&= \sigma(T[:i+1]) && \text{(by the definition of } T[:i+1]) \,.
\end{aligned}$$

■

By Theorem 32.4, if the machine enters state q on line 3, then q is the largest value such that $P[:q] \sqsupset T[:i]$. Thus, in line 4, $q = m$ if and only if the machine has just read an occurrence of the pattern P. Therefore, FINITE-AUTOMATON-MATCHER operates correctly.

Computing the transition function

The procedure COMPUTE-TRANSITION-FUNCTION on the following page computes the transition function δ from a given pattern $P[1:m]$. It computes $\delta(q, a)$ in a straightforward manner according to its definition in equation (32.4). The nested loops beginning on lines 1 and 2 consider all states q and all characters a, and lines 3–6 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 $q+1$, unless $q = m$, in which case k cannot be larger than m. It then decreases k until $P[:k]$ is a suffix of $P[:q]a$, which must eventually occur, since $P[:0] = \varepsilon$ is a suffix of every string.

```
COMPUTE-TRANSITION-FUNCTION(P, Σ, m)
1  for q = 0 to m
2      for each character a ∈ Σ
3          k = min{m, q + 1}
4          while P[:k] is not a suffix of P[:q]a
5              k = k − 1
6          δ(q, a) = k
7  return δ
```

The running time of COMPUTE-TRANSITION-FUNCTION is $O(m^3 |\Sigma|)$, because the outer loops contribute a factor of $m|\Sigma|$, the inner **while** loop can run at most $m + 1$ times, and the test for whether $P[:k]$ is a suffix of $P[:q]a$ on line 4 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), the time required to compute δ from P improves to $O(m|\Sigma|)$. This improved procedure for computing δ provides a way to 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

Draw a state-transition diagram for the string-matching automaton for the pattern $P = \texttt{aabab}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$ and illustrate its operation on the text string $T = \texttt{aaababaabaababaab}$.

32.3-2

Draw a state-transition diagram for the string-matching automaton for the pattern $P = \texttt{ababbabbababbababbabb}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$.

32.3-3

A pattern P is ***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

Let x and y be prefixes of the pattern P. Prove that $x \sqsupset y$ implies $\sigma(x) \le \sigma(y)$.

★ ***32.3-5***
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.

32.3-6
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

Knuth, Morris, and Pratt developed a linear-time string matching algorithm that avoids computing the transition function δ altogether. Instead, the KMP algorithm uses an auxiliary function π, which it precomputes from the pattern in $\Theta(m)$ time and stores in an array $\pi[1:m]$. The array π allows the algorithm 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 needed 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, the KMP algorithm saves a factor of $|\Sigma|$ in the preprocessing time by computing π rather than δ. Like the procedure FINITE-AUTOMATON-MATCHER, once preprocessing has completed, the KMP algorithm uses $\Theta(n)$ matching time.

The prefix function for a pattern

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. The KMP algorithm takes 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 32.9(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. Because these q text characters match, certain shifts must be 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 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 necessarily match.

More generally, suppose that you know that $P[:q] \sqsupset T[:s+q]$ or, equivalently, that $P[1:q] = T[s+1:s+q]$. You want to shift P so that some shorter prefix $P[:k]$ of P matches a suffix of $T[:s+q]$, if possible. You might have more than one choice for how much to shift, however. In Figure 32.9(b), shifting P by 2 positions works, so that $P[:3] \sqsupset T[:s+q]$, but so does shifting P by 4 positions, so that $P[:1] \sqsupset T[:s+q]$ in Figure 32.9(c). If more than one shift amount works, you should choose the smallest shift amount so that you do not miss any potential matches. Put more precisely, you want to answer this question:

> Given that pattern characters $P[1:q]$ match text characters $T[s+1:s+q]$ (that is, $P[:q] \sqsupset T[:s+q]$), what is the least shift $s' > s$ such that for some $k < q$,
>
> $$P[1:k] = T[s'+1:s'+k]\,, \tag{32.6}$$
>
> (that is, $P[:k] \sqsupset T[:s'+k]$), where $s'+k = s+q$?

Here's another way to look at this question. If you know $P[:q] \sqsupset T[:s+q]$, then how do you find the longest proper prefix $P[:k]$ of $P[:q]$ that is also a suffix of $T[:s+q]$? These questions are equivalent because given s and q, requiring $s'+k = s+q$ means that finding the smallest shift s' (2 in Figure 32.9(b)) is tantamount to finding the longest prefix length k (3 in Figure 32.9(b)). If you add the difference $q-k$ in the lengths of these prefixes of P to the shift s, you get the new shift s', so that $s' = s + (q-k)$. In the best case, $k = 0$, so that $s' = s+q$, immediately ruling out shifts $s+1, s+2, \ldots, s+q-1$. In any case, at the new shift s', it is redundant to compare the first k characters of P with the corresponding characters of T, since equation (32.6) guarantees that they match.

As Figure 32.9(d) demonstrates, you can precompute the necessary information by comparing the pattern against itself. Since $T[s'+1:s'+k]$ is part of the matched portion of the text, it is a suffix of the string $P[:q]$. Therefore, think of 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. It will be convenient to store, for each value of q, the number k of matching characters at the new shift s', rather than storing, say, the amount $s'-s$ to shift by.

Let's look at the precomputed information a little more formally. For a given pattern $P[1:m]$, the ***prefix function*** for 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 $P[:q]$. Here is the complete prefix function π for the pattern `ababaca`:

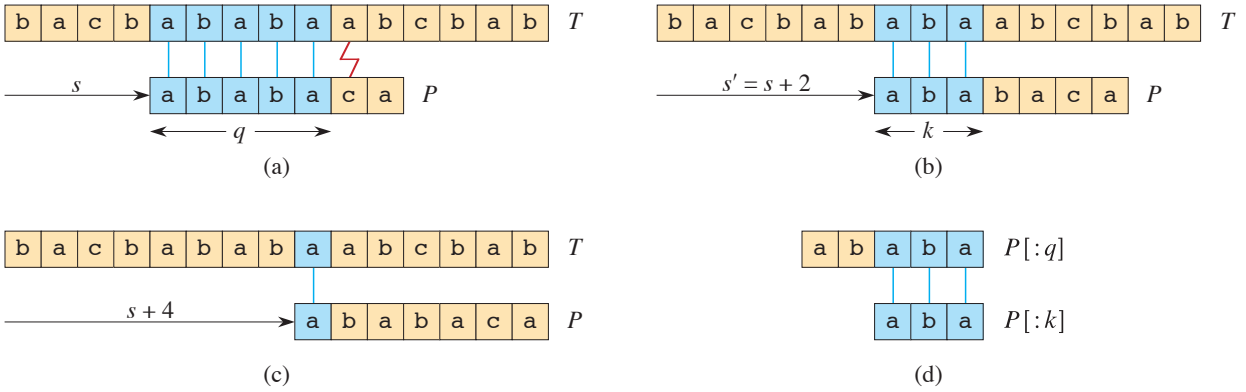


Figure 32.9 The prefix function π. **(a)** The pattern $P = \texttt{ababaca}$ aligns with a text T so that the first $q = 5$ characters match. Matching characters, in blue, are connected by blue lines. **(b)** Knowing these particular 5 matched characters ($P[:5]$) suffices to deduce that a shift of $s + 1$ is invalid, but that a shift of $s' = s + 2$ is consistent with everything known about the text and therefore is potentially valid. The prefix $P[:k]$, where $k = 3$, aligns with the text seen so far. **(c)** A shift of $s + 4$ is also potentially valid, but it leaves only the prefix $P[:1]$ aligned with the text seen so far. **(d)** To precompute useful information for such deductions, compare the pattern with itself. Here, the longest prefix of P that is also a proper suffix of $P[:5]$ is $P[:3]$. The array π represents this precomputed information, 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).

i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

The procedure KMP-MATCHER on the following page gives the Knuth-Morris-Pratt matching algorithm. The procedure follows from FINITE-AUTOMATON-MATCHER for the most part. To compute π, KMP-MATCHER calls the auxiliary procedure COMPUTE-PREFIX-FUNCTION. 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.

Next, let's analyze the running times of these procedures. Then we'll prove them correct, which 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 16.1). The only tricky part is showing that the **while** loop of lines 5–6 executes $O(m)$ times alto-

```
KMP-MATCHER(T, P, n, m)
 1  π = COMPUTE-PREFIX-FUNCTION(P, m)
 2  q = 0                              // number of characters matched
 3  for i = 1 to n                     // scan the text from left to right
 4      while q > 0 and P[q + 1] ≠ T[i]
 5          q = π[q]                   // next character does not match
 6      if P[q + 1] == T[i]
 7          q = q + 1                  // next character matches
 8      if q == m                      // is all of P matched?
 9          print "Pattern occurs with shift" i − m
10          q = π[q]                   // look for the next match

COMPUTE-PREFIX-FUNCTION(P, m)
 1  let π[1 : m] be a new array
 2  π[1] = 0
 3  k = 0
 4  for q = 2 to m
 5      while k > 0 and P[k + 1] ≠ P[q]
 6          k = π[k]
 7      if P[k + 1] == P[q]
 8          k = k + 1
 9      π[q] = k
10  return π
```

gether. Starting with some observations about k, we'll show that it makes at most $m-1$ iterations. First, line 3 starts k at 0, and the only way that k increases is by the increment operation in line 8, which executes at most once per iteration of the **for** loop of lines 4–9. 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 2 and 9 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 $\Theta(m)$ time.

Exercise 32.4-4 asks you to show, by a similar aggregate analysis, that the matching time of KMP-MATCHER is $\Theta(n)$.

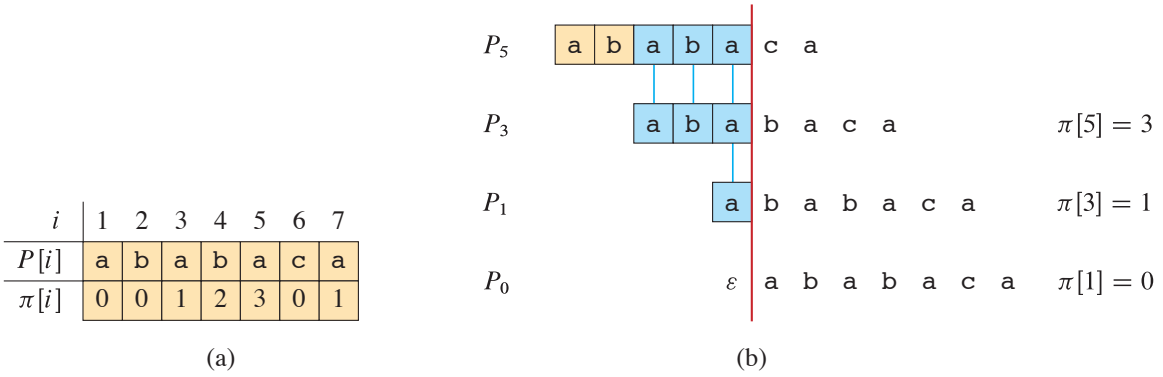


Figure 32.10 An illustration of Lemma 32.5 for the pattern $P = \texttt{ababaca}$ and $q = 5$. **(a)** The π function for the given pattern. Since $\pi[5] = 3$, $\pi[3] = 1$, and $\pi[1] = 0$, iterating π gives $\pi^*[5] = \{3, 1, 0\}$. **(b)** Sliding the template containing the pattern P to the right and noting when some prefix $P[:k]$ of P matches up with some proper suffix of $P[:5]$. Matches occur when $k = 3$, 1, and 0. In the figure, the first row gives P, and the vertical red 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 in blue. Blue 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.

Compared with FINITE-AUTOMATON-MATCHER, by using π rather than δ, the KMP algorithm reduces 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'll see a little later that the prefix function π helps to 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. Doing so requires finding all prefixes $P[:k]$ that are proper suffixes of a given prefix $P[:q]$. The value of $\pi[q]$ gives us the length of the longest such prefix, but the following lemma, illustrated in Figure 32.10, shows that iterating the prefix function π generates all the prefixes $P[:k]$ that are proper suffixes of $P[:q]$. Let

$$\pi^*[q] = \left\{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\right\} ,$$

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 $i \geq 1$ (so that $\pi[q] = \pi^{(1)}[q]$), and where the sequence in $\pi^*[q]$ stops upon reaching $\pi^{(t)}[q] = 0$ for some $t \geq 1$.

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. Now consider some $u \geq 1$ such that both $\pi^{(u)}[q]$ and $\pi^{(u+1)}[q]$ belong to $\pi^*[q]$. Let $i = \pi^{(u)}[q]$, so that $\pi[i] = \pi^{(u+1)}[q]$. The inductive hypothesis is that $P[:i] \sqsupset P[:q]$. Because the relations $<$ and $\sqsupset$ are transitive, we have $\pi[i] < i < q$ and $P[:\pi[i]] \sqsupset P[:i] \sqsupset P[:q]$, which establishes equation (32.7) 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 $\pi[q] \in \pi^*[q]$, it must be the case that $j < \pi[q]$. Having established that $\pi^*[q]$ contains at least one integer greater than j, let j' denote the smallest such integer. (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 2 of COMPUTE-PREFIX-FUNCTION is certainly correct, since $\pi[q] < q$ for all q. We'll 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 $P[:r] \sqsupset P[:q]$, and 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

$$\begin{aligned} E_{q-1} &= \{k \in \pi^*[q-1] : P[k+1] = P[q]\} \\ &= \{k : k < q-1 \text{ and } P[:k] \sqsupset P[:q-1] \text{ and } P[k+1] = P[q]\} \\ &\qquad \text{(by Lemma 32.5)} \\ &= \{k : k < q-1 \text{ and } P[:k+1] \sqsupset P[:q]\} \ . \end{aligned}$$

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 extending $P[:k]$ to $P[:k+1]$ produces 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. Then, for $q = 2, 3, \ldots, m$,

$$\pi[q] = \begin{cases} 0 & \text{if } E_{q-1} = \emptyset \ , \\ 1 + \max E_{q-1} & \text{if } E_{q-1} \neq \emptyset \ . \end{cases}$$

Proof If E_{q-1} is empty, there is no $k \in \pi^*[q-1]$ (including $k = 0$) such that extending $P[:k]$ to $P[:k+1]$ produces a proper suffix of $P[:q]$. Therefore, $\pi[q] = 0$.

If, instead, 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, the definition of $\pi[q]$ gives

$$\pi[q] \geq 1 + \max E_{q-1} \ . \tag{32.8}$$

Note that $\pi[q] > 0$. Let $r = \pi[q] - 1$, so that $r + 1 = \pi[q] > 0$, and therefore $P[:r+1] \sqsupset P[:q]$. If a nonempty string is a suffix of another, then the two strings must have the same last character. Since $r + 1 > 0$, the prefix $P[:r+1]$ is nonempty, and so $P[r+1] = P[q]$. Furthermore, $r \in \pi^*[q-1]$ by Lemma 32.6. Therefore, $r \in E_{q-1}$, and so $\pi[q] - 1 = r \leq \max E_{q-1}$ or, equivalently,

$$\pi[q] \leq 1 + \max E_{q-1} \ . \tag{32.9}$$

Combining equations (32.8) and (32.9) completes the proof. ■

We now finish the proof that COMPUTE-PREFIX-FUNCTION computes π correctly. The key is to combine the definition of E_{q-1} with the statement of Corollary 32.7, so that $\pi[q]$ equals 1 plus the greatest value of k in $\pi^*[q-1]$ such that

$P[k+1] = P[q]$. First, in COMPUTE-PREFIX-FUNCTION, $k = \pi[q-1]$ at the start of each iteration of the **for** loop of lines 4–9. This condition is enforced by lines 2 and 3 when the loop is first entered, and it remains true in each successive iteration because of line 9. Lines 5–8 adjust k so that it becomes the correct value of $\pi[q]$. The **while** loop of lines 5–6 searches through all values $k \in \pi^*[q-1]$ in decreasing order to find the value of $\pi[q]$. The loop terminates either because k reaches 0 or $P[k+1] = P[q]$. Because the "and" operator short-circuits, if the loop terminates because $P[k+1] = P[q]$, then k must have also been positive, and so k is the greatest value in E_{q-1}. In this case, lines 7–9 set $\pi[q]$ to $k+1$, according to Corollary 32.7. If, instead, the **while** loop terminates because $k = 0$, then there are two possibilities. If $P[1] = P[q]$, then $E_{q-1} = \{0\}$, and lines 7–9 set both k and $\pi[q]$ to 1. If $k = 0$ and $P[1] \neq P[q]$, however, then $E_{q-1} = \emptyset$. In this case, line 9 sets $\pi[q]$ to 0, again according to Corollary 32.7, which completes the proof of the correctness of COMPUTE-PREFIX-FUNCTION.

Correctness of the Knuth-Morris-Pratt algorithm

You 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'll 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 upon testing for equality with m (at line 8 in KMP-MATCHER and at line 4 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'll see a little later why line 10 in KMP-MATCHER is necessary).

Before formally proving 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 the state number is incremented: $\delta(q, a) = q + 1$. Otherwise, $a \neq P[q+1]$, so that a does not continue to match the pattern, and the state number does not increase: $0 \leq \delta(q, a) \leq q$. 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 4 immediately comes up false, the test in line 6 comes up true, and line 7 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 4–5 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 7 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 a state numbered 1 more than some state in $\pi^*[q]$.

Let's look at the example in Figures 32.6 and 32.10, which are for the pattern $P = \texttt{ababaca}$. Suppose that the automaton is in state $q = 5$, having matched `ababa`. The states in $\pi^*[5]$ are, in descending order, 3, 1, and 0. If the next character scanned is `c`, then you can see that the automaton moves to state $\delta(5, \texttt{c}) = 6$ in both FINITE-AUTOMATON-MATCHER (line 3) and KMP-MATCHER (line 7). 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 after executing line 5 once, and the automaton arrives in state $q' = \pi[5] = 3$. Since $P[q'+1] = P[4] = \texttt{b}$, the test in line 6 comes up true, and the automaton 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 4 executes, the test comes up true. The first time finds that $P[6] = \texttt{c} \neq \texttt{a}$, and the automaton moves to state $\pi[5] = 3$ (the first state in $\pi^*[5]$). The second time finds that $P[4] = \texttt{b} \neq \texttt{a}$, and the automaton moves to state $\pi[3] = 1$ (the second state in $\pi^*[5]$). The third time finds that $P[2] = \texttt{b} \neq \texttt{a}$, and the automaton moves to state $\pi[1] = 0$ (the last state in $\pi^*[5]$). The **while** loop exits once it arrives in state $q' = 0$. Now line 6 finds that $P[q'+1] = P[1] = \texttt{a}$, and line 7 moves the automaton to the new state $q'+1 = 1 = \delta(5, \texttt{a})$.

Thus, the 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 line 3 of FINITE-AUTOMATON-MATCHER executes. 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. By the inductive hypothesis, the state number q equals $\sigma(T[:i-1])$ at the start of the loop iteration. We need to show that when line 8 is reached, the new value of q is $\sigma(T[:i])$. (Again, we'll handle line 10 separately.)

Considering q to be the state number at the start of the **for** loop iteration, when KMP-MATCHER considers 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 $0 < \sigma(T[:i]) \le q$.

- 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 4–5 iterates through each value q' in $\pi^*[q]$, but although $P[:q'] \sqsupset P[:q] \sqsupset T[:i-1]$ for every $q' \in \pi^*[q]$ (because $<$ are $\sqsupset$ are transitive relations), the loop never finds a q' such that $P[q'+1] = T[i]$. The loop terminates when q reaches 0, and of course line 7 does not execute. Therefore, $q = 0$ at line 8, so that now $q = \sigma(T[:i])$.
- If $\sigma(T[:i]) = q+1$, then $P[q+1] = T[i]$, and the **while** loop test in line 4 fails the first time through. Line 7 executes, incrementing the state number to $q+1$, which equals $\sigma(T[:i])$.
- If $0 < \sigma(T[:i]) \le q'$, then the **while** loop of lines 4–5 iterates at least once, checking in decreasing order each value 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 7 increments q, the new state number q equals $\sigma(T[:i])$.

Line 10 is necessary in KMP-MATCHER, because otherwise, line 4 might try to reference $P[m+1]$ after finding an occurrence of P. (The argument that $q = \sigma(T[:i-1])$ upon the next execution of line 4 remains valid by the hint given in Exercise 32.4-8: that $\delta(m,a) = \delta(\pi[m],a)$ 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 5 (but not line 10) 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, `braze` and `zebra` 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 $P[q+1] \neq a$.)

32.5 Suffix arrays

The algorithms we have seen thus far in this chapter can efficiently find all occurrences of a pattern in a text. That is, however, all they can do. This section presents a different approach—suffix arrays—with which you can find all occurrences of a pattern in a text, but also quite a bit more. A suffix array won't find all occurrences

i	1	2	3	4	5	6	7
$T[i]$	r	a	t	a	t	a	t

i	$SA[i]$	$rank[i]$	$LCP[i]$	suffix $T[SA[i]:]$
1	6	4	0	at
2	4	3	2	atat
3	2	7	4	atatat
4	1	2	0	ratatat
5	7	6	0	t
6	5	1	1	tat
7	3	5	3	tatat

Figure 32.11 The suffix array SA, rank array $rank$, longest common prefix array LCP, and lexicographically sorted suffixes of the text $T =$ `ratatat` with length $n = 7$. The value of $rank[i]$ indicates the position of the suffix $T[i:]$ in the lexicographically sorted order: $rank[SA[i]] = i$ for $i = 1, 2, \ldots, n$. The $rank$ array is used to compute the LCP array.

of a pattern as quickly as, say, the Knuth-Morris-Pratt algorithm, but its additional flexibility makes it well worth studying.

A suffix array is simply a compact way to represent the lexicographically sorted order of all n suffixes of a length-n text. Given a text $T[1:n]$, let $T[i:]$ denote the suffix $T[i:n]$. The ***suffix array*** $SA[1:n]$ of T is defined such that if $SA[i] = j$, then $T[j:]$ is the ith suffix of T in lexicographic order.[3] That is, the ith suffix of T in lexicographic order is $T[SA[i]:]$. Along with the suffix array, another useful array is the ***longest common prefix array*** $LCP[1:n]$. The entry $LCP[i]$ gives the length of the longest common prefix between the ith and $(i-1)$st suffixes in the sorted order (with $LCP[SA[1]]$ defined to be 0, since there is no prefix lexicographically smaller than $T[SA[1]:]$). Figure 32.11 shows the suffix array and longest common prefix array for the 7-character text `ratatat`.

Given the suffix array for a text, you can search for a pattern via binary search on the suffix array. Each occurrence of a pattern in the text starts some suffix of the text, and because the suffix array is in lexicographically sorted order, all occurrences of a pattern will appear at the start of consecutive entries of the suffix array. For example, in Figure 32.11, the three occurrences of `at` in `ratatat` appear in entries 1 through 3 of the suffix array. If you find the length-m pattern in the length-n suffix array via binary search (taking $O(m \lg n)$ time because each comparison takes $O(m)$ time), then you can find all occurrences of the pattern in the text by searching backward and forward from that spot until you find a suffix that does not start with the pattern (or you go beyond the bounds of the suffix array). If the pattern occurs k times, then the time to find all k occurrences is $O(m \lg n + km)$.

---

[3] Informally, lexicographic order is "alphabetical order" in the underlying character set. A more precise definition of lexicographic order appears in Problem 12-2 on page 327.

With the longest common prefix array, you can find a longest repeated substring, that is, the longest substring that occurs more than once in the text. If $LCP[i]$ contains a maximum value in the LCP array, then a longest repeated substring appears in $T[SA[i] : SA[i] + LCP[i] - 1]$. In the example of Figure 32.11, the LCP array has one maximum value: $LCP[3] = 4$. Therefore, since $SA[3] = 2$, the longest repeated substring is $T[2:5] = \texttt{atat}$. Exercise 32.5-3 asks you to use the suffix array and longest common prefix array to find the longest common substrings between two texts. Next, we'll see how to compute the suffix array for an n-character text in $O(n \lg n)$ time and, given the suffix array and the text, how to compute the longest common prefix array in $\Theta(n)$ time.

Computing the suffix array

There are several algorithms to compute the suffix array of a length-n text. Some run in linear time, but are rather complicated. One such algorithm is given in Problem 32-2. Here we'll explore a simpler algorithm that runs in $\Theta(n \lg n)$ time.

The idea behind the $O(n \lg n)$-time procedure COMPUTE-SUFFIX-ARRAY on the following page is to lexicographically sort substrings of the text with increasing lengths. The procedure makes several passes over the text, with the substring length doubling each time. By the $\lceil \lg n \rceil$th pass, the procedure is sorting all the suffixes, thereby gaining the information needed to construct the suffix array. The key to attaining an $O(n \lg n)$-time algorithm will be to have each pass after the first sort in linear time, which will indeed be possible by using radix sort.

Let's start with a simple observation. Consider any two strings, s_1 and s_2. Decompose s_1 into s'_1 and s''_1, so that s_1 is s'_1 concatenated with s''_1. Likewise, let s_2 be s'_2 concatenated with s''_2. Now, suppose that s'_1 is lexicographically smaller than s'_2. Then, regardless of s''_1 and s''_2, it must be the case that s_1 is lexicographically smaller than s_2. For example, let $s_1 = \texttt{aaz}$ and $s_2 = \texttt{aba}$, and decompose s_1 into $s'_1 = \texttt{aa}$ and $s''_1 = \texttt{z}$ and s_2 into $s'_2 = \texttt{ab}$ and $s''_2 = \texttt{a}$. Because s'_1 is lexicographically smaller than s'_2, it follows that s_1 is lexicographically smaller than s_2, even though s''_2 is lexicographically smaller than s''_1.

Instead of comparing substrings directly, COMPUTE-SUFFIX-ARRAY represents substrings of the text with integer ***ranks***. Ranks have the simple property that one substring is lexicographically smaller then another if and only if it has a smaller rank. Identical substrings have equal ranks.

Where do these ranks come from? Initially, the substrings being considered are just single characters from the text. Assume that, as in many programming languages, there is a function, ord, that maps a character to its underlying encoding, which is a positive integer. The ord function could be the ASCII or Unicode encodings or any other function that produces a relative ordering of the characters. For example if all the characters are known to be lowercase letters, then $\text{ord}(\texttt{a}) = 1$,

```
COMPUTE-SUFFIX-ARRAY(T, n)
 1  allocate arrays substr-rank[1:n], rank[1:n], and SA[1:n]
 2  for i = 1 to n
 3      substr-rank[i].left-rank = ord(T[i])
 4      if i < n
 5          substr-rank[i].right-rank = ord(T[i + 1])
 6      else substr-rank[i].right-rank = 0
 7      substr-rank[i].index = i
 8  sort the array substr-rank into monotonically increasing order based
        on the left-rank attributes, using the right-rank attributes to break ties;
        if still a tie, the order does not matter
 9  l = 2
10  while l < n
11      MAKE-RANKS(substr-rank, rank, n)
12      for i = 1 to n
13          substr-rank[i].left-rank = rank[i]
14          if i + l ≤ n
15              substr-rank[i].right-rank = rank[i + l]
16          else substr-rank[i].right-rank = 0
17          substr-rank[i].index = i
18      sort the array substr-rank into monotonically increasing order based
            on the left-rank attributes, using the right-rank attributes
            to break ties; if still a tie, the order does not matter
19      l = 2l
20  for i = 1 to n
21      SA[i] = substr-rank[i].index
22  return SA

MAKE-RANKS(substr-rank, rank, n)
 1  r = 1
 2  rank[substr-rank[1].index] = r
 3  for i = 2 to n
 4      if substr-rank[i].left-rank ≠ substr-rank[i − 1].left-rank
               or substr-rank[i].right-rank ≠ substr-rank[i − 1].right-rank
 5          r = r + 1
 6      rank[substr-rank[i].index] = r
```

	After lines 2–7				After line 8				
i	*left-rank*	*right-rank*	*index*	substring	i	*left-rank*	*right-rank*	*index*	substring
1	114	97	1	`ra`	1	97	116	2	`at`
2	97	116	2	`at`	2	97	116	4	`at`
3	116	97	3	`ta`	3	97	116	6	`at`
4	97	116	4	`at`	4	114	97	1	`ra`
5	116	97	5	`ta`	5	116	0	7	`t`
6	97	116	6	`at`	6	116	97	3	`ta`
7	116	0	7	`t`	7	116	97	5	`ta`

Figure 32.12 The *substr-rank* array for indices $i = 1, 2, \ldots, 7$ after the **for** loop of lines 2–7 and after the sorting step in line 8 for input string $T =$ `ratatat`.

ord(`b`) = 2, ..., ord(`z`) = 26 would work. Once the substrings being considered contain multiple characters, their ranks will be positive integers less than or equal to n, coming from their relative order after being sorted. An empty substring always has rank 0, since it is lexicographically less than any nonempty substring.

The COMPUTE-SUFFIX-ARRAY procedure uses objects internally to keep track of the relative ordering of the substrings according to their ranks. When considering substrings of a given length, the procedure creates and sorts an array $substr\text{-}rank[1:n]$ of n objects, each with the following attributes:

- *left-rank* contains the rank of the left part of the substring.
- *right-rank* contains the rank of the right part of the substring.
- *index* contains the index into the text T of where the substring starts.

Before delving into the details of how the procedure works, let's look at how it operates on the input text `ratatat`, with $n = 7$. Assuming that the ord function returns the ASCII code for a character, Figure 32.12 shows the *substr-rank* array after the **for** loop of lines 2–7 and then after the sorting step in line 8. The *left-rank* and *right-rank* values after lines 2–7 are the ranks of length-1 substrings in positions i and $i + 1$, for $i = 1, 2, \ldots, n$. These initial ranks are the ASCII values of the characters. At this point, the *left-rank* and *right-rank* values give the ranks of the left and right part of each substring of length 2. Because the substring starting at index 7 consists of only one character, its right part is empty and so its *right-rank* is 0. After the sorting step in line 8, the *substr-rank* array gives the relative lexicographic order of all the substrings of length 2, with starting points of these substrings in the *index* attribute. For example, the lexicographically smallest length-2 substring is `at`, which starts at position $substr\text{-}rank[1].index$, which equals 2. This substring also occurs at positions $substr\text{-}rank[2].index = 4$ and $substr\text{-}rank[3].index = 6$.

The procedure then enters the **while** loop of lines 10–19. The loop variable l gives an upper bound on the length of substrings that have been sorted thus far.

After line 11	
i	*rank*
1	2
2	1
3	4
4	1
5	4
6	1
7	3

After lines 12–17				
i	*left-rank*	*right-rank*	*index*	substring
1	2	4	1	`rata`
2	1	1	2	`atat`
3	4	4	3	`tata`
4	1	1	4	`atat`
5	4	3	5	`tat`
6	1	0	6	`at`
7	3	0	7	`t`

After line 18				
i	*left-rank*	*right-rank*	*index*	substring
1	1	0	6	`at`
2	1	1	2	`atat`
3	1	1	4	`atat`
4	2	4	1	`rata`
5	3	0	7	`t`
6	4	3	5	`tat`
7	4	4	3	`tata`

Figure 32.13 The *rank* array after line 11 and the *substr-rank* array after lines 12–17 and after line 18 in the first iteration of the **while** loop of lines 10–19, where $l = 2$.

Entering the **while** loop, therefore, the substrings of length at most $l = 2$ are sorted. The call of MAKE-RANKS in line 11 gives each of these substrings its rank in the sorted order, from 1 up to the number of unique length-2 substrings, based on the values it finds in the *substr-rank* array. With $l = 2$, MAKE-RANKS sets $rank[i]$ to be the rank of the length-2 substring $T[i : i + 1]$. Figure 32.13 shows these new ranks, which are not necessarily unique. For example, since the length-2 substring `at` occurs at positions 2, 4, and 6, MAKE-RANKS finds that *substr-rank*[1], *substr-rank*[2], and *substr-rank*[3] have equal values in *left-rank* and in *right-rank*. Since *substr-rank*[1].*index* $= 2$, *substr-rank*[2].*index* $= 4$, and *substr-rank*[3].*index* $= 6$, and since `at` is the smallest substring in lexicographic order, MAKE-RANKS sets $rank[2] = rank[4] = rank[6] = 1$.

This iteration of the **while** loop will sort the substrings of length at most 4 based on the ranks from sorting the substrings of length at most 2. The **for** loop of lines 12–17 reconstitutes the *substr-rank* array, with *substr-rank*$[i]$.*left-rank* based on $rank[i]$ (the rank of the length-2 substring $T[i : i+1]$) and *substr-rank*$[i]$.*right-rank* based on $rank[i+2]$ (the rank of the length-2 substring $T[i+2 : i+3]$, which is 0 if this substring starts beyond the end of the length-n text). Together, these two ranks give the relative rank of the length-4 substring $T[i : i + 3]$. Figure 32.13 shows the effect of lines 12–17. The figure also shows the result of sorting the *substr-rank* array in line 18, based on the *left-rank* attribute, and using the *right-rank* attribute to break ties. Now *substr-rank* gives the lexicographically sorted order of all substrings with length at most 4.

The next iteration of the **while** loop, with $l = 4$, sorts the substrings of length at most 8 based on the ranks from sorting the substrings of length at most[4] 4. Figure 32.14 shows the ranks of the length-4 substrings and the *substr-rank* array

[4] Why keep saying "length at most"? Because for a given value of l, a substring of length l starting at position i is $T[i : i + l - 1]$. If $i + l - 1 > n$, then the substring cuts off at the end of the text.

After line 11		*After lines 12–17*					*After line 18*				
i	*rank*	*i*	*left-rank*	*right-rank*	*index*	substring	*i*	*left-rank*	*right-rank*	*index*	substring
1	3	1	3	5	1	`ratatat`	1	1	0	6	`at`
2	2	2	2	1	2	`atatat`	2	2	0	4	`atat`
3	6	3	6	4	3	`tatat`	3	2	1	2	`atatat`
4	2	4	2	0	4	`atat`	4	3	5	1	`ratatat`
5	5	5	5	0	5	`tat`	5	4	0	7	`t`
6	1	6	1	0	6	`at`	6	5	0	5	`tat`
7	4	7	4	0	7	`t`	7	6	4	3	`tatat`

Figure 32.14 The *rank* array after line 11 and the *substr-rank* array after lines 12–17 and after line 18 in the second—and final—iteration of the **while** loop of lines 10–19, where $l = 4$.

before and after sorting. This iteration is the final one, since with the length n of the text equaling 7, the procedure has sorted all substrings.

In general, as the loop variable l increases, more and more of the right parts of the substrings are empty. Therefore, more of the *right-rank* values are 0. Because i is at most n within the loop of lines 12–17, the left part of each substring is always nonempty, and so all *left-rank* values are always positive.

This example illuminates why the COMPUTE-SUFFIX-ARRAY procedure works. The initial ranks established in lines 2–7 are simply the ord values of the characters in the text, and so when line 8 sorts the *substr-rank* array, its ordering corresponds to the lexicographic ordering of the length-2 substrings. Each iteration of the **while** loop of lines 10–19 takes sorted substrings of length l and produces sorted substrings of length $2l$. Once l reaches or exceeds n, all substrings have been sorted.

Within an iteration of the **while** loop, the MAKE-RANKS procedure "re-ranks" the substrings that were sorted, either by line 8 before the first iteration or by line 18 in the previous iteration. MAKE-RANKS takes a *substr-rank* array, which has been sorted, and fills in an array $rank[1:n]$ so that $rank[i]$ is the rank of the ith substring represented in the *substr-rank* array. Each rank is a positive integer, starting from 1, and going up to the number of unique substrings of length $2l$. Substrings with equal values of *left-rank* and *right-rank* receive the same rank. Otherwise, a substring that is lexicographically smaller than another appears earlier in the *substr-rank* array, and it receives a smaller rank. Once the substrings of length $2l$ are re-ranked, line 18 sorts them by rank, preparing for the next iteration of the **while** loop.

Once l reaches or exceeds n and all substrings are sorted, the values in the *index* attributes give the starting positions of the sorted substrings. These indices are precisely the values that constitute the suffix array.

Let's analyze the running time of COMPUTE-SUFFIX-ARRAY. Lines 1–7 take $\Theta(n)$ time. Line 8 takes $O(n \lg n)$ time, using either merge sort (see Section 2.3.1) or heapsort (see Chapter 6). Because the value of l doubles in each iteration of

the **while** loop of lines 10–19, this loop makes $\lceil \lg n \rceil - 1$ iterations. Within each iteration, the call of MAKE-RANKS takes $\Theta(n)$ time, as does the **for** loop of lines 12–17. Line 18, like line 8, takes $O(n \lg n)$ time, using either merge sort or heapsort. Finally, the **for** loop of lines 20–21 takes $\Theta(n)$ time. The total time works out to $O(n \lg^2 n)$.

A simple observation allows us to reduce the running time to $\Theta(n \lg n)$. The values of *left-rank* and *right-rank* being sorted in line 18 are always integers in the range 0 to n. Therefore, radix sort can sort the *substr-rank* array in $\Theta(n)$ time by first running counting sort (see Chapter 8) based on *right-rank* and then running counting sort based on *left-rank*. Now each iteration of the **while** loop of lines 10–19 takes only $\Theta(n)$ time, giving a total time of $\Theta(n \lg n)$.

Exercise 32.5-2 asks you to make a simple modification to COMPUTE-SUFFIX-ARRAY that allows the **while** loop of lines 10–19 to iterate fewer than $\lceil \lg n \rceil - 1$ times for certain inputs.

Computing the *LCP* array

Recall that $LCP[i]$ is defined as the length of the longest common prefix of the $(i-1)$st and ith lexicographically smallest suffixes $T[SA[i-1]:]$ and $T[SA[i]:]$. Because $T[SA[1]:]$ is the lexicographically smallest suffix, we define $LCP[1]$ to be 0.

In order to compute the *LCP* array, we need an array *rank* that is the inverse of the *SA* array, just like the final *rank* array in COMPUTE-SUFFIX-ARRAY: if $SA[i] = j$, then $rank[j] = i$. That is, we have $rank[SA[i]] = i$ for $i = 1, 2, \ldots, n$. For a suffix $T[i:]$, the value of $rank[i]$ gives the position of this suffix in the lexicographically sorted order. Figure 32.11 includes the *rank* array for the `ratatat` example. For example, the suffix `tat` is $T[5:]$. To find this suffix's position in the sorted order, look up $rank[5] = 6$.

To compute the *LCP* array, we will need to determine where in the lexicographically sorted order a suffix appears, but with its first character removed. The *rank* array helps. Consider the ith smallest suffix, which is $T[SA[i]:]$. Dropping its first character gives the suffix $T[SA[i]+1:]$, that is, the suffix starting at position $SA[i]+1$ in the text. The location of this suffix in the sorted order is given by $rank[SA[i]+1]$. For example, for the suffix `atat`, let's see where to find `tat` (`atat` with its first character removed) in the lexicographically sorted order. The suffix `atat` appears in position 2 of the suffix array, and $SA[2] = 4$. Thus, $rank[SA[2]+1] = rank[5] = 6$, and sure enough the suffix `tat` appears in location 6 in the sorted order.

The procedure COMPUTE-LCP on the next page produces the *LCP* array. The following lemma helps show that the procedure is correct.

```
COMPUTE-LCP(T, SA, n)
 1  allocate arrays rank[1 : n] and LCP[1 : n]
 2  for i = 1 to n
 3      rank[SA[i]] = i                  // by definition
 4  LCP[1] = 0                           // also by definition
 5  l = 0                                // initialize length of LCP
 6  for i = 1 to n
 7      if rank[i] > 1
 8          j = SA[rank[i] − 1]          // T[j :] precedes T[i :] lexicographically
 9          m = max {i, j}
10          while m + l ≤ n and T[i + l] == T[j + l]
11              l = l + 1                // next character is in common prefix
12          LCP[rank[i]] = l             // length of LCP of T[j :] and T[i :]
13          if l > 0
14              l = l − 1                // peel off first character of common prefix
15  return LCP
```

Lemma 32.8

Consider suffixes $T[i-1:]$ and $T[i:]$, which appear at positions $rank[i-1]$ and $rank[i]$, respectively, in the lexicographically sorted order of suffixes. If $LCP[rank[i-1]] = l > 1$, then the suffix $T[i:]$, which is $T[i-1:]$ with its first character removed, has $LCP[rank[i]] \geq l - 1$.

Proof The suffix $T[i-1:]$ appears at position $rank[i-1]$ in the lexicographically sorted order. The suffix immediately preceding it in the sorted order appears at position $rank[i-1]-1$ and is $T[SA[rank[i-1]-1]:]$. By assumption and the definition of the LCP array, these two suffixes, $T[SA[rank[i-1]-1]:]$ and $T[i-1:]$, have a longest common prefix of length $l > 1$. Removing the first character from each of these suffixes gives the suffixes $T[SA[rank[i-1]-1]+1:]$ and $T[i:]$, respectively. These suffixes have a longest common prefix of length $l-1$. If $T[SA[rank[i-1]-1]+1:]$ immediately precedes $T[i:]$ in the lexicographically sorted order (that is, if $rank[SA[rank[i-1]-1]+1] = rank[i]-1$), then the lemma is proven.

So now assume that $T[SA[rank[i-1]-1]+1:]$ does not immediately precede $T[i:]$ in the sorted order. Since $T[SA[rank[i-1]-1]:]$ immediately precedes $T[i-1:]$ and they have the same first $l > 1$ characters, $T[SA[rank[i-1]-1]+1:]$ must appear in the sorted order somewhere before $T[i:]$, with one or more other suffixes intervening. Each of these suffixes must start with the same $l-1$ characters as $T[SA[rank[i-1]-1]+1:]$ and $T[i:]$, for otherwise it would appear either before

$T[SA[rank[i-1]-1]+1:]$ or after $T[i:]$. Therefore, whichever suffix appears in position $rank[i]-1$, immediately before $T[i:]$, has at least its first $l-1$ characters in common with $T[i:]$. Thus, $LCP[rank[i]] \geq l-1$. ■

The COMPUTE-LCP procedure works as follows. After allocating the *rank* and *LCP* arrays in line 1, lines 2–3 fill in the *rank* array and line 4 pegs $LCP[1]$ to 0, per the definition of the *LCP* array.

The **for** loop of lines 6–14 fills in the rest of the *LCP* array going by decreasing-length suffixes. That is, it fills the position of the *LCP* array in the order $rank[1]$, $rank[2]$, $rank[3]$, $\ldots$, $rank[n]$, with the assignment occurring in line 12. Upon considering a suffix $T[i:]$, line 8 determines the suffix $T[j:]$ that immediately precedes $T[i:]$ in the lexicographically sorted order. At this point, the longest common prefix of $T[j:]$ and $T[i:]$ has length at least l. This property certainly holds upon the first iteration of the **for** loop, when $l = 0$. Assuming that line 12 sets $LCP[rank[i]]$ correctly, line 14 (which decrements l if it is positive) and Lemma 32.8 maintain this property for the next iteration. The longest common prefix of $T[j:]$ and $T[i:]$ might be even longer than the value of l at the start of the iteration, however. Lines 9–11 increment l for each additional character the prefixes have in common so that it achieves the length of the longest common prefix. The index m is set in line 9 and used in the test in line 10 to make sure that the test $T[i+l] == T[j+l]$ for extending the longest common prefix does not run off the end of the text T. When the **while** loop of lines 10–11 terminates, l is the length of the longest common prefix of $T[j:]$ and $T[i:]$.

As a simple aggregate analysis shows, the COMPUTE-LCP procedure runs in $\Theta(n)$ time. Each of the two **for** loops iterates n times, and so it remains only to bound the total number of iterations by the **while** loop of lines 10–11. Each iteration increases l by 1, and the test $m + l \leq n$ ensures that l is always less than n. Because l has an initial value of 0 and decreases at most $n-1$ times in line 14, line 11 increments l fewer than $2n$ times. Thus, COMPUTE-LCP takes $\Theta(n)$ time.

Exercises

32.5-1

Show the *substr-rank* and *rank* arrays before each iteration of the **while** loop of lines 10–19 and after the last iteration of the **while** loop, the suffix array *SA* returned, and the sorted suffixes when COMPUTE-SUFFIX-ARRAY is run on the text `hippityhoppity`. Use the position of each letter in the alphabet as its ord value, so that ord(`b`) = 2. Then show the *LCP* array after each iteration of the **for** loop of lines 6–14 of COMPUTE-LCP given the text `hippityhoppity` and its suffix array.

32.5-2

For some inputs, the COMPUTE-SUFFIX-ARRAY procedure can produce the correct result with fewer than $\lceil \lg n \rceil - 1$ iterations of the **while** loop of lines 10–19. Modify COMPUTE-SUFFIX-ARRAY (and, if necessary, MAKE-RANKS) so that the procedure can stop before making all $\lceil \lg n \rceil - 1$ iterations in some cases. Describe an input that allows the procedure to make $O(1)$ iterations. Describe an input that forces the procedure to make the maximum number of iterations.

32.5-3

Given two texts, T_1 of length n_1 and T_2 of length n_2, show how to use the suffix array and longest common prefix array to find all of the ***longest common substrings***, that is, the longest substrings that appear in both T_1 and T_2. Your algorithm should run in $O(n \lg n + kl)$ time, where $n = n_1 + n_2$ and there are k such longest substrings, each with length l.

32.5-4

Professor Markram proposes the following method to find the longest palindromes in a string $T[1:n]$ by using its suffix array and LCP array. (Recall from Problem 14-2 that a palindrome is a nonempty string that reads the same forward and backward.)

> Let `@` be a character that does not appear in T. Construct the text T' as the concatenation of T, `@`, and the reverse of T. Denote the length of T' by $n' = 2n + 1$. Create the suffix array SA and LCP array LCP for T'. Since the indices for a palindrome and its reverse appear in consecutive positions in the suffix array, find the entries with the maximum LCP value $LCP[i]$ such that $SA[i-1] = n' - SA[i] - LCP[i] + 2$. (This constraint prevents a substring—and its reverse—from being construed as a palindrome unless it really is one.) For each such index i, one of the longest palindromes is $T'[SA[i] : SA[i] + LCP[i] - 1]$.

For example, if the text T is `unreferenced`, with $n = 12$, then the text T' is `unreferenced@decnerefernu`, with $n' = 25$ and the following suffix array and LCP array:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$T'[i]$	u	n	r	e	f	e	r	e	n	c	e	d	@	d	e	c	n	e	r	e	f	e	r	n	u
$SA[i]$	13	10	16	12	14	15	11	4	20	8	18	6	22	5	21	9	17	2	24	3	19	7	23	25	1
$LCP[i]$	0	0	1	0	1	0	1	1	4	1	1	3	2	0	3	0	1	1	1	0	5	2	1	0	1

The maximum LCP value is achieved at $LCP[21] = 5$, and $SA[20] = 3 = n' - SA[21] - LCP[21] + 2$. The suffixes of T' starting at indices $SA[20]$ and $SA[21]$ are `referenced@decnerefernu` and `refernu`, both of which start with the length-5 palindrome `refer`.

Alas, this method is not foolproof. Give an input string T that causes this method to give results that are shorter than the longest palindrome contained within T, and explain why your input causes the method to fail.

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 $\rho(P[1:i])$ 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 $\max\{\rho(P[1:i]) : 1 \le i \le m\}$. 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 procedure REPETITION-MATCHER correctly finds all occurrences of pattern $P[1:m]$ in text $T[1:n]$ in $O(\rho^*(P)n + m)$ time. (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.)

```
REPETITION-MATCHER(T, P, n, m)
 1  k = 1 + ρ*(P)
 2  q = 0
 3  s = 0
 4  while s ≤ n − m
 5      if T[s + q + 1] == P[q + 1]
 6          q = q + 1
 7          if q == m
 8              print "Pattern occurs with shift" s
 9      if q == m or T[s + q + 1] ≠ P[q + 1]
10          s = s + max{1, ⌈q/k⌉}
11          q = 0
```

32-2 A linear-time suffix-array algorithm

In this problem, you will develop and analyze a linear-time divide-and-conquer algorithm to compute the suffix array of a text $T[1:n]$. As in Section 32.5, assume that each character in the text is represented by an underlying encoding, which is a positive integer.

The idea behind the linear-time algorithm is to compute the suffix array for the suffixes starting at $2/3$ of the positions in the text, recursing as needed, use the resulting information to sort the suffixes starting at the remaining $1/3$ of the positions, and then merge the sorted information in linear time to produce the full suffix array.

For $i = 1, 2, \ldots, n$, if i mod 3 equals 1 or 2, then i is a ***sample position***, and the suffixes starting at such positions are ***sample suffixes***. Positions $3, 6, 9, \ldots$ are ***nonsample positions***, and the suffixes starting at nonsample positions are ***nonsample suffixes***.

The algorithm sorts the sample suffixes, sorts the nonsample suffixes (aided by the result of sorting the sample suffixes), and merges the sorted sample and nonsample suffixes. Using the example text $T = \texttt{bippityboppityboo}$, here is the algorithm in detail, listing substeps of each of the above steps:

1. The sample suffixes comprise about $2/3$ of the suffixes. Sort them by the following substeps, which work with a heavily modified version of T and may require recursion. In part (a) of this problem on page 999, you will show that the orders of the suffixes of T and the suffixes of the modified version of T are the same.

   A. Construct two texts P_1 and P_2 made up of "metacharacters" that are actually substrings of three consecutive characters from T. We delimit each such metacharacter with parentheses. Construct

   $$P_1 = (T[1:3])\,(T[4:6])\,(T[7:9])\cdots(T[n':n'+2])\,,$$

   where n' is the largest integer congruent to 1, modulo 3, that is less than or equal to n and T is extended beyond position n with the special character $\varnothing$, with encoding 0. With the example text $T = \texttt{bippityboppityboo}$, we get that

   $$P_1 = (\texttt{bip})\,(\texttt{pit})\,(\texttt{ybo})\,(\texttt{ppi})\,(\texttt{tyb})\,(\texttt{oo}\varnothing)\,.$$

   Similarly, construct

   $$P_2 = (T[2:4])\,(T[5:7])\,(T[8:10])\cdots(T[n'':n''+2])\,,$$

   where n'' is the largest integer congruent to 2, modulo 3, that is less than or equal to n. For our example, we have

   $$P_2 = (\texttt{ipp})\,(\texttt{ity})\,(\texttt{bop})\,(\texttt{pit})\,(\texttt{ybo})\,(\texttt{o}\varnothing\varnothing)\,.$$

position in T	1	4	7	10	13	16	2	5	8	11	14	17
metacharacter in P	(bip)	(pit)	(ybo)	(ppi)	(tyb)	(oo∅)	(ipp)	(ity)	(bop)	(pit)	(ybo)	(o∅∅)
character in P'	1	7	10	8	9	6	3	4	2	7	10	5
position in P'	1	2	3	4	5	6	7	8	9	10	11	12
$SA_{P'}$	1	9	7	8	12	6	10	2	4	5	11	3
positions in T of sorted sample suffixes of T	1	8	2	5	17	16	11	4	10	13	14	7

Figure 32.15 Computed values when sorting the sample suffixes of the linear-time suffix-array algorithm for the text $T =$ `bippityboppityboo`.

If n is a multiple of 3, append the metacharacter $(\varnothing\varnothing\varnothing)$ to the end of P_1. In this way, P_1 is guaranteed to end with a metacharacter containing $\varnothing$. (This property helps in part (a) of this problem.) The text P_2 may or may not end with a metacharacter containing $\varnothing$.

B. Concatenate P_1 and P_2 to form a new text P. Figure 32.15 shows P for our example, along with the corresponding positions of T.

C. Sort and rank the unique metacharacters of P, with ranks starting from 1. In the example, P has 10 unique metacharacters: in sorted order, they are (`bip`), (`bop`), (`ipp`), (`ity`), (`o`$\varnothing\varnothing$), (`oo`$\varnothing$), (`pit`), (`ppi`), (`tyb`), (`ybo`). The metacharacters (`pit`) and (`ybo`) each appear twice.

D. As Figure 32.15 shows, construct a new "text" P' by renaming each metacharacter in P by its rank. If P contains k unique metacharacters, then each "character" in P' is an integer from 1 to k. The suffix arrays for P and P' are identical.

E. Compute the suffix array $SA_{P'}$ of P'. If the characters of P' (i.e., the ranks of metacharacters in P) are unique, then you can compute its suffix array directly, since the ordering of the individual characters gives the suffix array. Otherwise, recurse to compute the suffix array of P', treating the ranks in P' as the input characters in the recursive call. Figure 32.15 shows the suffix array $SA_{P'}$ for our example. Since the number of metacharacters in P, and hence the length of P', is approximately $2n/3$, this recursive subproblem is smaller than the current problem.

F. From $SA_{P'}$ and the positions in T corresponding to the sample positions, compute the list of positions of the sorted sample suffixes of the original text T. Figure 32.15 shows the list of positions in T of the sorted sample suffixes in our example.

2. The nonsample suffixes comprise about $1/3$ of the suffixes. Using the sorted sample suffixes, sort the nonsample suffixes by the following substeps.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$T[i]$	b	i	p	p	i	t	y	b	o	p	p	i	t	y	b	o	o	∅	∅
r_i	1	3	□	8	4	□	12	2	□	9	7	□	10	11	□	6	5	0	0

Figure 32.16 The ranks r_1 through r_{n+3} for the text $T =$ `bippityboppityboo` with $n = 17$.

G. Extending the text T by the two special characters ∅∅, so that T now has $n + 2$ characters, consider each suffix $T[i:]$ for $i = 1, 2, \ldots, n + 2$. Assign a rank r_i to each suffix $T[i:]$. For the two special characters ∅∅, set $r_{n+1} = r_{n+2} = 0$. For the sample positions of T, base the rank on the list of sorted sample positions of T. The rank is currently undefined for the nonsample positions of T. For these positions, set $r_i = \square$. Figure 32.16 shows the ranks for $T =$ `bippityboppityboo` with $n = 17$.

H. Sort the nonsample suffixes by comparing tuples $(T[i], r_{i+1})$. In our example, we get $T[15:] < T[12\ \ :] < T[9\ \ :] < T[3\ \ :] < T[6\ \ :]$ because $(\texttt{b}, 6) < (\texttt{i}, 10) < (\texttt{o}, 9) < (\texttt{p}, 8) < (\texttt{t}, 12)$.

3. Merge the sorted sets of suffixes. From the sorted set of suffixes, determine the suffix array of T.

This completes the description of a linear-time algorithm for computing suffix arrays. The following parts of this problem ask you to show that certain steps of this algorithm are correct and to analyze the algorithm's running time.

a. Define a ***nonempty suffix*** at position i of the text P created in substep B as all metacharacters from position i of P up to and including the first metacharacter of P in which ∅ appears or the end of P. In the example shown in Figure 32.15, the nonempty suffixes of P starting at positions 1, 4, and 11 of P are (`bip`) (`pit`) (`ybo`) (`ppi`) (`tyb`) (`oo`∅), (`ppi`) (`tyb`) (`oo`∅), and (`ybo`) (`o`∅∅), respectively. Prove that the order of suffixes of P is the same as the order of its nonempty suffixes. Conclude that the order of suffixes of P gives the order of the sample suffixes of T. (*Hint:* If P contains duplicate metacharacters, consider separately the cases in which two suffixes both start in P_1, both start in P_2, and one starts in P_1 and the other starts in P_2. Use the property that ∅ appears in the last metacharacter of P_1.)

b. Show how to perform substep C in $\Theta(n)$ time, bearing in mind that in a recursive call, the characters in T are actually ranks in P' in the caller.

c. Argue that the tuples in substep H are unique. Then show how to perform this substep in $\Theta(n)$ time.

d. Consider two suffixes $T[i:]$ and $T[j:]$, where $T[i:]$ is a sample suffix and $T[j:]$ is a nonsample suffix. Show how to determine in $\Theta(1)$ time whether $T[i:]$ is lexicographically smaller than $T[j:]$. (*Hint:* Consider separately the cases in which $i \bmod 3 = 1$ and $i \bmod 3 = 2$. Compare tuples whose elements are characters in T and ranks as shown in Figure 32.16. The number of elements per tuple may depend on whether $i \bmod 3$ equals 1 or 2.) Conclude that step 3 can be performed in $\Theta(n)$ time.

e. Justify the recurrence $T(n) \le T(2n/3 + 2) + \Theta(n)$ for the running time of the full algorithm, and show that its solution is $O(n)$. Conclude that the algorithm runs in $\Theta(n)$ time.

32-3 Burrows-Wheeler transform

The ***Burrows-Wheeler transform***, or ***BWT***, for a text T is defined as follows. First, append a new character that compares as lexicographically less than every character of T, and denote this character by `$` and the resulting string by T'. Letting n be the length of T', create n rows of characters, where each row is one of the n cyclic rotations of T'. Next, sort the rows lexicographically. The BWT is then the string of n characters in the rightmost column, read top to bottom.

For example, let $T =$ `rutabaga`, so that $T' =$ `rutabaga$`. The cyclic rotations are

```
rutabaga$
utabaga$r
tabaga$ru
abaga$rut
baga$ruta
aga$rutab
ga$rutaba
a$rutabag
$rutabaga
```

Sorting the rows and numbering the sorted rows gives

```
1  $rutabaga
2  a$rutabag
3  abaga$rut
4  aga$rutab
5  baga$ruta
6  ga$rutaba
7  rutabaga$
8  tabaga$ru
9  utabaga$r
```

The BWT is the rightmost column, `agtbaa$ur`. (The row numbering will be helpful in understanding how to compute the inverse BWT.)

The BWT has applications in bioinformatics, and it can also be a step in text compression. That is because it tends to place identical characters together, as in the BWT of `rutabaga`, which places two of the instances of `a` together. When identical characters are placed together, or even nearby, additional means of compressing become available. Following the BWT, combinations of move-to-front encoding, run-length encoding, and Huffman coding (see Section 15.3) can provide significant text compression. Compression ratios with the BWT tend improve as the text length increases.

a. Given the suffix array for T', show how to compute the BWT in $\Theta(n)$ time.

In order to decompress, the BWT must be invertible. Assuming that the alphabet size is constant, the inverse BWT can be computed in $\Theta(n)$ time from the BWT. Let's look at the BWT of `rutabaga`, denoting it by $BWT[1:n]$. Each character in the BWT has a unique lexicographic rank from 1 to n. Denote the rank of $BWT[i]$ by $rank[i]$. If a character appears multiple times in the BWT, each instance of the character has a rank 1 greater than the previous instance of the character. Here are *BWT* and *rank* for `rutabaga`:

i	1	2	3	4	5	6	7	8	9
$BWT[i]$	a	g	t	b	a	a	$	u	r
$rank[i]$	2	6	8	5	3	4	1	9	7

For example, $rank[1] = 2$ because $BWT[1] =$ `a` and the only character that precedes the first `a` lexicographically is `$` (which we defined to precede all other characters, so that `$` has rank 1). Next, we have $rank[2] = 6$ because $BWT[2] =$ `g` and five characters in the BWT precede `g` lexicographically: `$`, the three instances of `a`, and `b`. Jumping ahead to $rank[5] = 3$, that is because $BWT[5] =$ `a`, and because this `a` is the second instance of `a` in the BWT, its *rank* value is 1 greater than the *rank* value for the previous instance of `a`, in position 1.

There is enough information in *BWT* and *rank* to reconstruct T' from back to front. Suppose that you know the rank r of a character c in T'. Then c is the first character in row r of the sorted cyclic rotations. The last character in row r must be the character that precedes c in T'. But you know which character is the last character in row r, because it is $BWT[r]$. To reconstruct T' from back to front, start with `$`, which you can find in *BWT*. Then work backward using *BWT* and *rank* to reconstruct T'.

Let's see how this strategy works for `rutabaga`. The last character of T', `$`, appears in position 7 of *BWT*. Since $rank[7] = 1$, row 1 of the sorted cyclic rotations of T' begins with `$`. The character that precedes `$` in T' is the last character in row 1, which is $BWT[1]$: `a`. Now we know that the last two characters of T'

are a$. Looking up *rank*[1], it equals 2, so that row 2 of the sorted cyclic rotations of T' begins with a. The last character in row 2 precedes a in T', and that character is $BWT[2] = \texttt{g}$. Now we know that the last three characters of T' are ga$. Continuing on, we have $rank[2] = 6$, so that row 6 of the sorted cyclic rotations begins with g. The character preceding g in T' is $BWT[6] = \texttt{a}$, and so the last four characters of T' are aga$. Because $rank[6] = 4$, a begins row 4 of the sorted cyclic rotations of T'. The character preceding a in T' is the last character in row 4, $BWT[4] = \texttt{b}$, and the last five characters of T' are baga$. And so on, until all n characters of T' have been identified, from back to front.

b. Given the array $BWT[1:n]$, write pseudcode to compute the array $rank[1:n]$ in $\Theta(n)$ time, assuming that the alphabet size is constant.

c. Given the arrays $BWT[1:n]$ and $rank[1:n]$, write pseudocode to compute T' in $\Theta(n)$ time.

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 [267] was invented independently by Knuth and Pratt and by Morris, but they published their work jointly. Matiyasevich [317] earlier discovered a similar algorithm, which applied only to an alphabet with two characters and was specified for a Turing machine with a two-dimensional tape. Reingold, Urban, and Gries [377] give an alternative treatment of the Knuth-Morris-Pratt algorithm. The Rabin-Karp algorithm was proposed by Karp and Rabin [250]. Galil and Seiferas [173] give an interesting deterministic linear-time string-matching algorithm that uses only $O(1)$ space beyond that required to store the pattern and text.

The suffix-array algorithm in Section 32.5 is by Manber and Myers [312], who first proposed the notion of suffix arrays. The linear-time algorithm to compute the longest common prefix array presented here is by Kasai et al. [252]. Problem 32-2 is based on the DC3 algorithm by Kärkkäinen, Sanders, and Burkhardt [245]. For a survey of suffix-array algorithms, see the article by Puglisi, Smyth, and Turpin [370]. To learn more about the Burrows-Wheeler transform from Problem 32-3, see the articles by Burrows and Wheeler [78] and Manzini [314].

33 Machine-Learning Algorithms

Machine learning may be viewed as a subfield of artificial intelligence. Broadly speaking, artificial intelligence aims to enable computers to carry out complex perception and information-processing tasks with human-like performance. The field of AI is vast and uses many different algorithmic methods.

Machine learning is rich and fascinating, with strong ties to statistics and optimization. Technology today produces enormous amounts of data, providing myriad opportunities for machine-learning algorithms to formulate and test hypotheses about patterns within the data. These hypotheses can then be used to make predictions about the characteristics or classifications in new data. Because machine learning is particularly good with challenging tasks involving uncertainty, where observed data follows unknown rules, it has markedly transformed fields such as medicine, advertising, and speech recognition.

This chapter presents three important machine-learning algorithms: k-means clustering, multiplicative weights, and gradient descent. You can view each of these tasks as a learning problem, whereby an algorithm uses the data collected so far to produce a hypothesis that describes the regularities learned and/or makes predictions about new data. The boundaries of machine learning are imprecise and evolving—some might say that the k-means clustering algorithm should be called "data science" and not "machine learning," and gradient descent, though an immensely important algorithm for machine learning, also has a multitude of applications outside of machine learning (most notably for optimization problems).

Machine learning typically starts with a ***training phase*** followed by a ***prediction phase*** in which predictions are made about new data. For ***online learning***, the training and prediction phases are intermingled. The training phase takes as input ***training data***, where each input data point has an associated output or ***label***; the label might be a category name or some real-valued attribute. It then produces as an output one or more ***hypotheses*** about how the labels depend on the attributes of the input data points. Hypotheses can take many forms, typically some type of formula or algorithm. The learning algorithm used is often a form of gradient

descent. The prediction phase then uses the hypothesis on new data in order to make ***predictions*** regarding the labels of new data points.

The type of learning just described is known as ***supervised learning***, since it starts with a set of inputs that are each labeled. As an example, consider a machine-learning algorithm to recognize spam emails. The training data comprises a collection of emails, each of which is labeled either "spam" or "not spam." The machine-learning algorithm frames a hypothesis, possibly a rule of the form "if an email has one of a set of words, then it is likely to be spam." Or it might learn rules that assign a spam score to each word and then evaluates a document by the sum of the spam scores of its constituent words, so that a document with a total score above a certain threshold value is classified as spam. The machine-learning algorithm can then predict whether a new email is spam or not.

A second form of machine learning is ***unsupervised learning***, where the training data is unlabeled, as in the clustering problem of Section 33.1. Here the machine-learning algorithm produces hypotheses regarding the centers of groups of input data points.

A third form of machine learning (not covered further here) is ***reinforcement learning***, where the machine-learning algorithm takes actions in an environment, receives feedback for those actions from the environment, and then updates its model of the environment based on the feedback. The learner is in an environment that has some state, and the actions of the learner have an effect on that state. Reinforcement learning is a natural choice for situations such as game playing or operating a self-driving car.

Sometimes the goal in a supervised machine-learning application is not making accurate predictions of labels for new examples, but rather performing causal ***inference***: finding an explanatory model that describes how the various features of an input data point affect its associated label. Finding a model that fits a given set of training data well can be tricky. It may involve sophisticated optimization methods that need to balance between producing a hypothesis that fits the data well and producing a hypothesis that is simple.

This chapter focuses on three problem domains: finding hypotheses that group the input data points well (using a clustering algorithm), learning which predictors (experts) to rely upon for making predictions in an online learning problem (using the multiplicative-weights algorithm), and fitting a model to data (using gradient descent).

Section 33.1 considers the clustering problem: how to divide a given set of n training data points into a given number k of groups, or "clusters," based on a measure of how similar (or more accurately, how dissimilar) points are to each other. The approach is iterative, beginning with an arbitrary initial clustering and incorporating successive improvements until no further improvements occur. Clustering

is often used as an initial step when working on a machine-learning problem to discover what structure exists in the data.

Section 33.2 shows how to make online predictions quite accurately when you have a set of predictors, often called "experts," to rely on, many of which might be poor predictors, but some of which are good predictors. At first, you do not know which predictors are poor and which are good. The goal is to make predictions on new examples that are nearly as good as the predictions made by the best predictor. We study an effective multiplicative-weights prediction method that associates a positive real weight with each predictor and multiplicatively decreases the weights associated with predictors when they make poor predictions. The model in this section is online (see Chapter 27): at each step, we do not know anything about the future examples. In addition, we are able to make predictions even in the presence of adversarial experts, who are collaborating against us, a situation that actually happens in game-playing settings.

Finally, Section 33.3 introduces gradient descent, a powerful optimization technique used to find parameter settings in machine-learning models. Gradient descent also has many applications outside of machine learning. Intuitively, gradient descent finds the value that produces a local minimum for a function by "walking downhill." In a learning application, a "downhill step" is a step that adjusts hypothesis parameters so that the hypothesis does better on the given set of labeled examples.

This chapter makes extensive use of vectors. In contrast to the rest of the book, vector names in this chapter appear in boldface, such as $\mathbf{x}$, to more clearly delineate which quantities are vectors. Components of vectors do not appear in boldface, so if vector $\mathbf{x}$ has d dimensions, we might write $\mathbf{x} = (x_1, x_2, \ldots, x_d)$.

33.1 Clustering

Suppose that you have a large number of data points (examples), and you wish to group them into classes based on how similar they are to each other. For example, each data point might represent a celestial star, giving its temperature, size, and spectral characteristics. Or, each data point might represent a fragment of recorded speech. Grouping these speech fragments appropriately might reveal the set of accents of the fragments. Once a grouping of the training data points is found, new data can be placed into an appropriate group, facilitating star-type recognition or speech recognition.

These situations, along with many others, fall under the umbrella of clustering. The input to a ***clustering*** problem is a set of n examples (objects) and an integer k, with the goal of dividing the examples into at most k disjoint clusters such that

the examples in each cluster are similar to each other. The clustering problem has several variations. For example, the integer k might not be given, but instead arises out of the clustering procedure. In this section we presume that k is given.

Feature vectors and similarity

Let's formally define the clustering problem. The input is a set of n ***examples***. Each example has a set of ***attributes*** in common with all other examples, though the attribute values may vary among examples. For example, the clustering problem shown in Figure 33.1 clusters $n = 49$ examples—48 state capitals plus the District of Columbia—into $k = 4$ clusters. Each example has two attributes: the latitude and longitude of the capital. In a given clustering problem, each example has d attributes, with an example $\mathbf{x}$ specified by a d-dimensional ***feature vector***

$$\mathbf{x} = (x_1, x_2, \ldots, x_d) .$$

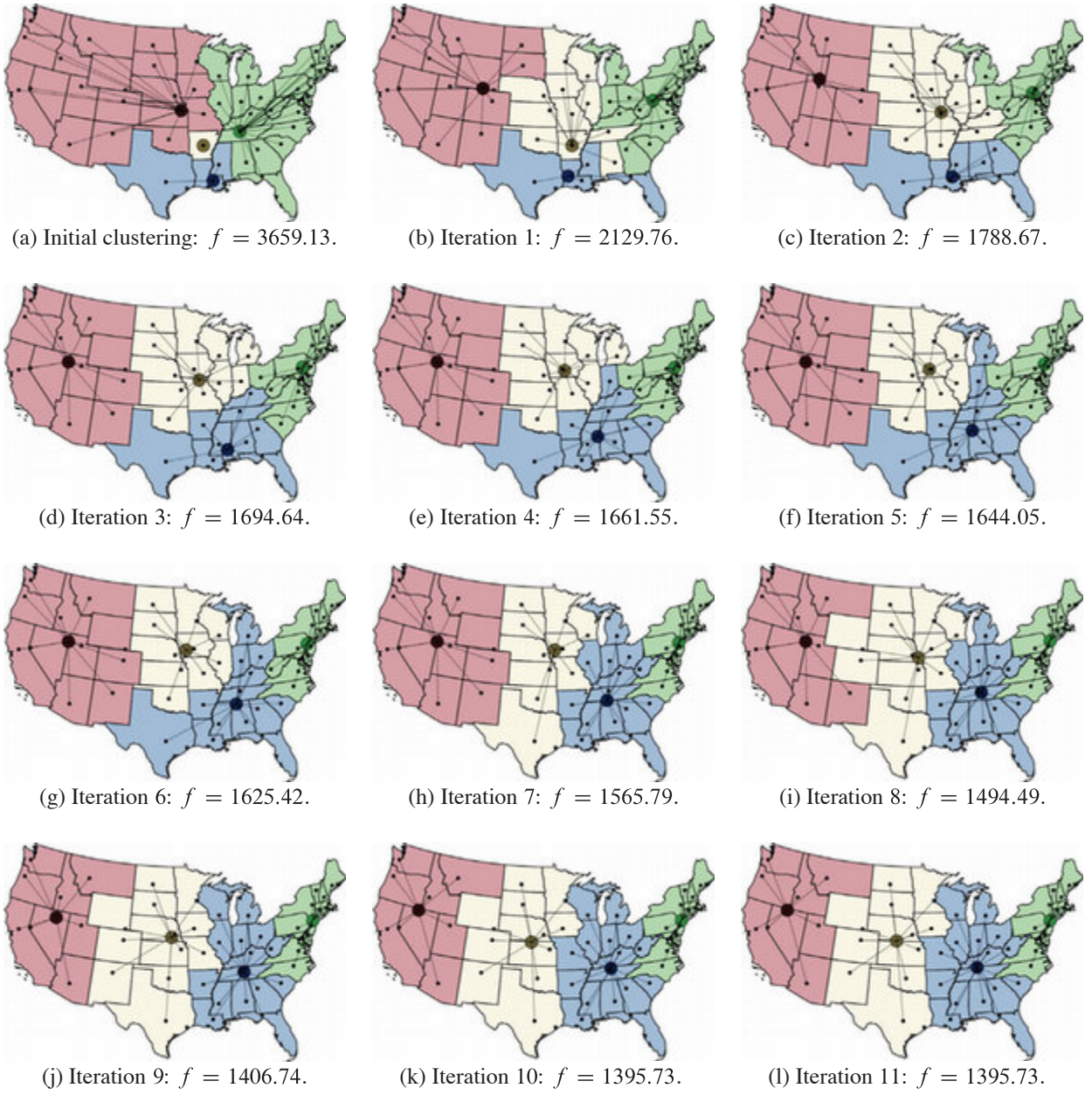
Here, x_a for $a = 1, 2, \ldots, d$ is a real number giving the value of attribute a for example $\mathbf{x}$. We call $\mathbf{x}$ the ***point*** in $\mathbb{R}^d$ representing the example. For the example in Figure 33.1, each capital $\mathbf{x}$ has its latitude in x_1 and its longitude in x_2.

In order to cluster similar points together, we need to define similarity. Instead, let's define the opposite: the ***dissimilarity*** $\Delta(\mathbf{x}, \mathbf{y})$ of points $\mathbf{x}$ and $\mathbf{y}$ is the squared Euclidean distance between them:

$$\begin{aligned} \Delta(\mathbf{x}, \mathbf{y}) &= \|\mathbf{x} - \mathbf{y}\|^2 \\ &= \sum_{a=1}^{d} (x_a - y_a)^2 . \end{aligned} \tag{33.1}$$

Of course, for $\Delta(\mathbf{x}, \mathbf{y})$ to be well defined, all attribute values must be present. If any are missing, then you might just ignore that example, or you could fill in a missing attribute value with the median value for that attribute.

The attribute values are often "messy" in other ways, so that some "data cleaning" is necessary before the clustering algorithm is run. For example, the scale of attribute values can vary widely across attributes. In the example of Figure 33.1, the scales of the two attributes vary by a factor of 2, since latitude ranges from -90 to $+90$ degrees but longitude ranges from -180 to $+180$ degrees. You can imagine other scenarios where the differences in scales are even greater. If the examples contain information about students, one attribute might be grade-point average but another might be family income. Therefore, the attribute values are usually scaled or normalized, so that no single attribute can dominate the others when computing dissimilarities. One way to do so is by scaling attribute values with a linear transform so that the minimum value becomes 0 and the maximum value becomes 1. If the attribute values are binary values, then no scaling may be needed. Another



(a) Initial clustering: $f = 3659.13$. (b) Iteration 1: $f = 2129.76$. (c) Iteration 2: $f = 1788.67$.

(d) Iteration 3: $f = 1694.64$. (e) Iteration 4: $f = 1661.55$. (f) Iteration 5: $f = 1644.05$.

(g) Iteration 6: $f = 1625.42$. (h) Iteration 7: $f = 1565.79$. (i) Iteration 8: $f = 1494.49$.

(j) Iteration 9: $f = 1406.74$. (k) Iteration 10: $f = 1395.73$. (l) Iteration 11: $f = 1395.73$.

Figure 33.1 The iterations of Lloyd's procedure when clustering the capitals of the lower 48 states and the District of Columbia into $k = 4$ clusters. Each capital has two attributes: latitude and longitude. Each iteration reduces the value f, measuring the sum of squares of distances of all capitals to their cluster centers, until the value of f does not change. **(a)** The initial four clusters, with the capitals of Arkansas, Kansas, Louisiana, and Tennessee chosen as centers. **(b)–(k)** Iterations of Lloyd's procedure. **(l)** The 11th iteration results in the same value of f as the 10th iteration in part (k), and so the procedure terminates.

option is scaling so that the values for each attribute have mean 0 and unit variance. Sometimes it makes sense to choose the same scaling rule for several related attributes (for example, if they are lengths measured to the same scale).

Also, the choice of dissimilarity measure is somewhat arbitrary. The use of the sum of squared differences as in equation (33.1) is not required, but it is a conventional choice and mathematically convenient. For the example of Figure 33.1, you might use the actual distance between capitals rather than equation (33.1).

Clusterings

With the notion of similarity (actually, *dis*similarity) defined, let's see how to define clusters of similar points. Let S denote the given set of n points in $\mathbb{R}^d$. In some applications the points are not necessarily distinct, so that S is a multiset rather than a set.

Because the goal is to create k clusters, we define a ***k-clustering*** of S as a decomposition of S into a sequence $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ of k disjoint subsets, or ***clusters***, so that

$$S = S^{(1)} \cup S^{(2)} \cup \cdots \cup S^{(k)} .$$

A cluster may be empty, for example if $k > 1$ but all of the points in S have the same attribute values.

There are many ways to define a k-clustering of S and many ways to evaluate the quality of a given k-clustering. We consider here only k-clusterings of S that are defined by a sequence C of k ***centers***

$$C = \langle \mathbf{c^{(1)}}, \mathbf{c^{(2)}}, \ldots, \mathbf{c^{(k)}} \rangle ,$$

where each center is a point in $\mathbb{R}^d$, and the ***nearest-center rule*** says that a point $\mathbf{x}$ may belong to cluster $S^{(\ell)}$ if the center of no other cluster is closer to $\mathbf{x}$ than the center $\mathbf{c^{(\ell)}}$ of $S^{(\ell)}$:

$$\mathbf{x} \in S^{(\ell)} \text{ only if } \Delta(\mathbf{x}, \mathbf{c^{(\ell)}}) = \min \{\Delta(\mathbf{x}, \mathbf{c^{(j)}}) : 1 \le j \le k\} .$$

A center can be anywhere, and not necessarily a point in S.

Ties are possible and must be broken so that each point lies in exactly one cluster. In general, ties may be broken arbitrarily, although we'll need the property that we never change which cluster a point $\mathbf{x}$ is assigned to unless the distance from $\mathbf{x}$ to its new cluster center is *strictly smaller* than the distance from $\mathbf{x}$ to its old cluster center. That is, if the current cluster has a center that is one of the closest cluster centers to $\mathbf{x}$, then don't change which cluster $\mathbf{x}$ is assigned to.

The ***k-means problem*** is then the following: given a set S of n points and a positive integer k, find a sequence $C = \langle \mathbf{c^{(1)}}, \mathbf{c^{(2)}}, \ldots, \mathbf{c^{(k)}} \rangle$ of k center points

minimizing the sum $f(S, C)$ of the squared distance from each point to its nearest center, where

$$\begin{aligned} f(S, C) &= \sum_{\mathbf{x} \in S} \min \{\Delta(\mathbf{x}, \mathbf{c}^{(j)}) : 1 \le j \le k\} \\ &= \sum_{\ell=1}^{k} \sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c}^{(\ell)}) \, . \end{aligned} \tag{33.2}$$

In the second line, the k-clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ is defined by the centers C and the nearest-center rule. See Exercise 33.1-1 for an alternative formulation based on pairwise interpoint distances.

Is there a polynomial-time algorithm for the k-means problem? Probably not, because it is NP-hard [310]. As we'll see in Chapter 34, NP-hard problems have no known polynomial-time algorithm, but nobody has ever proven that polynomial-time algorithms for NP-hard problems cannot exist. Although we know of no polynomial-time algorithm that finds the global minimum over all clusterings (according to equation (33.2)), we *can* find a local minimum.

Lloyd [304] proposed a simple procedure that finds a sequence C of k centers that yields a local minimum of $f(S, C)$. A local minimum in the k-means problem satisfies two simple properties: each cluster has an optimal center (defined below), and each point is assigned to the cluster (or one of the clusters) with the closest center. Lloyd's procedure finds a good clustering—possibly optimal—that satisfies these two properties. These properties are necessary, but not sufficient, for optimality.

Optimal center for a given cluster

In an optimal solution to the k-means problem, each center point must be the ***centroid***, or ***mean***, of the points in its cluster. The centroid is a d-dimensional point, where the value in each dimension is the mean of the values of all the points in the cluster in that dimension (that is, the mean of the corresponding attribute values in the cluster). That is, if $\mathbf{c}^{(\ell)}$ is the centroid for cluster $S^{(\ell)}$, then for attributes $a = 1, 2, \ldots, d$, we have

$$c_a^{(\ell)} = \frac{1}{|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} x_a \, .$$

Over all attributes, we write

$$\mathbf{c}^{(\ell)} = \frac{1}{|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} \mathbf{x} \, . \tag{33.3}$$

Theorem 33.1
Given a nonempty cluster $S^{(\ell)}$, its centroid (or mean) is the unique choice for the cluster center $\mathbf{c^{(\ell)}} \in \mathbb{R}^d$ that minimizes

$$\sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c^{(\ell)}}) \ .$$

Proof We wish to minimize, by choosing $\mathbf{c^{(\ell)}} \in \mathbb{R}^d$, the sum

$$\begin{aligned} \sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c^{(\ell)}}) &= \sum_{\mathbf{x} \in S^{(\ell)}} \sum_{a=1}^{d} (x_a - c_a^{(\ell)})^2 \\ &= \sum_{a=1}^{d} \left(\sum_{\mathbf{x} \in S^{(\ell)}} x_a^2 - 2 \left(\sum_{\mathbf{x} \in S^{(\ell)}} x_a \right) c_a^{(\ell)} + \left| S^{(\ell)} \right| (c_a^{(\ell)})^2 \right) \ . \end{aligned}$$

For each attribute a, the term summed is a convex quadratic function in $c_a^{(\ell)}$. To minimize this function, take its derivative with respect to $c_a^{(\ell)}$ and set it to 0:

$$-2 \sum_{\mathbf{x} \in S^{(\ell)}} x_a + 2 \left| S^{(\ell)} \right| c_a^{(\ell)} = 0$$

or, equivalently,

$$c_a^{(\ell)} = \frac{1}{|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} x_a \ .$$

Since the minimum is obtained uniquely when each coordinate of $c_a^{(\ell)}$ is the average of the corresponding coordinate for $\mathbf{x} \in S^{(\ell)}$, the overall minimum is obtained when $\mathbf{c^{(\ell)}}$ is the centroid of the points $\mathbf{x}$, as in equation (33.3). ■

Optimal clusters for given centers

The following theorem shows that the nearest-center rule—assigning each point $\mathbf{x}$ to one of the clusters whose center is nearest to $\mathbf{x}$—yields an optimal solution to the k-means problem.

Theorem 33.2
Given a set S of n points and a sequence $\langle \mathbf{c^{(1)}}, \mathbf{c^{(2)}}, \ldots, \mathbf{c^{(k)}} \rangle$ of k centers, a clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ minimizes

$$\sum_{\ell=1}^{k} \sum_{\mathbf{x} \in S^{(\ell)}} \Delta(\mathbf{x}, \mathbf{c^{(\ell)}}) \tag{33.4}$$

if and only if it assigns each point $\mathbf{x} \in S$ to a cluster $S^{(\ell)}$ that minimizes $\Delta(\mathbf{x}, \mathbf{c^{(\ell)}})$.

Proof The proof is straightforward: each point $\mathbf{x} \in S$ contributes exactly once to the sum (33.4), and choosing to put $\mathbf{x}$ in a cluster whose center is nearest minimizes the contribution from $\mathbf{x}$. ■

Lloyd's procedure

Lloyd's procedure just iterates two operations—assigning points to clusters based on the nearest-center rule, followed by recomputing the centers of clusters to be their centroids—until the results converge. Here is Lloyd's procedure:

Input: A set S of points in $\mathbb{R}^d$, and a positive integer k.

Output: A k-clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ of S with a sequence of centers $\langle \mathbf{c^{(1)}}, \mathbf{c^{(2)}}, \ldots, \mathbf{c^{(k)}} \rangle$.

1. **Initialize centers:** Generate an initial sequence $\langle \mathbf{c^{(1)}}, \mathbf{c^{(2)}}, \ldots, \mathbf{c^{(k)}} \rangle$ of k centers by picking k points independently from S at random. (If the points are not necessarily distinct, see Exercise 33.1-3.) Assign all points to cluster $S^{(1)}$ to begin.
2. **Assign points to clusters:** Use the nearest-center rule to define the clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$. That is, assign each point $\mathbf{x} \in S$ to a cluster $S^{(\ell)}$ having a nearest center (breaking ties arbitrarily, but not changing the assignment for a point $\mathbf{x}$ unless the new cluster center is strictly closer to $\mathbf{x}$ than the old one).
3. **Stop if no change:** If step 2 did not change the assignments of any points to clusters, then stop and return the clustering $\langle S^{(1)}, S^{(2)}, \ldots, S^{(k)} \rangle$ and the associated centers $\langle \mathbf{c^{(1)}}, \mathbf{c^{(2)}}, \ldots, \mathbf{c^{(k)}} \rangle$. Otherwise, go to step 4.
4. **Recompute centers as centroids:** For $\ell = 1, 2, \ldots, k$, compute the center $\mathbf{c^{(\ell)}}$ of cluster $S^{(\ell)}$ as the centroid of the points in $S^{(\ell)}$. (If $S^{(\ell)}$ is empty, let $\mathbf{c^{(\ell)}}$ be the zero vector.) Then go to step 2.

It is possible for some of the clusters returned to be empty, particularly if many of the input points are identical.

Lloyd's procedure always terminates. By Theorem 33.1, recomputing the centers of each cluster as the cluster centroid cannot increase $f(S, C)$. Lloyd's procedure ensures that a point is reassigned to a different cluster only when such an operation strictly decreases $f(S, C)$. Thus each iteration of Lloyd's procedure, except the last iteration, must strictly decrease $f(S, C)$. Since there are only a finite number of possible k-clusterings of S (at most k^n), the procedure must terminate. Furthermore, once one iteration of Lloyd's procedure yields no decrease in f, further iterations would not change anything, and the procedure can stop at this locally optimum assignment of points to clusters.

If Lloyd's procedure really required k^n iterations, it would be impractical. In practice, it sometimes suffices to terminate the procedure when the percentage decrease in $f(S, C)$ in the latest iteration falls below a predetermined threshold. Because Lloyd's procedure is guaranteed to find only a locally optimal clustering, one approach to finding a good clustering is to run Lloyd's procedure many times with different randomly chosen initial centers, taking the best result.

The running time of Lloyd's procedure is proportional to the number T of iterations. In one iteration, assigning points to clusters based on the nearest-center rule requires $O(dkn)$ time, and recomputing new centers for each cluster requires $O(dn)$ time (because each point is in one cluster). The overall running time of the k-means procedure is thus $O(Tdkn)$.

Lloyd's algorithm illustrates an approach common to many machine-learning algorithms:

- First, define a hypothesis space in terms an appropriate sequence θ of parameters, so that each θ is associated with a specific hypothesis h_θ. (For the k-means problem, θ is a dk-dimensional vector, equivalent to C, containing the d-dimensional center of each of the k clusters, and h_θ is the hypothesis that each data point $\mathbf{x}$ should be grouped with a cluster having a center closest to $\mathbf{x}$.)
- Second, define a measure $f(E, \theta)$ describing how poorly hypothesis h_θ fits the given training data E. Smaller values of $f(E, \theta)$ are better, and a (locally) optimal solution (locally) minimizes $f(E, \theta)$. (For the k-means problem, $f(E, \theta)$ is just $f(S, C)$.)
- Third, given a set of training data E, use a suitable optimization procedure to find a value of θ^* that minimizes $f(E, \theta^*)$, at least locally. (For the k-means problem, this value of θ^* is the sequence C of k center points returned by Lloyd's algorithm.)
- Return θ^* as the answer.

In this framework, we see that optimization becomes a powerful tool for machine learning. Using optimization in this way is flexible. For example, ***regularization*** terms can be incorporated in the function to be minimized, in order to penalize hypotheses that are "too complicated" and that "overfit" the training data. (Regularization is a complex topic that isn't pursued further here.)

Examples

Figure 33.1 demonstrates Lloyd's procedure on a set of $n = 49$ cities: 48 U.S. state capitals and the District of Columbia. Each city has $d = 2$ dimensions: latitude and longitude. The initial clustering in part (a) of the figure has the initial cluster centers arbitrarily chosen as the capitals of Arkansas, Kansas, Louisiana,

and Tennessee. As the procedure iterates, the value of the function f decreases, until the 11th iteration in part (l), where it remains the same as in the 10th iteration in part (k). Lloyd's procedure then terminates with the clusters shown in part (l).

As Figure 33.2 shows, Lloyd's procedure can also apply to "vector quantization." Here, the goal is to reduce the number of distinct colors required to represent a photograph, thereby allowing the photograph to be greatly compressed (albeit in a lossy manner). In part (a) of the figure, an original photograph 700 pixels wide and 500 pixels high uses 24 bits (three bytes) per pixel to encode a triple of red, green, and blue (RGB) primary color intensities. Parts (b)–(e) of the figure show the results of using Lloyd's procedure to compress the picture from a initial space of 2^{24} possible values per pixel to a space of only $k = 4, k = 16, k = 64$, or $k = 256$ possible values per pixel; these k values are the cluster centers. The photograph can then be represented with only 2, 4, 6, or 8 bits per pixel, respectively, instead of the 24-bits per pixel needed by the initial photograph. An auxiliary table, the "palette," accompanies the compressed image; it holds the k 24-bit cluster centers and is used to map each pixel value to its 24-bit cluster center when the photo is decompressed.

Exercises

33.1-1

Show that the objective function $f(S, C)$ of equation (33.2) may be alternatively written as

$$f(S, C) = \sum_{\ell=1}^{k} \frac{1}{2\,|S^{(\ell)}|} \sum_{\mathbf{x} \in S^{(\ell)}} \sum_{\mathbf{y} \in S^{(\ell)}: \mathbf{x} \neq \mathbf{y}} \Delta(\mathbf{x}, \mathbf{y}) \; .$$

33.1-2

Give an example in the plane with $n = 4$ points and $k = 2$ clusters where an iteration of Lloyd's procedure does not improve $f(S, C)$, yet the k-clustering is not optimal.

33.1-3

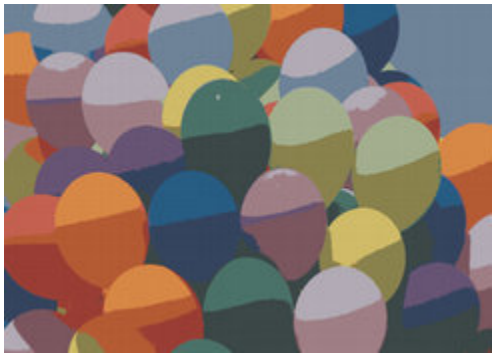
When the input to Lloyd's procedure contains many repeated points, a different initialization procedure might be used. Describe a way to pick a number of centers at random that maximizes the number of distinct centers picked. (*Hint:* See Exercise 5.3-5.)

33.1-4

Show how to find an optimal k-clustering in polynomial time when there is just one attribute ($d = 1$).

(a) Original

(b) $k = 4$ ($f = 1.29 \times 10^9$; 31 iterations)



(c) $k = 16$ ($f = 3.31 \times 10^8$; 36 iterations)

(d) $k = 64$ ($f = 5.50 \times 10^7$; 59 iterations)

(e) $k = 256$ ($f = 1.52 \times 10^7$; 104 iterations)

Figure 33.2 Using Lloyd's procedure for vector quantization to compress a photo by using fewer colors. **(a)** The original photo has 350,000 pixels (700×500), each a 24-bit RGB (red/blue/green) triple of 8-bit values; these pixels (colors) are the "points" to be clustered. Points repeat, so there are only 79,083 distinct colors (less than 2^{24}). After compression, only k distinct colors are used, so each pixel is represented by only $\lceil \lg k \rceil$ bits instead of 24. A "palette" maps these values back to 24-bit RGB values (the cluster centers). **(b)–(e)** The same photo with $k = 4$, 16, 64, and 256 colors. (Photo from standuppaddle, pixabay.com.)

33.2 Multiplicative-weights algorithms

This section considers problems that require you to make a series of decisions. After each decision you receive feedback as to whether your decision was correct. We will study a class of algorithms that are called ***multiplicative-weights algorithms***. This class of algorithms has a wide variety of applications, including game playing in economics, approximately solving linear-programming and multicommodity-flow problems, and various applications in online machine learning. We emphasize the online nature of the problem here: you have to make a sequence of decisions, but some of the information needed to make the ith decision appears only after you have already made the $(i-1)$st decision. In this section, we look at one particular problem, known as "learning from experts," and develop an example of a multiplicative-weights algorithm, called the weighted-majority algorithm.

Suppose that a series of events will occur, and you want to make predictions about these events. For example, over a series of days, you want to predict whether it is going to rain. Or perhaps you want to predict whether the price of a stock will increase or decrease. One way to approach this problem is to assemble a group of "experts" and use their collective wisdom in order to make good predictions. Let's denote the experts, n of them, by $E_1, E_2, \ldots, E_n$, and let's say that T events are going to take place. Each event has an outcome of either 0 or 1, with $o^{(t)}$ denoting the outcome of the tth event. Before event t, each expert $E^{(i)}$ makes a prediction $q_i^{(t)} \in \{0,1\}$. You, as the "learner," then take the set of n expert predictions for event t and produce a single prediction $p^{(t)} \in \{0,1\}$ of your own. You base your prediction only on the predictions of the experts and anything you have learned about the experts from their previous predictions. You do not use any additional information about the event. Only after making your prediction do you ascertain the outcome $o^{(t)}$ of event t. If your prediction $p^{(t)}$ matches $o^{(t)}$, then you were correct; otherwise, you made a mistake. The goal is to minimize the total number m of mistakes, where $m = \sum_{t=1}^{T} \left|p^{(t)} - o^{(t)}\right|$. You can also keep track of the number of mistakes each expert makes: expert E_i makes m_i mistakes, where $m_i = \sum_{t=1}^{T} \left|q_i^{(t)} - o^{(t)}\right|$.

For example, suppose that you are following the price of a stock, and each day you decide whether to invest in it for just that day by buying it at the beginning of the day and selling it at the end of the day. If, on some day, you buy the stock and it goes up, then you made the correct decision, but if the stock goes down, then you made a mistake. Similarly, if on some day, you do not buy the stock and it goes down, then you made the correct decision, but if the stock goes up, then you made a mistake. Since you would like to make as few mistakes as possible, you use the advice of the experts to make your decisions.

We'll assume nothing about the movement of the stock. We'll also assume nothing about the experts: the experts' predictions could be correlated, they could be chosen to deceive you, or perhaps some are not really experts after all. What algorithm would you use?

Before designing an algorithm for this problem, we need to consider what is a fair way to evaluate our algorithm. It is reasonable to expect that our algorithm performs better when the expert predictions are better, and that it performs worse when the expert predictions are worse. The goal of the algorithm is to limit the number of mistakes you make to be close to the number of mistakes that the best of the experts makes. At first, this goal might seem impossible, because you do not know until the end which expert is best. We'll see, however, that by taking the advice provided by all the experts into account, you can achieve this goal. More formally, we use the notion of "regret," which compares our algorithm to the performance of the best expert (in hindsight) over all. Letting $m^* = \min\{m_i : 1 \le i \le n\}$ denote the number of mistakes made by the best expert, the ***regret*** is $m - m^*$. The goal is to design an algorithm with low regret. (Regret can be negative, although it typically isn't, since it is rare that you do better than the best expert.)

As a warm-up, let's consider the case in which one of the experts makes a correct prediction each time. Even without knowing who that expert is, you can still achieve good results.

Lemma 33.3

Suppose that out of n experts, there is one who always makes the correct prediction for all T events. Then there is an algorithm that makes at most $\lceil \lg n \rceil$ mistakes.

Proof The algorithm maintains a set S consisting of experts who have not yet made a mistake. Initially, S contains all n experts. The algorithm's prediction is always the majority vote of the predictions of the experts remaining in set S. In case of a tie, the algorithm makes any prediction. After each outcome is learned, set S is updated to remove all the experts who made an incorrect prediction about that outcome.

We now analyze the algorithm. The expert who always makes the correct prediction will always be in set S. Every time the algorithm makes a mistake, at least half of the experts who were still in S also make a mistake, and these experts are removed from S. If S' is the set of experts remaining after removing those who made a mistake, we have that $|S'| \le |S|/2$. The size of S can be halved at most $\lceil \lg n \rceil$ times until $|S| = 1$. From this point on, we know that the algorithm never makes a mistake, since the set S consists only of the one expert who never makes a mistake. Therefore, overall the algorithm makes at most $\lceil \lg n \rceil$ mistakes. ■

Exercise 33.2-1 asks you to generalize this result to the case when there is no expert who makes perfect predictions and show that, for any set of experts, there is an algorithm that makes at most $m^* \lceil \lg n \rceil$ mistakes. The generalized algorithm begins in the same way. The set S might become empty at some point, however. If that ever happens, reset S to contain all the experts and continue the algorithm.

You can substantially improve your prediction ability by not just tracking which experts have not made any mistakes, or have not made any mistakes recently, to a more nuanced evaluation of the quality of each expert. The key idea is to use the feedback you receive to update your evaluation of how much trust to put in each expert. As the experts make predictions, you observe whether they were correct and decrease your confidence in the experts who make more mistakes. In this way, you can learn over time which experts are more reliable and which are less reliable, and weight their predictions accordlingly. The change in weights is accomplished via multiplication, hence the term "multiplicative weights."

The algorithm appears in the procedure WEIGHTED-MAJORITY on the following page, which takes a set $E = \{E_1, E_2, \ldots, E_n\}$ of experts, a number T of events, the number n of experts, and a parameter $0 < \gamma \leq 1/2$ that controls how the weights change. The algorithm maintains weights $w_i^{(t)}$ for $i = 1, 2, \ldots, n$ and $t = 1, 2, \ldots, T$, where $0 < w_i^{(t)} \leq 1$. The **for** loop of lines 1–2 sets the initial weights $w_i^{(1)}$ to 1, capturing the idea that with no knowledge, you trust each expert equally. Each iteration of the main **for** loop of lines 3–18 does the following for an event $t = 1, 2, \ldots, T$. Each expert E_i makes a prediction for event t in line 4. Lines 5–8 compute $upweight^{(t)}$, the sum of the weights of the experts who predict 1 for event t, and $downweight^{(t)}$, the sum of the weights of the experts who predict 0 for the event. Lines 9–11 decide the algorithm's prediction $p^{(t)}$ for event t based on whichever weighted sum is larger (breaking ties in favor of deciding 1). The outcome of event t is revealed in line 12. Finally, lines 14–17 decrease the weights of the experts who made an incorrect prediction for event t by multiplying their weights by $1 - \gamma$, leaving alone the weights of the experts who correctly predicted the event's outcome. Thus, the fewer mistakes each expert makes, the higher that expert's weight.

The WEIGHTED-MAJORITY procedure doesn't do much worse than any expert. In particular, it doesn't do much worse than the best expert. To quantify this claim, let $m^{(t)}$ be the number of mistakes made by the procedure through event t, and let $m_i^{(t)}$ be the number of mistakes made by expert E_i through event t. The following theorem is the key.

```
WEIGHTED-MAJORITY(E, T, n, γ)
 1  for i = 1 to n
 2      w_i^(1) = 1                          // trust each expert equally
 3  for t = 1 to T
 4      each expert E_i ∈ E makes a prediction q_i^(t)
 5      U = {E_i : q_i^(t) = 1}              // experts who predicted 1
 6      upweight^(t) = Σ_{i:E_i∈U} w_i^(t)   // sum of weights of who predicted 1
 7      D = {E_i : q_i^(t) = 0}              // experts who predicted 0
 8      downweight^(t) = Σ_{i:E_i∈D} w_i^(t) // sum of weights of who predicted 0
 9      if upweight^(t) ≥ downweight^(t)
10          p^(t) = 1                        // algorithm predicts 1
11      else p^(t) = 0                       // algorithm predicts 0
12      outcome o^(t) is revealed
13      // If p^(t) ≠ o^(t), the algorithm made a mistake.
14      for i = 1 to n
15          if q_i^(t) ≠ o^(t)               // if expert E^(i) made a mistake ...
16              w_i^(t+1) = (1 − γ) w_i^(t)  // ... then decrease that expert's weight
17          else w_i^(t+1) = w_i^(t)
18      return p^(t)
```

Theorem 33.4

When running WEIGHTED-MAJORITY, we have, for every expert E_i and every event $T' \le T$,

$$m^{(T')} \le 2(1+\gamma)m_i^{(T')} + \frac{2\ln n}{\gamma} . \tag{33.5}$$

Proof Every time an expert E_i makes a mistake, its weight, which is initially 1, is multiplied by $1-\gamma$, and so we have

$$w_i^{(t)} = (1-\gamma)^{m_i^{(t)}} \tag{33.6}$$

for $t = 1, 2, \ldots, T$.

We use a potential function $W(t) = \sum_{i=1}^{n} w_i^{(t)}$, summing the weights for all n experts after iteration t of the **for** loop of lines 3–18. Initially, we have $W(0) = n$ since all n weights start out with the value 1. Because each expert belongs to either the set U or the set D (defined in lines 5 and 7 of WEIGHTED-MAJORITY), we always have $W(t) = upweight^{(t)} + downweight^{(t)}$ after each execution of line 8.

Consider an iteration t in which the algorithm makes a mistake in its prediction, which means that either the algorithm predicts 1 and the outcome is 0 or the al-

gorithm predicts 0 and the outcome is 1. Without loss of generality, assume that the algorithm predicts 1 and the outcome is 0. The algorithm predicted 1 because $upweight^{(t)} \geq downweight^{(t)}$ in line 9, which implies that

$$upweight^{(t)} \geq W(t)/2 \,. \tag{33.7}$$

Each expert in U then has its weight multiplied by $1-\gamma$, and each expert in D has its weight unchanged. Thus, we have

$$\begin{aligned} W(t+1) &= upweight^{(t)}(1-\gamma) + downweight^{(t)} \\ &= upweight^{(t)} + downweight^{(t)} - \gamma \cdot upweight^{(t)} \\ &= W(t) - \gamma \cdot upweight^{(t)} \\ &\leq W(t) - \gamma \frac{W(t)}{2} \qquad \text{(by inequality (33.7))} \\ &= W(t)(1-\gamma/2) . \end{aligned}$$

Therefore, for every iteration t in which the algorithm makes a mistake, we have

$$W(t+1) \leq (1-\gamma/2)W(t) . \tag{33.8}$$

In an iteration where the algorithm does not make a mistake, some of the weights decrease and some remain unchanged, so that we have

$$W(t+1) \leq W(t) \,. \tag{33.9}$$

Since there are $m^{(T')}$ mistakes made through iteration T', and $W(1) = n$, we can repeatedly apply inequality (33.8) to iterations where the algorithm makes a mistake and inequality (33.9) to iterations where the algorithm does not make a mistake, obtaining

$$W(T') \leq n(1-\gamma/2)^{m^{(T')}} \,. \tag{33.10}$$

Because the function W is the sum of the weights and all weights are positive, its value exceeds any single weight. Therefore, using equation (33.6) we have, for any expert E_i and for any iteration $T' \leq T$,

$$W(T') \geq w_i^{(T')} = (1-\gamma)^{m_i^{(T')}} \,. \tag{33.11}$$

Combining inequalities (33.10) and (33.11) gives

$$(1-\gamma)^{m_i^{(T')}} \leq n(1-\gamma/2)^{m^{(T')}} \,.$$

Taking the natural logarithm of both sides yields

$$m_i^{(T')} \ln(1-\gamma) \leq m^{(T')} \ln(1-\gamma/2) + \ln n \,. \tag{33.12}$$

We now use the Taylor series expansion to derive upper and lower bounds on the logarithmic factors in inequality (33.12). The Taylor series for $\ln(1+x)$ is given in equation (3.22) on page 67. Substituting $-x$ for x, we have that for $0 < x \le 1/2$,

$$\ln(1-x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \cdots . \tag{33.13}$$

Since each term on the right-hand side is negative, we can drop all terms except the first and obtain an upper bound of $\ln(1-x) \le -x$. Since $0 < \gamma \le 1/2$, we have

$$\ln(1-\gamma/2) \le -\gamma/2 \tag{33.14}$$

For the lower bound, Exercise 33.2-2 asks you to show that $\ln(1-x) \ge -x - x^2$ when $0 < x \le 1/2$, so that

$$-\gamma - \gamma^2 \le \ln(1-\gamma) . \tag{33.15}$$

Thus, we have

$$\begin{aligned} m_i^{(T')}(-\gamma-\gamma^2) &\le m_i^{(T')} \ln(1-\gamma) && \text{(by inequality (33.15))} \\ &\le m^{(T')} \ln(1-\gamma/2) + \ln n && \text{(by inequality (33.12)} \\ &\le m^{(T')}(-\gamma/2) + \ln n && \text{(by inequality (33.14))} , \end{aligned}$$

so that

$$m_i^{(T')}(-\gamma-\gamma^2) \le m^{(T')}(-\gamma/2) + \ln n . \tag{33.16}$$

Subtracting $\ln n$ from both sides of inequality (33.16) and then multiplying both sides by $-2/\gamma$ yields $m^{(T')} \le 2(1+\gamma)m_i^{(T')} + (2\ln n)/\gamma$, thus proving the theorem. ■

Theorem 33.4 applies to any expert and any event $T' \le T$. In particular, we can compare against the best expert after all events have occurred, producing the following corollary.

Corollary 33.5
At the end of procedure WEIGHTED-MAJORITY, we have

$$m^{(T)} \le 2(1+\gamma)m^* + \frac{2\ln n}{\gamma} . \tag{33.17}$$

■

Let's explore this bound. Assuming that $\sqrt{\ln n/m^*} \le 1/2$, we can choose $\gamma = \sqrt{\ln n/m^*}$ and plug into inequality (33.17) to obtain

$$\begin{aligned} m^{(T)} &\le 2\left(1+\sqrt{\frac{\ln n}{m^*}}\right)m^* + \frac{2\ln n}{\sqrt{\ln n/m^*}} \\ &= 2m^* + 2\sqrt{m^*\ln n} + 2\sqrt{m^*\ln n} \\ &= 2m^* + 4\sqrt{m^*\ln n}\ , \end{aligned}$$

and so the number of errors is at most twice the number of errors made by the best expert plus a term that is often slower growing than m^*. Exercise 33.2-4 shows that you can decrease the bound on the number of errors by a factor of 2 by using randomization, which leads to much stronger bounds. In particular, the upper bound on regret $(m - m^*)$ is reduced from $(1+2\gamma)m^* + (2\ln n)/\gamma$ to an expected value of $\epsilon m^* + (\ln n)/\epsilon$, where both γ and ϵ are at most $1/2$. Numerically, we can see that if $\gamma = 1/2$, WEIGHTED-MAJORITY makes at most 3 times the number of errors as the best expert, plus $4\ln n$ errors. As another example, suppose that $T = 1000$ predictions are being made by $n = 20$ experts, and the best expert is correct 95% of the time, making 50 errors. Then WEIGHTED-MAJORITY makes at most $100(1+\gamma)+2\ln 20/\gamma$ errors. By choosing $\gamma = 1/4$, WEIGHTED-MAJORITY makes at most 149 errors, or a success rate of at least 85%.

Multiplicative weights methods typically refer to a broader class of algorithms that includes WEIGHTED-MAJORITY. The outcomes and predictions need not be only 0 or 1, but can be real numbers, and there can be a loss associated with a particular outcome and prediction. The weights can be updated by a multiplicative factor that depends on the loss, and the algorithm can, given a set of weights, treat them as a distribution on experts and use them to choose an expert to follow in each event. Even in these more general settings, bounds similar to Theorem 33.4 hold.

Exercises

33.2-1

The proof of Lemma 33.3 assumes that some expert never makes a mistake. It is possible to generalize the algorithm and analysis to remove this assumption. The new algorithm begins in the same way. The set S might become empty at some point, however. If that ever happens, reset S to contain all the experts and continue the algorithm. Show that the number of mistakes that this algorithm makes is at most $m^* \lceil \lg n \rceil$.

33.2-2

Show that $\ln(1-x) \ge -x - x^2$ when $0 < x \le 1/2$. (*Hint:* Start with equation (33.13), group all the terms after the first three, and use equation (A.7) on page 1142.)

33.2-3

Consider a randomized variant of the algorithm given in the proof of Lemma 33.3, in which some expert never makes a mistake. At each step, choose an expert E_i uniformly at random from the set S and then make the same predication as E_i. Show that the expected number of mistakes made by this algorithm is $\lceil \lg n \rceil$.

33.2-4

Consider a randomized version of WEIGHTED-MAJORITY. The algorithm is the same, except for the prediction step, which interprets the weights as a probability distribution over the experts and chooses an expert E_i according to that distribution. It then chooses its prediction to be the same as the prediction made by expert E_i. Show that, for any $0 < \epsilon < 1/2$, the expected number of mistakes made by this algorithm is at most $(1 + \epsilon)m^* + (\ln n)/\epsilon$.

33.3 Gradient descent

Suppose that you have a set $\{p_1, p_2, \ldots, p_n\}$ of points and you want to find the line that best fits these points. For any line ℓ, there is a distance d_i between each point p_i and the line. You want to find the line that minimizes some function $f(d_1, \ldots, d_n)$. There are many possible choices for the definition of distance and for the function f. For example, the distance can be the projection distance to the line and the function can be the sum of the squares of the distances. This type of problem is common in data science and machine learning—the line is the hypothesis that best describes the data—where the particular definition of best is determined by the definition of distance and the objective f. If the definition of distance and the function f are linear, then we have a linear-programming problem, as discussed in Chapter 29. Although the linear-programming framework captures several important problems, many other problems, including various machine-learning problems, have objectives and constraints that are not necessarily linear. We need frameworks and algorithms to solve such problems.

In this section, we consider the problem of optimizing a continuous function and discuss one of the most popular methods to do so: gradient descent. Gradient descent is a general method for finding a local minimum of a function $f : \mathbb{R}^n \to \mathbb{R}$, where informally, a local minimum of a function f is a point $\mathbf{x}$ for which $f(\mathbf{x}) \le f(\mathbf{x}')$ for all $\mathbf{x}'$ that are "near" $\mathbf{x}$. When the function is convex, it can find a point near the ***global minimizer*** of f: an n-vector argument $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ such that $f(\mathbf{x})$ is minimum. For the intuitive idea behind gradient descent, imagine being in a landscape of hills and valleys, and wanting to get to a low point as quickly as possible. You survey the terrain and choose to

move in the direction that takes you downhill the fastest from your current position. You move in that direction, but only for a short while, because as you proceed, the terrain changes and you might need to choose a different direction. So you stop, reevaluate the possible directions and move another short distance in the steepest downhill direction, which might differ from the direction of your previous movement. You continue this process until you reach a point from which all directions lead up. Such a point is a local minimum.

In order to make this informal procedure more formal, we need to define the gradient of a function, which in the analogy above is a measure of the steepness of the various directions. Given a function $f : \mathbb{R}^n \to \mathbb{R}$, its ***gradient*** ∇f is a function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ comprising n partial derivatives: $(\nabla f)(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$. Analogous to the derivative of a function of a single variable, the gradient can be viewed as a direction in which the function value locally increases the fastest, and the rate of that increase. This view is informal; in order to make it formal we would have to define what local means and place certain conditions, such as continuity or existence of derivatives, on the function. Nevertheless, this view motivates the key step of gradient descent—move in the direction opposite to the gradient, by a distance influenced by the magnitude of the gradient.

The general procedure of gradient descent proceeds in steps. You start at some initial point $\mathbf{x^{(0)}}$, which is an n-vector. At each step t, you compute the value of the gradient of f at point $\mathbf{x^{(t)}}$, that is, $(\nabla f)(\mathbf{x^{(t)}})$, which is also an n-vector. You then move in the direction opposite to the gradient in each dimension at $\mathbf{x^{(t)}}$ to arrive at the next point $\mathbf{x^{(t+1)}}$, which again is an n-vector. Because you moved in a monotonically decreasing direction in each dimension, you should have that $f(\mathbf{x^{(t+1)}}) \leq f(\mathbf{x^{(t)}})$. Several details are needed to turn this idea into an actual algorithm. The two main details are that you need an initial point and that you need to decide how far to move in the direction of the negative gradient. You also need to understand when to stop and what you can conclude about the quality of the solution found. We will explore these issues further in this section, for both constrained minimization, where there are additional constraints on the points, and unconstrained minimization, where there are none.

Unconstrained gradient descent

In order to gain intuition, let's consider unconstrained gradient descent in just one dimension, that is, when f is a function of a scalar x, so that $f : \mathbb{R} \to \mathbb{R}$. In this case, the gradient ∇f of f is just $f'(x)$, the derivative of f with respect to x. Consider the function f shown in blue in Figure 33.3, with minimizer x^* and starting point $x^{(0)}$. The gradient (derivative) $f'(x^{(0)})$, shown in orange, has a negative slope, so that a small step from $x^{(0)}$ in the direction of increasing x results in a point x' for which $f(x') < f(x^{(0)})$. Too large a step, however, results in a

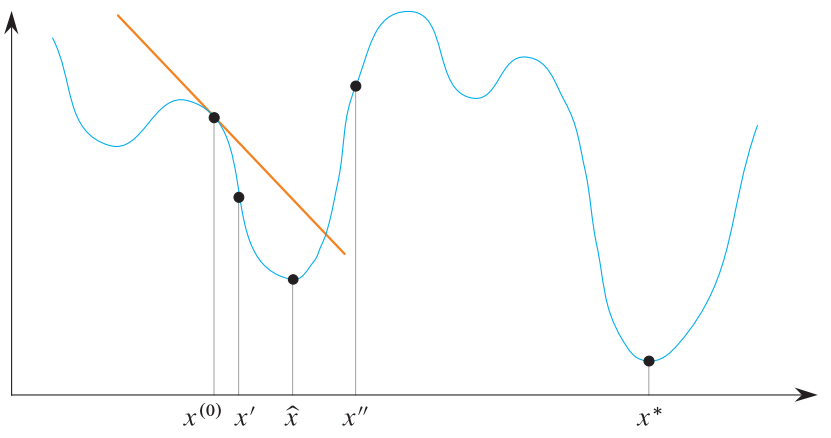


Figure 33.3 A function $f : \mathbb{R} \to \mathbb{R}$, shown in blue. Its gradient at point $x^{(0)}$, in orange, has a negative slope, and so a small increase in x from $x^{(0)}$ to x' results in $f(x') < f(x^{(0)})$. Small increases in x from $x^{(0)}$ head toward $\hat{x}$, which gives a local minimum. Too large an increase in x can end up at x'', where $f(x'') > f(x^{(0)})$. Small steps starting from $x^{(0)}$ and going only in the direction of decreasing values of f cannot end up at the global minimizer x^*.

point x'' for which $f(x'') > f(x^{(0)})$, so this is a bad idea. Restricting ourselves to small steps, where each one has $f(x') < f(x)$, eventually results in getting close to point $\hat{x}$, which gives a local minimum. By taking only small downhill steps, however, gradient descent has no chance to get to the global minimizer x^*, given the starting point $x^{(0)}$.

We draw two observations from this simple example. First, gradient descent converges toward a local minimum, and not necessarily a global minimum. Second, the speed at which it converges and how it behaves are related to properties of the function, to the initial point, and to the step size of the algorithm.

The procedure GRADIENT-DESCENT on the facing page takes as input a function f, an initial point $\mathbf{x^{(0)}} \in \mathbb{R}^n$, a fixed step-size multiplier $\gamma > 0$, and a number $T > 0$ of steps to take. Each iteration of the **for** loop of lines 2–4 performs a step by computing the n-dimensional gradient at point $\mathbf{x}^{(t)}$ and then moving distance γ in the opposite direction in the n-dimensional space. The complexity of computing the gradient depends on the function f and can sometimes be expensive. Line 3 sums the points visited. After the loop terminates, line 6 returns **x-avg**, the average of all the points visited except for the last one, $\mathbf{x}^{(T)}$. It might seem more natural to return $\mathbf{x}^{(T)}$, and in fact, in many circumstances, you might prefer to have the function return $\mathbf{x}^{(T)}$. For the version we will analyze, however, we use **x-avg**.

```
GRADIENT-DESCENT(f, x^(0), γ, T)
1  sum = 0                                  // n-dimensional vector, initially all 0
2  for t = 0 to T − 1
3      sum = sum + x^(t)                    // add each of n dimensions into sum
4      x^(t+1) = x^(t) − γ · (∇f)(x^(t))    // (∇f)(x^(t)), x^(t+1) are n-dimensional
5  x-avg = sum/T                            // divide each of n dimensions by T
6  return x-avg
```

Figure 33.4 depicts how gradient descent ideally runs on a convex 1-dimensional function.[1] We'll define convexity more formally below, but the figure shows that each iteration moves in the direction opposite to the gradient, with the distance moved being proportional to the magnitude of the gradient. As the iterations proceed, the magnitude of the gradient decreases, and thus the distance moved along the horizontal axis decreases. After each iteration, the distance to the optimal point $\mathbf{x}^*$ decreases. This ideal behavior is not guaranteed to occur in general, but the analysis in the remainder of this section formalizes when this behavior occurs and quantifies the number of iterations needed. Gradient descent does not always work, however. We have already seen that if the function is not convex, gradient descent can converge to a local, rather than global, minimum. We have also seen that if the step size is too large, GRADIENT-DESCENT can overshoot the minimum and wind up farther away. (It is also possible to overshoot the minimum and wind up closer to the optimum.)

Analysis of unconstrained gradient descent for convex functions

Our analysis of gradient descent focuses on convex functions. Inequality (C.29) on page 1194 defines a convex function of one variable, as shown in Figure 33.5. We can extend that definition to a function $f : \mathbb{R}^n \to \mathbb{R}$ and say that f is ***convex*** if for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ and for all $0 \le \lambda \le 1$, we have

$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) . \tag{33.18}$$

(Inequalities (33.18) and (C.29) are the same, except for the dimensions of $\mathbf{x}$ and $\mathbf{y}$.) We also assume that our convex functions are closed[2] and differentiable.

---

[1] Although the curve in Figure 33.4 looks concave, according to the definition of convexity that we'll see below, the function f in the figure is convex.

[2] A function $f : \mathbb{R}^n \to \mathbb{R}$ is closed if, for each $\alpha \in \mathbb{R}$, the set $\{\mathbf{x} \in \mathrm{dom}(f) : f(\mathbf{x}) \le \alpha\}$ is closed, where $\mathrm{dom}(f)$ is the domain of f.

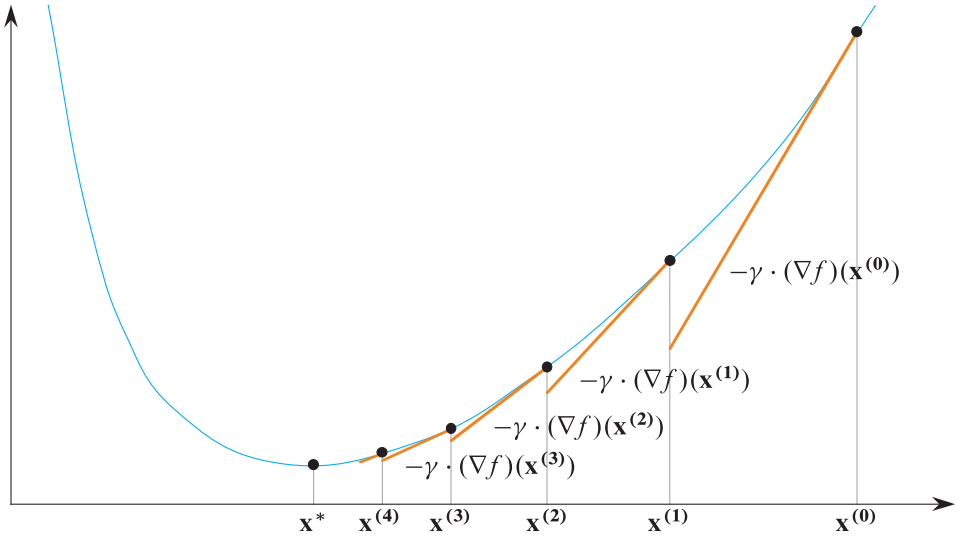


Figure 33.4 An example of running gradient descent on a convex function $f : \mathbb{R} \to \mathbb{R}$, shown in blue. Beginning at point $\mathbf{x}^{(0)}$, each iteration moves in the direction opposite to the gradient, and the distance moved is proportional to the magnitude of the gradient. Orange lines represent the negative of the gradient at each point, scaled by the step size γ. As the iterations proceed, the magnitude of the gradient decreases, and the distance moved decreases correspondingly. After each iteration, the distance to the optimal point $\mathbf{x}^*$ decreases.

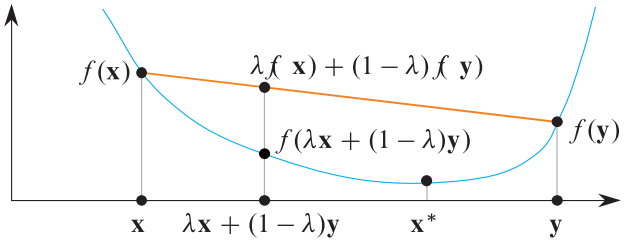


Figure 33.5 A convex function $f : \mathbb{R} \to \mathbb{R}$, shown in blue, with local and global minimizer $\mathbf{x}^*$. Because f is convex, $f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$ for any two values $\mathbf{x}$ and $\mathbf{y}$ and all $0 \le \lambda \le 1$, shown for a particular value of λ. Here, the orange line segment represents all values $\lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y})$ for $0 \le \lambda \le 1$, and it is above the blue line.

A convex function has the property that any local minimum is also a global minimum. To verify this property, consider inequality (33.18), and suppose for the purpose of contradiction that $\mathbf{x}$ is a local minimum but not a global minimum and $\mathbf{y} \neq \mathbf{x}$ is a global minimum, so $f(\mathbf{y}) < f(\mathbf{x})$. Then we have

$$\begin{aligned} f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) &\le \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}) \quad \text{(by inequality (33.18))} \\ &< \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{x}) \\ &= f(\mathbf{x}) \,. \end{aligned}$$

Thus, letting λ approach 1, we see that there is another point near $\mathbf{x}$, say $\mathbf{x}'$, such that $f(\mathbf{x}') < f(\mathbf{x})$, so $\mathbf{x}$ is not a local minimum.

Convex functions have several useful properties. The first property, whose proof we leave as Exercise 33.3-1, says that a convex function always lies above its tangent hyperplane. In the context of gradient descent, angle brackets denote the notation for inner product defined on page 1219 rather than denoting a sequence.

Lemma 33.6

For any convex differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ and for all $x, y \in \mathbb{R}^n$, we have $f(\mathbf{x}) \le f(\mathbf{y}) + \langle (\nabla f)(\mathbf{x}), \mathbf{x} - \mathbf{y} \rangle$. ■

The second property, which Exercise 33.3-2 asks you to prove, is a repeated application of the definition of convexity in inequality (33.18).

Lemma 33.7

For any convex function $f : \mathbb{R}^n \to \mathbb{R}$, for any integer $T \ge 1$, and for all $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(T-1)} \in \mathbb{R}^n$, we have

$$f\left(\frac{\mathbf{x}^{(0)} + \cdots + \mathbf{x}^{(T-1)}}{T}\right) \le \frac{f(\mathbf{x}^{(0)}) + \cdots + f(\mathbf{x}^{(T-1)})}{T} . \tag{33.19}$$

■

The left-hand side of inequality (33.19) is the value of f at the vector **x-avg** that GRADIENT-DESCENT returns.

We now proceed to analyze GRADIENT-DESCENT. It might not return the exact global minimizer $\mathbf{x}^*$. We use an error bound ϵ, and we want to choose T so that $f(\mathbf{x}\text{-}\mathbf{avg}) - f(\mathbf{x}^*) \le \epsilon$ at termination. The value of ϵ depends on the number T of iterations and two additional values. First, since you expect it to be better to start close to the global minimizer, ϵ is a function of

$$R = \|\mathbf{x}^{(0)} - \mathbf{x}^*\| , \tag{33.20}$$

the euclidean norm (or distance, defined on page 1219) of the difference between $\mathbf{x}^{(0)}$ and $\mathbf{x}^*$. The error bound ϵ is also a function of a quantity we call L, which is an upper bound on the magnitude $\|(\nabla f)(\mathbf{x})\|$ of the gradient, so that

$$\|(\nabla f)(\mathbf{x})\| \le L , \tag{33.21}$$

where $\mathbf{x}$ ranges over all the points $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(T-1)}$ whose gradients are computed by GRADIENT-DESCENT. Of course, we don't know the values of L and R, but for now let's assume that we do. We'll discuss later how to remove these assumptions. The analysis of GRADIENT-DESCENT is summarized in the following theorem.

Theorem 33.8

Let $\mathbf{x}^* \in \mathbb{R}^n$ be the minimizer of a convex function f, and suppose that an execution of GRADIENT-DESCENT$(f, \mathbf{x^{(0)}}, \gamma, T)$ returns **x-avg**, where $\gamma = R/(L\sqrt{T})$ and R and L are defined in equations (33.20) and (33.21). Let $\epsilon = RL/\sqrt{T}$. Then we have $f(\textbf{x-avg}) - f(\mathbf{x}^*) \le \epsilon$. ■

We now prove this theorem. We do not give an absolute bound on how much progress each iteration makes. Instead, we use a potential function, as in Section 16.3. Here, we define a potential $\Phi(t)$ after computing $\mathbf{x^{(t)}}$, such that $\Phi(t) \ge 0$ for $t = 0, \ldots, T$. We define the ***amortized progress*** in the iteration that computes $\mathbf{x^{(t)}}$ as

$$p(t) = f(\mathbf{x^{(t)}}) - f(\mathbf{x}^*) + \Phi(t+1) - \Phi(t) \; . \tag{33.22}$$

Along with including the change in potential ($\Phi(t+1) - \Phi(t)$), equation (33.22) also subtracts the minimum value $f(\mathbf{x}^*)$ because ultimately, you care not about the values $f(\mathbf{x^{(t)}})$ but about how close they are to $f(\mathbf{x}^*)$. Suppose that we can show that $p(t) \le B$ for some value B and $t = 0, \ldots, T-1$. Then we can substitute for $p(t)$ using equation (33.22), giving

$$f(\mathbf{x^{(t)}}) - f(\mathbf{x}^*) \le B - \Phi(t+1) + \Phi(t) \; . \tag{33.23}$$

Summing inequality (33.23) over $t = 0, \ldots, T-1$ yields

$$\sum_{t=0}^{T-1} (f(\mathbf{x^{(t)}}) - f(\mathbf{x}^*)) \le \sum_{t=0}^{T-1} (B - \Phi(t+1) + \Phi(t)) \; .$$

Observing that we have a telescoping series on the right and regrouping terms, we have that

$$\left(\sum_{t=0}^{T-1} f(\mathbf{x^{(t)}}) \right) - T \cdot f(\mathbf{x}^*) \le TB - \Phi(T) + \Phi(0) \; .$$

Dividing by T and dropping the positive term $\Phi(T)$ gives

$$\frac{\sum_{t=0}^{T-1} f(\mathbf{x^{(t)}})}{T} - f(\mathbf{x}^*) \le B + \frac{\Phi(0)}{T} \; , \tag{33.24}$$

and thus we have

$$\begin{aligned} f(\textbf{x-avg}) - f(\mathbf{x}^*) &= f\left(\frac{\sum_{t=0}^{T-1} \mathbf{x^{(t)}}}{T} \right) - f(\mathbf{x}^*) && \text{(by the definition of \textbf{x-avg})} \\ &\le \frac{\sum_{t=0}^{T-1} f(\mathbf{x^{(t)}})}{T} - f(\mathbf{x}^*) && \text{(by Lemma 33.7)} \\ &\le B + \frac{\Phi(0)}{T} && \text{(by inequality (33.24))} \; . \end{aligned} \tag{33.25}$$

In other words, if we can show that $p(t) \le B$ for some value B and choose a potential function where $\Phi(0)$ is not too large, then inequality (33.25) tells us how close the function value $f(\mathbf{x\text{-}avg})$ is to the function value $f(\mathbf{x}^*)$ after T iterations. That is, we can set the error bound ϵ to $B + \Phi(0)/T$.

In order to bound the amortized progress, we need to come up with a concrete potential function. Define the potential function $\Phi(t)$ by

$$\Phi(t) = \frac{\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2}{2\gamma} , \tag{33.26}$$

that is, the potential function is proportional to the square of the distance between the current point and the minimizer $\mathbf{x}^*$. With this potential function in hand, the next lemma provides a bound on the amortized progress made in any iteration of GRADIENT-DESCENT.

Lemma 33.9
Let $\mathbf{x}^* \in \mathbb{R}^n$ be the minimizer of a convex function f, and consider an execution of GRADIENT-DESCENT$(f, \mathbf{x}^{(0)}, \gamma, T)$. Then for each point $\mathbf{x}^{(t)}$ computed by the procedure, we have that

$$p(t) = f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) + \Phi(t+1) - \Phi(t) \le \frac{\gamma L^2}{2} .$$

Proof We first bound the potential change $\Phi(t+1) - \Phi(t)$. Using the definition of $\Phi(t)$ from equation (33.26), we have

$$\Phi(t+1) - \Phi(t) = \frac{1}{2\gamma}\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 . \tag{33.27}$$

From line 4 in GRADIENT-DESCENT, we know that

$$\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)} = -\gamma \cdot (\nabla f)(\mathbf{x}^{(t)}) , \tag{33.28}$$

and so we would like to rewrite equation (33.27) to have $\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$ terms. As Exercise 33.3-3 asks you to prove, for any two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, we have

$$\|\mathbf{a} + \mathbf{b}\|^2 - \|\mathbf{a}\|^2 = 2\langle \mathbf{b}, \mathbf{a}\rangle + \|\mathbf{b}\|^2 . \tag{33.29}$$

Letting $\mathbf{a} = \mathbf{x}^{(t)} - \mathbf{x}^*$ and $\mathbf{b} = \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}$, we can write the right-hand side of equation (33.27) as $\frac{1}{2\gamma}\left(\|\mathbf{a} + \mathbf{b}\|^2 - \|\mathbf{a}\|^2\right)$. Then we can express the potential change as

$$
\begin{aligned}
&\Phi(t+1) - \Phi(t) \\
&= \frac{1}{2\gamma}\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 && \text{(by equation (33.27))} \\
&= \frac{1}{2\gamma}\left(2\langle \mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}, \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|\mathbf{x}^{(t+1)} - \mathbf{x}^{(t)}\right\|^2\right) && \text{(by equation (33.29))} \\
&= \frac{1}{2\gamma}\left(2\langle -\gamma \cdot (\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|-\gamma \cdot (\nabla f)(\mathbf{x}^{(t)})\right\|^2\right) \\
& && \text{(by equation (33.28))} \\
&= -\langle (\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 && (33.30) \\
& && \text{(by equation (D.3) on page 1219)} \\
&\le -(f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 && \text{(by Lemma 33.6) ,}
\end{aligned}
$$

and thus we have

$$
\Phi(t+1) - \Phi(t) \le -(f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 \qquad (33.31)
$$

We can now proceed to bound $p(t)$. By the bound on the potential change from inequality (33.31), and using the definition of L (inequality (33.21)), we have

$$
\begin{aligned}
p(t) &= f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) + \Phi(t+1) - \Phi(t) && \text{(by equation (33.22))} \\
&\le f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*) - (f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)) + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 \\
&= \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 && \text{(by inequality (33.31))} \\
&\le \frac{\gamma L^2}{2} && \text{(by inequality (33.21)) .} \quad \blacksquare
\end{aligned}
$$

Having bounded the amortized progress in one step, we now analyze the entire GRADIENT-DESCENT procedure, completing the proof of Theorem 33.8.

Proof of Theorem 33.8 Inequality (33.25) tells us that if we have an upper bound of B for $p(t)$, then we also have the bound $f(\mathbf{x\text{-}avg}) - f(\mathbf{x}^*) \le B + \Phi(0)/T$. By equations (33.20) and (33.26), we have that $\Phi(0) = R^2/(2\gamma)$. Lemma 33.9 gives us the upper bound of $B = \gamma L^2/2$, and so we have

$$
\begin{aligned}
f(\mathbf{x\text{-}avg}) - f(\mathbf{x}^*) &\le B + \frac{\Phi(0)}{T} && \text{(by inequality (33.25))} \\
&= \frac{\gamma L^2}{2} + \frac{R^2}{2\gamma T} .
\end{aligned}
$$

Our choice of $\gamma = R/(L\sqrt{T})$ in the statement of Theorem 33.8 balances the two terms, and we obtain

$$\begin{aligned}\frac{\gamma L^2}{2} + \frac{R^2}{2\gamma T} &= \frac{R}{L\sqrt{T}} \cdot \frac{L^2}{2} + \frac{R^2}{2T} \cdot \frac{L\sqrt{T}}{R} \\ &= \frac{RL}{2\sqrt{T}} + \frac{RL}{2\sqrt{T}} \\ &= \frac{RL}{\sqrt{T}} \,.\end{aligned}$$

Since we chose $\epsilon = RL/\sqrt{T}$ in the theorem statement, the proof is complete. ■

Continuing under the assumption that we know R (from equation (33.20)) and L (from inequality (33.21)), we can think of the analysis in a slightly different way. We can presume that we have a target accuracy ϵ and then compute the number of iterations needed. That is, we can solve $\epsilon = RL/\sqrt{T}$ for T, obtaining $T = R^2L^2/\epsilon^2$. The number of iterations thus depends on the square of R and L and, most importantly, on $1/\epsilon^2$. (The definition of L from inequality (33.21) depends on T, but we may know an upper bound on L that doesn't depend on the particular value of T.) Thus, if you want to halve your error bound, you need to run four times as many iterations.

It is quite possible that we don't really know R and L, since you'd need to know $\mathbf{x}^*$ in order to know R (since $R = \|\mathbf{x}^{(0)} - \mathbf{x}^*\|$), and you might not have an explicit upper bound on the gradient, which would provide L. You can, however, interpret the analysis of gradient descent as a proof that there is some step size for which the procedure makes progress toward the minimum. You can then compute a step size γ for which $f(\mathbf{x}^{(t)}) - f(\mathbf{x}^{(t+1)})$ is large enough. In fact, not having a fixed step size multiplier can actually help in practice, as you are free to use any step size s that achieves sufficient decrease in the value of f. You can search for a step size that achieves a large decrease via a binary-search-like routine, which is often called ***line search***. For a given function f and step size s, define the function $g(\mathbf{x}^{(t)}, s) = f(\mathbf{x}^{(t)}) - s(\nabla f)(\mathbf{x}^{(t)})$. Start with a small step size s for which $g(\mathbf{x}^{(t)}, s) \le f(\mathbf{x}^{(t)})$. Then repeatedly double s until $g(\mathbf{x}^{(t)}, 2s) \ge g(\mathbf{x}^{(t)}, s)$, and then perform a binary search in the interval $[s, 2s]$. This procedure can produce a step size that achieves a significant decrease in the objective function. In other circumstances, however, you may know good upper bounds on R and L, typically from problem-specific information, which can suffice.

The dominant computational step in each iteration of the **for** loop of lines 2–4 is computing the gradient. The complexity of computing and evaluating a gradient varies widely, depending on the application at hand. We'll discuss several applications later.

Constrained gradient descent

We can adapt gradient descent for constrained minimization to minimize a closed convex function $f(\mathbf{x})$, subject to the additional requirement that $\mathbf{x} \in K$, where K is a closed convex body. A ***body*** $K \subseteq \mathbb{R}^n$ is ***convex*** if for all $\mathbf{x}, \mathbf{y} \in K$, the convex combination $\lambda\mathbf{x} + (1-\lambda)\mathbf{y} \in K$ for all $0 \le \lambda \le 1$. A ***closed*** convex body contains its limit points. Somewhat surprisingly, restricting to the constrained problem does not significantly increase the number of iterations of gradient descent. The idea is that you run the same algorithm, but in each iteration, check whether the current point $\mathbf{x}^{(t)}$ is still within the convex body K. If it is not, just move to the closest point in K. Moving to the closest point is known as ***projection***. We formally define the projection $\Pi_K(\mathbf{x})$ of a point $\mathbf{x}$ in n dimensions onto a convex body K as the point $\mathbf{y} \in K$ such that $\|\mathbf{x} - \mathbf{y}\| = \min\{\|\mathbf{x} - \mathbf{z}\| : z \in K\}$. If we have $\mathbf{x} \in K$, then $\Pi_K(\mathbf{x}) = \mathbf{x}$.

This one change yields the procedure GRADIENT-DESCENT-CONSTRAINED, in which line 4 of GRADIENT-DESCENT is replaced by two lines. It assumes that $\mathbf{x}^{(0)} \in K$. Line 4 of GRADIENT-DESCENT-CONSTRAINED moves in the direction of the negative gradient, and line 5 projects back onto K. The lemma that follows helps to show that when $\mathbf{x}^* \in K$, if the projection step in line 5 moves from a point outside of K to a point in K, it cannot be moving away from $\mathbf{x}^*$.

```
GRADIENT-DESCENT-CONSTRAINED(f, x^(0), γ, T, K)
1  sum = 0                                  // n-dimensional vector, initially all 0
2  for t = 0 to T − 1
3     sum = sum + x^(t)                     // add each of n dimensions into sum
4     x'^(t+1) = x^(t) − γ · (∇f)(x^(t))    // (∇f)(x^(t)), x'^(t+1) are n-dimensional
5     x^(t+1) = Π_K(x'^(t+1))               // project onto K
6  x-avg = sum/T                            // divide each of n dimensions by T
7  return x-avg
```

Lemma 33.10

Consider a convex body $K \subseteq \mathbb{R}^n$ and points $\mathbf{a} \in K$ and $\mathbf{b}' \in \mathbb{R}^n$. Let $\mathbf{b} = \Pi_K(\mathbf{b}')$. Then $\|\mathbf{b} - \mathbf{a}\|^2 \le \|\mathbf{b}' - \mathbf{a}\|^2$.

Proof If $\mathbf{b}' \in K$, then $\mathbf{b} = \mathbf{b}'$ and the claim is true. Otherwise, $\mathbf{b}' \ne \mathbf{b}$, and as Figure 33.6 shows, we can extend the line segment between $\mathbf{b}$ and $\mathbf{b}'$ to a line ℓ. Let $\mathbf{c}$ be the projection of $\mathbf{a}$ onto ℓ. Point $\mathbf{c}$ may or may not be in K, and if $\mathbf{a}$ is on the boundary of K, then $\mathbf{c}$ could coincide with $\mathbf{b}$. If $\mathbf{c}$ coincides with $\mathbf{b}$ (part (c) of the figure), then $\mathbf{abb}'$ is a right triangle, and so $\|\mathbf{b} - \mathbf{a}\|^2 \le \|\mathbf{b}' - \mathbf{a}\|^2$.

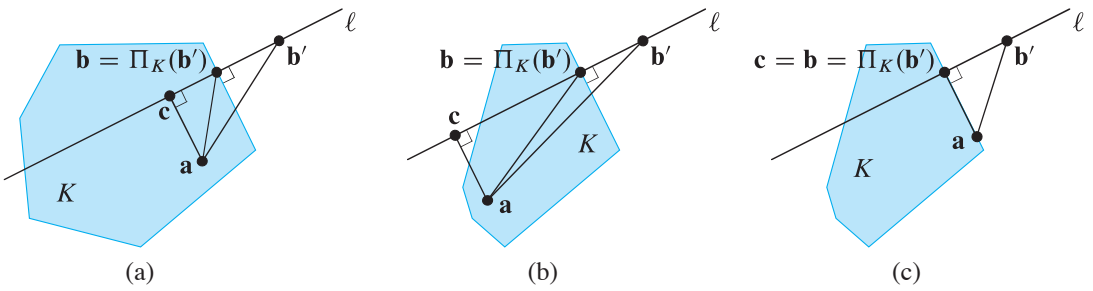


Figure 33.6 Projecting a point $\mathbf{b}'$ outside the convex body K to the closest point $\mathbf{b} = \Pi_K(\mathbf{b}')$ in K. Line ℓ is the line containing $\mathbf{b}$ and $\mathbf{b}'$, and point $\mathbf{c}$ is the projection of $\mathbf{a}$ onto ℓ. **(a)** When $\mathbf{c}$ is in K. **(b)** When $\mathbf{c}$ is not in K. **(c)** When $\mathbf{a}$ is on the boundary of K and $\mathbf{c}$ coincides with $\mathbf{b}$.

If $\mathbf{c}$ does not coincide with $\mathbf{b}$ (parts (a) and (b) of the figure), then because of convexity, the angle $\angle\mathbf{abb}'$ must be obtuse. Because angle $\angle\mathbf{abb}'$ is obtuse, $\mathbf{b}$ lies between $\mathbf{c}$ and $\mathbf{b}'$ on ℓ. Furthermore, because $\mathbf{c}$ is the projection of $\mathbf{a}$ onto line ℓ, $\mathbf{acb}$ and $\mathbf{acb}'$ must be right triangles. By the Pythagorean theorem, we have that $\|\mathbf{b}' - \mathbf{a}\|^2 = \|\mathbf{a} - \mathbf{c}\|^2 + \|\mathbf{c} - \mathbf{b}'\|^2$ and $\|\mathbf{b} - \mathbf{a}\|^2 = \|\mathbf{a} - \mathbf{c}\|^2 + \|\mathbf{c} - \mathbf{b}\|^2$. Subtracting these two equations gives $\|\mathbf{b}' - \mathbf{a}\|^2 - \|\mathbf{b} - \mathbf{a}\|^2 = \|\mathbf{c} - \mathbf{b}'\|^2 - \|\mathbf{c} - \mathbf{b}\|^2$. Because $\mathbf{b}$ is between $\mathbf{c}$ and $\mathbf{b}'$, we must have $\|\mathbf{c} - \mathbf{b}'\|^2 \geq \|\mathbf{c} - \mathbf{b}\|^2$, and thus $\|\mathbf{b}' - \mathbf{a}\|^2 - \|\mathbf{b} - \mathbf{a}\|^2 \geq 0$. The lemma follows. ■

We can now repeat the entire proof for the unconstrained case and obtain the same bounds. Lemma 33.10 with $\mathbf{a} = \mathbf{x}^*$, $\mathbf{b} = \mathbf{x}^{(t+1)}$, and $\mathbf{b}' = \mathbf{x}'^{(t+1)}$ tells us that $\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\|^2 \leq \|\mathbf{x}'^{(t+1)} - \mathbf{x}^*\|^2$. We can therefore derive an upper bound that matches inequality (33.31). We continue to define $\Phi(t)$ as in equation (33.26), but noting that $\mathbf{x}^{(t+1)}$, computed in line 5 of GRADIENT-DESCENT-CONSTRAINED, has a different meaning here from in inequality (33.31):

$$
\begin{aligned}
&\Phi(t+1) - \Phi(t) \\
&\quad = \frac{1}{2\gamma}\left\|\mathbf{x}^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 && \text{(by equation (33.27))} \\
&\quad \leq \frac{1}{2\gamma}\left\|\mathbf{x}'^{(t+1)} - \mathbf{x}^*\right\|^2 - \frac{1}{2\gamma}\left\|\mathbf{x}^{(t)} - \mathbf{x}^*\right\|^2 && \text{(by Lemma 33.10)} \\
&\quad = \frac{1}{2\gamma}\left(2\langle\mathbf{x}'^{(t+1)} - \mathbf{x}^{(t)}, \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|\mathbf{x}'^{(t+1)} - \mathbf{x}^*\right\|^2\right) \\
& && \text{(by equation (33.29))} \\
&\quad = \frac{1}{2\gamma}\left(2\langle-\gamma\cdot(\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \left\|-\gamma\cdot(\nabla f)(\mathbf{x}^{(t)})\right\|^2\right) \\
& && \text{(by line 4 of GRADIENT-DESCENT-CONSTRAINED)} \\
&\quad = -\langle(\nabla f)(\mathbf{x}^{(t)}), \mathbf{x}^{(t)} - \mathbf{x}^*\rangle + \frac{\gamma}{2}\left\|(\nabla f)(\mathbf{x}^{(t)})\right\|^2 .
\end{aligned}
$$

With the same upper bound on the change in the potential function as in equation (33.30), the entire proof of Lemma 33.9 can proceed as before. We can therefore conclude that the procedure GRADIENT-DESCENT-CONSTRAINED has the same asymptotic complexity as GRADIENT-DESCENT. We summarize this result in the following theorem.

Theorem 33.11

Let $K \subseteq \mathbb{R}^n$ be a convex body, $\mathbf{x}^* \in \mathbb{R}^n$ be the minimizer of a convex function f over K, and $\gamma = R/(L\sqrt{T})$, where R and L are defined in equations (33.20) and (33.21). Suppose that the vector **x-avg** is returned by an execution of GRADIENT-DESCENT-CONSTRAINED$(f, \mathbf{x^{(0)}}, \gamma, T, K)$. Let $\epsilon = RL/\sqrt{T}$. Then we have $f(\textbf{x-avg}) - f(\mathbf{x}^*) \le \epsilon$. ■

Applications of gradient descent

Gradient descent has many applications to minimizing functions and is widely used in optimization and machine learning. Here we sketch how it can be used to solve linear systems. Then we discuss an application to machine learning: prediction using linear regression.

In Chapter 28, we saw how to use Gaussian elimination to solve a system of linear equations $A\mathbf{x} = \mathbf{b}$, thereby computing $\mathbf{x} = A^{-1}\mathbf{b}$. If A is an $n \times n$ matrix and $\mathbf{b}$ is a length-n vector, then the running time of Gaussian elimination is $\Theta(n^3)$, which for large matrices might be prohibitively expensive. If an approximate solution is acceptable, however, you can use gradient descent.

First, let's see how to use gradient descent as a roundabout—and admittedly inefficient—way to solve for x in the scalar equation $ax = b$, where $a, x, b \in \mathbb{R}$. This equation is equivalent to $ax - b = 0$. If $ax - b$ is the derivative of a convex function $f(x)$, then $ax - b = 0$ for the value of x that minimizes $f(x)$. Given $f(x)$, gradient descent can then determine this minimizer. Of course, $f(x)$ is just the integral of $ax - b$, that is, $f(x) = \frac{1}{2}ax^2 - bx$, which is convex if $a \ge 0$. Therefore, one way to solve $ax = b$ for $a \ge 0$ is to find the minimizer for $\frac{1}{2}ax^2 - bx$ via gradient descent.

We now generalize this idea to higher dimensions, where using gradient descent may actually lead to a faster algorithm. One n-dimensional analog is the function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^{\mathrm{T}}A\mathbf{x} - \mathbf{b}^{\mathrm{T}}\mathbf{x}$, where A is an $n \times n$ matrix. The gradient of f with respect to $\mathbf{x}$ is the function $A\mathbf{x} - \mathbf{b}$. To find the value of $\mathbf{x}$ that minimizes f, we set the gradient of f to 0 and solve for $\mathbf{x}$. Solving $A\mathbf{x} - \mathbf{b} = 0$ for $\mathbf{x}$, we obtain $\mathbf{x} = A^{-1}\mathbf{b}$, Thus, minimizing $f(\mathbf{x})$ is equivalent to solving $A\mathbf{x} = \mathbf{b}$. If $f(\mathbf{x})$ is convex, then gradient descent can approximately compute this minimum.

A 1-dimensional function is convex when its second derivative is positive. The equivalent definition for a multidimensional function is that it is convex when its

Hessian matrix is positive-semidefinite (see page 1222 for a definition), where the ***Hessian matrix*** $(\nabla^2 f)(\mathbf{x})$ of a function $f(\mathbf{x})$ is the matrix in which entry (i, j) is the partial derivative of f with respect to i and j:

$$(\nabla^2 f)(\mathbf{x}) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{pmatrix} .$$

Analogous to the 1-dimensional case, the Hessian of f is just A, and so if A is a positive-semidefinite matrix, then we can use gradient descent to find a point $\mathbf{x}$ where $A\mathbf{x} \approx \mathbf{b}$. If R and L are not too large, then this method is faster than using Gaussian elimination.

Gradient descent in machine learning

As a concrete example of supervised learning for prediction, suppose that you want to predict whether a patient will develop heart disease. For each of m patients, you have n different attributes. For example, you might have $n = 4$ and the four pieces of data are age, height, blood pressure, and number of close family members with heart disease. Denote the data for patient i as a vector $\mathbf{x}^{(i)} \in \mathbb{R}^n$, with $x_j^{(i)}$ giving the jth entry in vector $\mathbf{x}^{(i)}$. The ***label*** of patient i is denoted by a scalar $y^{(i)} \in \mathbb{R}$, signifying the severity of the patient's heart disease. The hypothesis should capture a relationship between the $\mathbf{x}^{(i)}$ values and $y^{(i)}$. For this example, we make the modeling assumption that the relationship is linear, and therefore the goal is to compute the "best" linear relationship between the $\mathbf{x}^{(i)}$ values and $y^{(i)}$: a linear function $f : \mathbb{R}^n \to \mathbb{R}$ such that $f(\mathbf{x}^{(i)}) \approx y^{(i)}$ for each patient i. Of course, no such function may exist, but you would like one that comes as close as possible. A linear function f can be defined by a vector of weights $\mathbf{w} = (w_0, w_1, \ldots, w_n)$, with

$$f(\mathbf{x}) = w_0 + \sum_{j=1}^{n} w_j x_j \ . \tag{33.32}$$

When evaluating a machine-learning model, you need to measure how close each value $f(\mathbf{x}^{(i)})$ is to its corresponding label $y^{(i)}$. In this example, we define the error $e^{(i)} \in \mathbb{R}$ associated with patient i as $e^{(i)} = f(\mathbf{x}^{(i)}) - y^{(i)}$. The objective function we choose is to minimize the sum of squares of the errors, which is

$$\sum_{i=1}^{m} \left(e^{(i)}\right)^2 = \sum_{i=1}^{m} \left(f(\mathbf{x}^{(i)}) - y^{(i)}\right)^2$$
$$= \sum_{i=1}^{m} \left(w_0 + \sum_{j=1}^{n} w_j x_j^{(i)} - y^{(i)}\right)^2 . \tag{33.33}$$

The objective function is typically called the ***loss function***, and the ***least-squares error*** given by equation (33.33) is just one example of many possible loss functions. The goal is then, given the $\mathbf{x}^{(i)}$ and $y^{(i)}$ values, to compute the weights $w_0, w_1, \ldots, w_n$ so as to minimize the loss function in equation (33.33). The variables here are the weights $w_0, w_1, \ldots, w_n$ and not the $\mathbf{x}^{(i)}$ or $y^{(i)}$ values.

This particular objective is sometimes known as a ***least-squares fit***, and the problem of finding a linear function to fit data and minimize the least-squares error is called ***linear regression***. Finding a least-squares fit is also addressed in Section 28.3.

When the function f is linear, the loss function defined in equation (33.33) is convex, because it is the sum of squares of linear functions, which are themselves convex. Therefore, we can apply gradient descent to compute a set of weights to approximately minimize the least-squares error. The concrete goal of learning is to be able to make predictions on new data. Informally, if the features are all reported in the same units and are from the same range (perhaps from being normalized), then the weights tend to have a natural interpretation because the features of the data that are better predictors of the label have a larger associated weight. For example, you would expect that, after normalization, the weight associated with the number of family members with heart disease would be larger than the weight associated with height.

The computed weights form a model of the data. Once you have a model, you can make predictions, so that given new data, you can predict its label. In our example, given a new patient $\mathbf{x}'$ who is not part of the original training data set, you would still hope to predict the chance that the new patient develops heart disease. You can do so by computing the label $f(\mathbf{x}')$, incorporating the weights computed by gradient descent.

For this linear-regression problem, the objective is to minimize the expression in equation (33.33), which is a quadratic in each of the $n+1$ weights w_j. Thus, entry j in the gradient is linear in w_j. Exercise 33.3-5 asks you to explicitly compute the gradient and see that it can be computed in $O(nm)$ time, which is linear in the input size. Compared with the exact method of solving equation (33.33) in Chapter 28, which needs to invert a matrix, gradient descent is typically much faster.

Section 33.1 briefly discussed regularization—the idea that a complicated hypothesis should be penalized in order to avoid overfitting the training data. Regularization often involves adding a term to the objective function, but it can also

be achieved by adding a constraint. One way to regularize this example would be to explicitly limit the norm of the weights, adding a constraint that $\|\mathbf{w}\| \leq B$ for some bound $B > 0$. (Recall again that the components of the vector $\mathbf{w}$ are the variables in the present application.) Adding this constraint controls the complexity of the model, as the number of values w_j that can have large absolute value is now limited.

In order to run GRADIENT-DESCENT-CONSTRAINED for any problem, you need to implement the projection step, as well as to compute bounds on R and L. We conclude this section by describing these calculations for gradient descent with the constraint $\|\mathbf{w}\| \leq B$. First, consider the projection step in line 5. Suppose that the update in line 4 results in a vector $\mathbf{w}'$. The projection is implemented by computing $\Pi_k(\mathbf{w}')$ where K is defined by $\|\mathbf{w}\| \leq B$. This particular projection can be accomplished by simply scaling $\mathbf{w}'$, since we know that closest point in K to $\mathbf{w}'$ must be the point along the vector whose norm is exactly B. The amount z by which we need to scale $\mathbf{w}'$ to hit the boundary of K is the solution to the equation $z\|\mathbf{w}'\| = B$, which is solved by $z = B/\|\mathbf{w}'\|$. Hence line 5 is implemented by computing $\mathbf{w} = \mathbf{w}'B/\|\mathbf{w}'\|$. Because we always have $\|\mathbf{w}\| \leq B$, Exercise 33.3-6 asks you to show that the upper bound on the magnitude L of the gradient is $O(B)$. We also get a bound on R, as follows. By the constraint $\|\mathbf{w}\| \leq B$, we know that both $\|\mathbf{w}^{(0)}\| \leq B$ and $\|\mathbf{w}^*\| \leq B$, and thus $\|\mathbf{w}^{(0)} - \mathbf{w}^*\| \leq 2B$. Using the definition of R in equation (33.20), we have $R = O(B)$. The bound $RL/\sqrt{T}$ on the accuracy of the solution after T iterations in Theorem 33.11 becomes $O(B)L/\sqrt{T} = O(B^2/\sqrt{T})$.

Exercises

33.3-1

Prove Lemma 33.6. Start from the definition of a convex function given in equation (33.18). (*Hint:* You can prove the statement when $n = 1$ first. The proof for general values of n is similar.)

33.3-2

Prove Lemma 33.7.

33.3-3

Prove equation (33.29). (*Hint:* The proof for $n = 1$ dimension is straightforward. The proof for general values of n dimensions follows along similar lines.)

33.3-4

Show that the function f in equation (33.32) is a convex function of the variables $w_0, w_1, \ldots, w_n$.

33.3-5
Compute the gradient of expression (33.33) and explain how to evaluate the gradient in $O(nm)$ time.

33.3-6
Consider the function f defined in equation (33.32), and suppose that you have a bound $\|\mathbf{w}\| \leq B$, as is considered in the discussion on regularization. Show that $L = O(B)$ in this case.

33.3-7
Equation (33.2) on page 1009 gives a function that, when minimized, gives an optimal solution to the k-means problem. Explain how to use gradient descent to solve the k-means problem.

Problems

33-1 Newton's method
Gradient descent iteratively moves closer to a desired value (the minimum) of a function. Another algorithm in this spirit is known as ***Newton's method***, which is an iterative algorithm that finds the root of a function. Here, we consider Newton's method which, given a function $f : \mathbb{R} \to \mathbb{R}$, finds a value x^* such that $f(x^*) = 0$. The algorithm moves through a series of points $x^{(0)}, x^{(1)}, \ldots$. If the algorithm is currently at a point $x^{(t)}$, then to find point $x^{(t+1)}$, it first takes the equation of the line tangent to the curve at $x = x^{(t)}$,

$$y = f'(x^{(t)})(x - x^{(t)}) + f(x^{(t)}) .$$

It then uses the x-intercept of this line as the next point $x^{(t+1)}$.

a. Show that the algorithm described above can be summarized by the update rule

$$x^{(t+1)} = x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})} .$$

We restrict our attention to some domain I and assume that $f'(x) \neq 0$ for all $x \in I$ and that $f''(x)$ is continuous. We also assume that the starting point $x^{(0)}$ is sufficiently close to x^*, where "sufficiently close" means that we can use only the first two terms of the Taylor expansion of $f(x^*)$ about $x^{(0)}$, namely

$$f(x^*) = f(x^{(0)}) + f'(x^{(0)})(x^* - x^{(0)}) + \frac{1}{2} f''(\gamma^{(0)})(x^* - x^{(0)})^2 , \tag{33.34}$$

where $\gamma^{(0)}$ is some value between $x^{(0)}$ and x^*. If the approximation in equation (33.34) holds for $x^{(0)}$, it also holds for any point closer to x^*.

b. Assume that the function f has exactly one point x^* for which $f(x^*) = 0$. Let $\epsilon^{(t)} = \left|x^{(t)} - x^*\right|$. Using the Taylor expansion in equation (33.34), show that

$$\epsilon^{(t+1)} = \frac{\left|f''(\gamma^{(t)})\right|}{2\left|f'(\gamma^{(t)})\right|}\epsilon^{(t)} ,$$

where $\gamma^{(t)}$ is some value between $x^{(t)}$ and x^*.

c. If

$$\frac{\left|f''(\gamma^{(t)})\right|}{2\left|f'(\gamma^{(t)})\right|} \le c$$

for some constant c and $\epsilon^{(0)} < 1$, then we say that the function f has ***quadratic convergence***, since the error decreases quadratically. Assuming that f has quadratic convergence, how many iterations are needed to find a root of $f(x)$ to an accuracy of δ? Your answer should include δ.

d. Suppose you wish to find a root of the function $f(x) = (x-3)^2$, which is also the minimizer, and you start at $x^{(0)} = 3.5$. Compare the number of iterations needed by gradient descent to find the minimizer and Newton's method to find the root.

33-2 Hedge

Another variant in the multiplicative-weights framework is known as HEDGE. It differs from WEIGHTED MAJORITY in two ways. First, HEDGE makes the prediction randomly—in iteration t, it assigns a probability $p_i^{(t)} = w_i^{(t)}/Z^{(t)}$ to expert E_i, where $Z^{(t)} = \sum_{i=1}^{n} w_i^{(t)}$. It then chooses an expert $E_{i'}$ according to this probability distribution and predicts according to $E_{i'}$. Second, the update rule is different. If an expert makes a mistake, line 16 updates that expert's weight by the rule $w_i^{(t+1)} = w_i^{(t)} e^{-\epsilon}$, for some $0 < \epsilon < 1$. Show that the expected number of mistakes made by HEDGE, running for T rounds, is at most $m^* + (\ln n)/\epsilon + \epsilon T$.

33-3 Nonoptimality of Lloyd's procedure in one dimension

Give an example to show that even in one dimension, Lloyd's procedure for finding clusters does not always return an optimum result. That is, Lloyd's procedure may terminate and return as a result a set C of clusters that does not minimize $f(S, C)$, even when S is a set of points on a line.

33-4 Stochastic gradient descent

Consider the problem described in Section 33.3 of fitting a line $f(x) = ax + b$ to a given set of point/value pairs $S = \{(x_1, y_1), ..., (x_T, y_T)\}$ by optimizing the choice of the parameters a and b using gradient descent to find a best least-squares fit. Here we consider the case where x is a real-valued variable, rather than a vector.

Suppose that you are not given the point/value pairs in S all at once, but only one at a time in an online manner. Furthermore, the points are given in random order. That is, you know that there are n points, but in iteration t you are given only (x_i, y_i) where i is independently and randomly chosen from $\{1, \ldots, T\}$.

You can use gradient descent to compute an estimate to the function. As each point (x_i, y_i) is considered, you can update the current values of a and b by taking the derivative with respect to a and b of the term of the objective function depending on (x_i, y_i). Doing so gives you a stochastic estimate of the gradient, and you can then take a small step in the opposite direction.

Give pseudcode to implement this variant of gradient descent. What would the expected value of the error be as a function of T, L, and R? (*Hint:* Replicate the analysis of GRADIENT-DESCENT in Section 33.3 for this variant.)

This procedure and its variants are known as ***stochastic gradient descent***.

Chapter notes

For a general introduction to artificial intelligence, we recommend Russell and Norvig [391]. For a general introduction to machine learning, we recommend Murphy [340].

Lloyd's procedure for the k-means problem was first proposed by Lloyd [304] and also later by Forgy [151]. It is sometimes called "Lloyd's algorithm" or the "Lloyd-Forgy algorithm." Although Mahajan et al. [310] showed that finding an optimal clustering is NP-hard, even in the plane, Kanungo et al. [241] have shown that there is an approximation algorithm for the k-means problem with approximation ratio $9 + \epsilon$, for any $\epsilon > 0$.

The multiplicative-weights method is surveyed by Arora, Hazan, and Kale [25]. The main idea of updating weights based on feedback has been rediscovered many times. One early use is in game theory, where Brown defined "Fictitious Play" [74] and conjectured its convergence to the value of a zero-sum game. The convergence properties were established by Robinson [382].

In machine learning, the first use of multiplicative weights was by Littlestone in the Winnow algorithm [300], which was later extended by Littlestone and Warmuth to the weighted-majority algorithm described in Section 33.2 [301]. This work is closely connected to the boosting algorithm, originally due to Freund and Shapire

[159]. The multiplicative-weights idea is also closely related to several more general optimization algorithms, including the perceptron algorithm [328] and algorithms for optimization problems such as packing linear programs [177, 359].

The treatment of gradient descent in this chapter draws heavily on the unpublished manuscript of Bansal and Gupta [35]. They emphasize the idea of using a potential function and using ideas from amortized analysis to explain gradient descent. Other presentations and analyses of gradient descent include works by Bubeck [75], Boyd and Vanderberghe [69], and Nesterov [343].

Gradient descent is known to converge faster when functions obey stronger properties than general convexity. For example, a function f is ***α-strongly convex*** if $f(\mathbf{y}) \geq f(\mathbf{x}) + \langle (\nabla f)(\mathbf{x}), (\mathbf{y} - \mathbf{x}) \rangle + \alpha \|\mathbf{y} - \mathbf{x}\|$ for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. In this case, GRADIENT-DESCENT can use a variable step size and return $\mathbf{x}^{(T)}$. The step size at step t becomes $\gamma_t = 1/(\alpha(t+1))$, and the procedure returns a point such that $f(\mathbf{x}\text{-avg}) - f(\mathbf{x}^*) \leq L^2/(\alpha(T+1))$. This convergence is better than that of Theorem 33.8 because the number of iterations needed is linear, rather than quadratic, in the desired error parameter ϵ, and because the performance is independent of the initial point.

Another case in which gradient descent can be shown to perform better than the analysis in Section 33.3 suggests is for smooth convex functions. We say that a function is ***β-smooth*** if $f(\mathbf{y}) \leq f(\mathbf{x}) + \langle (\nabla f)(\mathbf{x}), (\mathbf{y} - \mathbf{x}) \rangle + \frac{\beta}{2} \|\mathbf{y} - \mathbf{x}\|^2$. This inequality goes in the opposite direction from the one for α-strong convexity. Better bounds on gradient descent are possible here as well.

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 long you're willing to wait for an answer.[1] There are also problems that can be solved, but not in $O(n^k)$ time 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 the problems appears to be slight:

Shortest versus longest simple paths: In Chapter 22, we saw that even with negative edge weights, we can find *shortest* paths from a single source in a directed

[1] For the Halting Problem and other unsolvable problems, there are proofs that no algorithm can exist that, for every input, eventually produces the correct answer. A procedure attempting to solve an unsolvable problem might always produce an answer but is sometimes incorrect, or all the answers it produces might be correct but for some inputs it never produces an answer.

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 versus hamiltonian cycle: An ***Euler tour*** of a strongly 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. Problem 20-3 on page 583 asks you to show how to determine whether a strongly connected, directed graph has an Euler tour and, if it does, the order of the edges in the Euler tour, all 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'll prove that determining whether an *undirected* graph has a hamiltonian cycle is NP-complete.)

2-CNF satisfiability versus 3-CNF satisfiability: Boolean formulas contain binary 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'll 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 $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$ is in 2-CNF (with satisfying assignment $x_1 = 1$, $x_2 = 0$, and $x_3 = 1$). Although there is a polynomial-time algorithm to determine whether a 2-CNF formula is satisfiable, we'll 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 refer to three classes of problems: P, NP, and NPC, the latter class being the NP-complete problems. We describe them informally here, with formal definitions to appear 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 $O(n^k)$ time for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous chapters belong to P.

The class NP consists of those problems that are "verifiable" in polynomial time. What do we mean by a problem being verifiable? If you were somehow given a "certificate" of a solution, then you 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. You could check in polynomial time that the sequence contains each of the $|V|$ vertices exactly once, 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 another example, for 3-CNF satisfiability, a certificate could be an assignment of values to variables. You could check in polynomial time that this assignment satisfies the boolean formula.

Any problem in P also belongs to NP, since if a problem belongs to P then it is solvable in polynomial time without even being supplied a certificate. We'll formalize this notion later in this chapter, but for now you can believe that $\text{P} \subseteq \text{NP}$. The famous open question is whether P is a proper subset of NP.

Informally, a problem belongs to the class NPC—and we call it ***NP-complete*** —if it belongs to NP and is as "hard" as any problem in NP. We'll formally define what it means to be as hard as any problem in NP later in this chapter. In the meantime, we 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 could turn out to be 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 used 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. If you can demonstrate that a problem is NP-complete, you 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. If you prove a problem NP-complete, you are saying that searching for efficient algorithm is likely to be a fruitless endeavor. 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, although the specific techniques used for showing NP-completeness differ from the decision-tree method used in Section 8.1.

We rely on three key concepts in showing a problem to be NP-complete:

Decision problems versus optimization problems

Many problems of interest are ***optimization problems***, in which each feasible (i.e., "legal") solution has an associated value, and the goal is to find a feasible solution with the best value. For example, in a problem that we call SHORTEST-PATH, the input is an undirected graph G and vertices u and v, and the goal is 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, there is usually a way to 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 an undirected 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 your favor when you 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, you 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 you can provide evidence that a decision problem is hard, you 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. Almost every NP-completeness proof takes advantage of this idea, as follows. Consider a decision problem A, which you 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

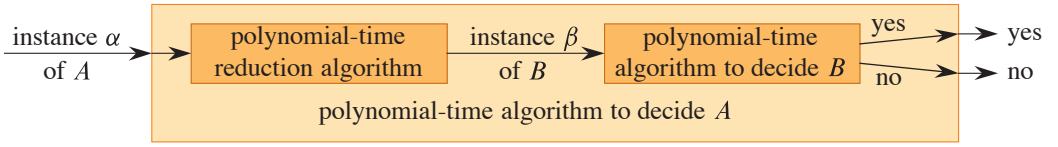


Figure 34.1 How to use a polynomial-time reduction algorithm to solve a decision problem A in polynomial time, given a polynomial-time decision algorithm for another problem B. In polynomial time, transform an instance α of A into an instance β of B, solve B in polynomial time, and use the answer for β as the answer for α.

instance is a particular graph G, particular vertices u and v of G, and a particular integer k. Now suppose that you already know how to solve a different decision problem B in polynomial time. Finally, suppose that you 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."

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 you have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B, you 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, you use polynomial-time reductions in the opposite way to show that a problem is NP-complete. Let's take the idea a step further and show how you can use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B. Suppose that you have a decision problem A for which you already know that no polynomial-time algorithm can exist. (Ignore for the moment how to find such a problem A.) Suppose further that you have a polynomial-time reduction transforming instances of A to instances of B. Now you can use a simple proof by contradiction to show that no polynomial-time algorithm can exist for B. Suppose otherwise, that is, suppose that B has a

polynomial-time algorithm. Then, using the method shown in Figure 34.1, you would have a way to solve problem A in polynomial time, which contradicts the assumption that there is no polynomial-time algorithm for A.

To prove that a problem B is NP-complete, the methodology is similar. Although you cannot assume that there is absolutely no polynomial-time algorithm for problem A, you 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, there must be some "first" NP-complete problem. We'll use the circuit-satisfiability problem, in which the input is a boolean combinational circuit composed of AND, OR, and NOT gates, and the question is whether there exists some set of boolean inputs to this circuit that causes its output to be 1. Section 34.3 will prove that this first problem is NP-complete.

Chapter outline

This chapter studies the aspects of NP-completeness that bear most directly on the analysis of algorithms. Section 34.1 formalizes the notion of "problem" and defines the complexity class P of polynomial-time solvable decision problems. We'll 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 $\mathrm{P} \neq \mathrm{NP}$ question.

Section 34.3 shows how to relate problems via polynomial-time "reductions." It defines NP-completeness and sketches a proof that the circuit-satisfiability problem is NP-complete. With one problem proven NP-complete, Section 34.4 demonstrates how to prove other problems to be NP-complete much more simply by the methodology of reductions. To illustrate this methodology, the section shows that two formula-satisfiability problems are NP-complete. Section 34.5 proves a variety of other problems to be NP-complete by using reductions. You will probably find several of these reductions to be quite creative, because they convert a problem in one domain to a problem in a completely different domain.

34.1 Polynomial time

Since NP-completeness relies on notions of solving a problem and verifying a certificate in polynomial time, let's first examine what it means for a problem to be solvable in polynomial time.

Recall that we generally regard problems that have polynomial-time solutions as tractable. Here are three reasons why:

1. Although no reasonable person considers a problem that requires $\Theta(n^{100})$ time to be tractable, 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.
2. 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.[2] 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.
3. 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, you must first have a formal notion of what a "problem" is. We define an ***abstract problem*** Q to be a

[2] See the books by Hopcroft and Ullman [228], Lewis and Papadimitriou [299], or Sipser [413] for a thorough treatment of the Turing-machine model.

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 necessary 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 PATH, then $\text{PATH}(i) = 1$ (yes) if G contains a path from u to v with 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, you can usually recast an optimization problem as a decision problem that is no harder.

Encodings

In order for a computer program to solve an abstract problem, its problem instances must appear 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.[3] 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 01000001. You 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. The ***size*** of an instance i is just the length of its string, which we denote by $|i|$. 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 $O(T(n))$ time if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$

[3] The codomain of e need not be *binary* strings: any set of strings over a finite alphabet having at least two symbols will do.

time.[4] A concrete problem is ***polynomial-time solvable***, therefore, if there exists an algorithm to solve it in $O(n^k)$ time 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.

Encodings 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 $e(Q)$.[5] 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, 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, ideally with the definition 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 the sole input to an algorithm is an integer k, 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 the input k is provided using the more natural binary representation, however, then the input length is $n = \lfloor \lg k \rfloor + 1$, so the size of the unary encoding is exponential in the size of the binary encoding. With the binary representation, 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.

The encoding of 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.

---

[4] 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 $O(T(n))$.

[5] The notation $\{0,1\}^*$ denotes the set of all strings composed of symbols from the set $\{0,1\}$.

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 $f_{21}(e_2(i)) = e_1(i)$.[6] 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 $e_2(Q) \in \mathrm{P}$.

Proof We need only prove the forward direction, since the backward direction is symmetric. Suppose, therefore, that $e_1(Q)$ can be solved in $O(n^k)$ time for some constant k. Furthermore, suppose that for any problem instance i, the encoding $e_1(i)$ can be computed from the encoding $e_2(i)$ in $O(n^c)$ time for some constant c, where $n = |e_2(i)|$. To solve problem $e_2(Q)$ on input $e_2(i)$, first compute $e_1(i)$ and then run the algorithm for $e_1(Q)$ on $e_1(i)$. How long does this procedure take? Converting encodings takes $O(n^c)$ time, 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 $O(|e_1(i)|^k) = O(n^{ck})$ time, 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. If instances are encoded in unary, however, its complexity may change. In order to be able to converse in an encoding-independent fashion, we generally assume that problem instances are encoded in any reasonable, concise fashion, unless we specifically say otherwise. To be precise, we 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

---

[6] 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.

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 enclose the object in angle brackets. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

As long as the encoding implicitly used 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. From now on, we will generally assume that all problem instances are binary strings encoded using the standard encoding, unless we explicitly specify the contrary. We'll 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 representations 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^*.

Languages support a variety of operations. Set-theoretic operations, such as ***union*** and ***intersection***, follow directly from the set-theoretic definitions. We define the ***complement*** of a language 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

$$\begin{aligned}\text{PATH} = \{\langle G, u, v, k\rangle : \ & G = (V, E) \text{ is an undirected graph,} \\ & u, v \in V, \\ & k \geq 0 \text{ is an integer, and} \\ & G \text{ contains a path from } u \text{ to } v \text{ with at most } k \text{ edges}\} .\end{aligned}$$

(Where convenient, we'll 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 does not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm might 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 $O(n^k)$ time. 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 $O(n^k)$ time. 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 path from u to v in G with the fewest edges, and then compares the number of edges on the 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 straightforward 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.[7]

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, as the following theorem shows, 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 polynomial-time algorithm A. We use a classic "simulation" argument to construct another polynomial-time algorithm A' that decides L. Because A accepts L in $O(n^k)$ time 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. ■

The proof of Theorem 34.2 is nonconstructive. For a given language $L \in \mathrm{P}$, 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.

---

[7] For more on complexity classes, see the seminal paper by Hartmanis and Stearns [210].

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 15.2-2 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 $L_1, L_2 \in$ P, then $L_1 \cup L_2 \in$ P, $L_1 \cap L_2 \in$ P, $L_1 L_2 \in$ P, $\overline{L}_1 \in$ P, and $L_1^* \in$ P.

34.2 Polynomial-time verification

Now, let's 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, you are also given a path p from u to v. You can check whether p is a path in G and whether the length of p is at most k, and if so, you 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 much. After all, PATH belongs to P—in fact, you can solve PATH in linear time—and so verifying membership from a given certificate takes as long as solving the problem from scratch. Instead, let's 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***, and 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 containing all the vertices.[8] 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.

Here is how to 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 whether it is a hamiltonian cycle.

[8] In a letter dated 17 October 1856 to his friend John T. Graves, Hamilton [206, 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."

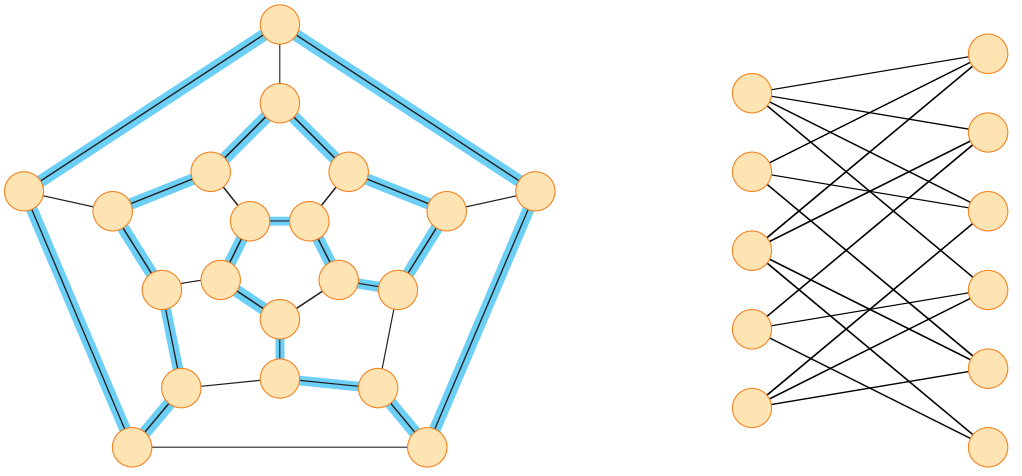


Figure 34.2 **(a)** A graph representing the vertices, edges, and faces of a dodecahedron, with a hamiltonian cycle shown by edges highlighted in blue. **(b)** A bipartite graph with an odd number of vertices. Any such graph is nonhamiltonian.

What is the running time of this algorithm? It depends on the encoding of the graph G. Let's say that G is encoded as its adjacency matrix. If the adjacency matrix contains n entries, so that the length of the encoding of G equals n, then the number m of vertices in the graph is $\Omega(\sqrt{n})$. There are $m!$ possible permutations 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'll 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 the friend 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

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x, y) = 1\} .$$

Think of an algorithm A as verifying 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 that the graph is indeed hamiltonian. 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 so-called 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.[9] 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 about the hamiltonian-cycle problem, you can 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 converted to a two-argument verification algorithm that simply ignores any certificate and accepts exactly those input strings it determines to belong to L. Thus, P $\subseteq$ NP.

That leaves the question of whether P = NP. A definitive answer is unknown, but most researchers believe that P and NP are not the same class. Think of the class P as consisting of problems that can be solved quickly and the class NP as

---

[9] 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 [228] give a good presentation of NP-completeness in terms of nondeterministic models of computation.

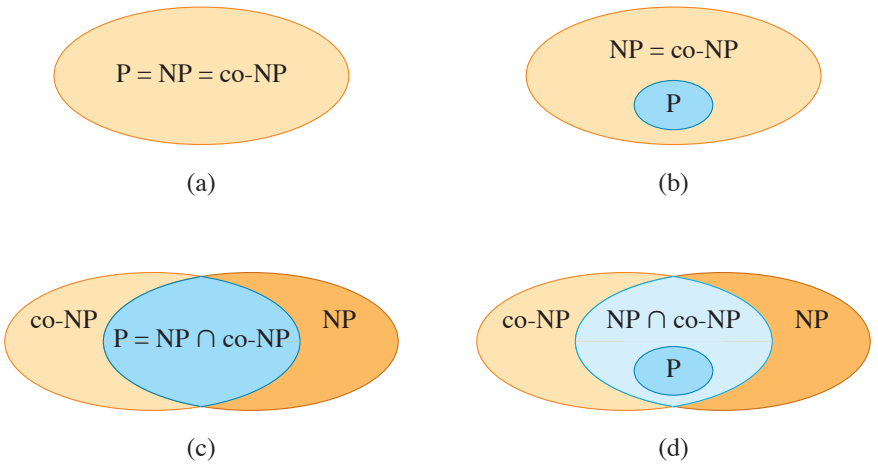


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. **(a)** $\mathrm{P} = \mathrm{NP} = \text{co-NP}$. Most researchers regard this possibility as the most unlikely. **(b)** If NP is closed under complement, then $\mathrm{NP} = \text{co-NP}$, but it need not be the case that $\mathrm{P} = \mathrm{NP}$. **(c)** $\mathrm{P} = \mathrm{NP} \cap \text{co-NP}$, but NP is not closed under complement. **(d)** $\mathrm{NP} \neq \text{co-NP}$ and $\mathrm{P} \neq \mathrm{NP} \cap \text{co-NP}$. Most researchers regard this possibility as the most likely.

consisting 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 do not belong to P.

There is more compelling, though not conclusive, evidence that $\mathrm{P} \neq \mathrm{NP}$—the existence of languages that are "NP-complete." Section 34.3 will study this class.

Many other fundamental questions beyond the $\mathrm{P} \neq \mathrm{NP}$ 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 $L \in \mathrm{NP}$ imply $\overline{L} \in \mathrm{NP}$? We define the ***complexity class* co-NP** as the set of languages L such that $\overline{L} \in \mathrm{NP}$, so that the question of whether NP is closed under complement is also whether $\mathrm{NP} = \text{co-NP}$. Since P is closed under complement (Exercise 34.1-6), it follows from Exercise 34.2-9 ($\mathrm{P} \subseteq \text{co-NP}$) that $\mathrm{P} \subseteq \mathrm{NP} \cap \text{co-NP}$. Once again, however, no one knows whether $\mathrm{P} = \mathrm{NP} \cap \text{co-NP}$ or whether there is some language in $(\mathrm{NP} \cap \text{co-NP}) - \mathrm{P}$.

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 $= \{\langle G_1, G_2\rangle : G_1$ and G_2 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 with a running time of $2^{O(n^k)}$ 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 three 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 P $\neq$ 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, P = NP. Despite decades 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 there were an algorithm to decide HAM-CYCLE in polynomial time, then every problem in NP could be solved in polynomial time. The NP-complete languages are, in a sense, the "hardest" languages in NP. In fact, if NP $-$ P turns out to be nonempty, we will be able to say with certainty that HAM-CYCLE $\in$ NP $-$ P.

This section starts by showing how to compare the relative "hardness" of languages using a precise notion called "polynomial-time reducibility." It then formally defines the NP-complete languages, finishing by sketching a proof that one such language, called CIRCUIT-SAT, is NP-complete. Sections 34.4 and 34.5 will use the notion of reducibility to show that many other problems are NP-complete.

Reducibility

One way that sometimes works for solving a problem is to recast it as a different problem. We call that strategy "reducing" one problem to another. Think of a problem Q as being reducible to another problem Q' if any instance of Q can be recast as an instance of Q', and the solution to the instance of Q' 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 a linear-equation instance $ax + b = 0$ (with solution $x = -b/a$), you can transform it to the quadratic equation $ax^2 + bx + 0 = 0$. This quadratic equation has the solutions $x = (-b \pm \sqrt{b^2 - 4ac})/2a$, where $c = 0$, so that $\sqrt{b^2 - 4ac} = b$. The

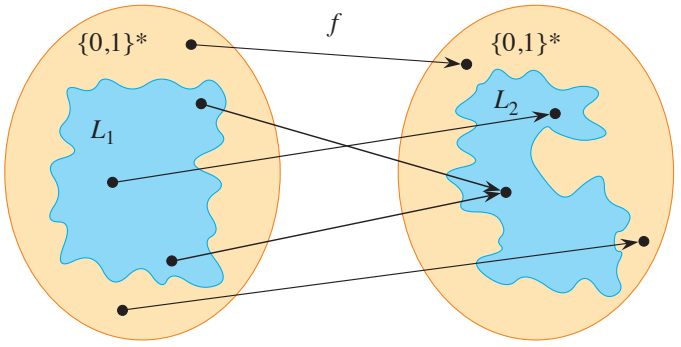


Figure 34.4 A function f that reduces language L_1 to language L_2. For any input $x \in \{0,1\}^*$, the question of whether $x \in L_1$ has the same answer as the question of whether $f(x) \in L_2$.

solutions are then $x = (-b+b)/2a = 0$ and $x = (-b-b)/2a = -b/a$, thereby providing 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 $L_1 \leq_P L_2$, 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 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 mapping such that if $x \in L_1$, 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 $x \in L_1$. If, in addition, f can be computed in polynomial time, it is a polynomial-time reduction function.

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 $L_1 \in P$.

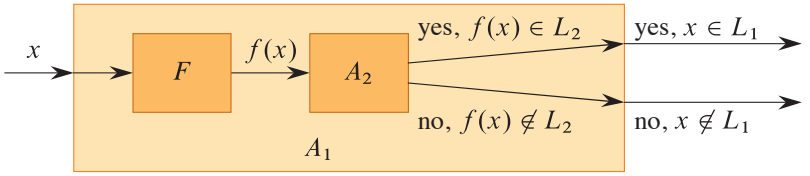


Figure 34.5 The proof of Lemma 34.3. The algorithm F is a reduction algorithm that computes the reduction function f from L_1 to L_2 in polynomial time, and A_2 is a polynomial-time algorithm that decides L_2. Algorithm A_1 decides whether $x \in L_1$ by using F to transform any input x into $f(x)$ and then using A_2 to decide whether $f(x) \in L_2$.

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 show how to 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 allow us to formally show 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 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.

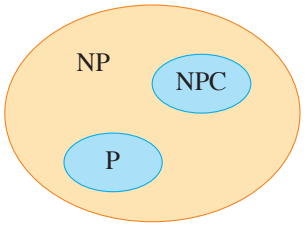


Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $\mathrm{P} \cap \mathrm{NPC} = \emptyset$.

Proof Suppose that $L \in \mathrm{P}$ and also that $L \in \mathrm{NPC}$. For any $L' \in \mathrm{NP}$, we have $L' \leq_{\mathrm{P}} L$ by property 2 of the definition of NP-completeness. Thus, by Lemma 34.3, we also have that $L' \in \mathrm{P}$, which proves the first statement of the theorem.

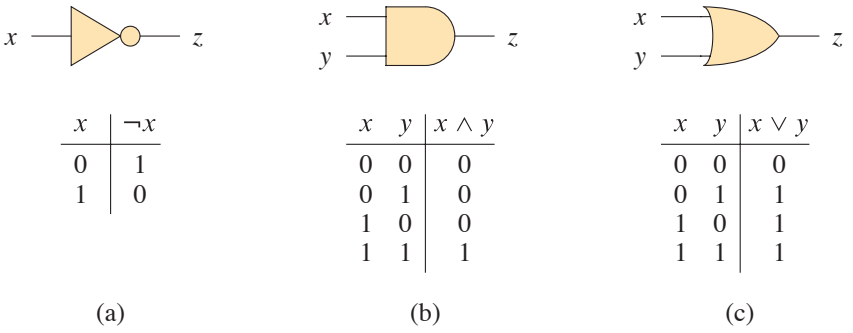
To prove the second statement, consider the contrapositive of the first statement: if $\mathrm{P} \neq \mathrm{NP}$, then there does not exist an NP-complete problem that is polynomial-time solvable. But $\mathrm{P} \neq \mathrm{NP}$ means that there is some problem in NP that is not polynomial-time solvable, and hence the second statement is the contrapositive of the first statement. ■

It is for this reason that research into the $\mathrm{P} \neq \mathrm{NP}$ question centers around the NP-complete problems. Most theoretical computer scientists believe that $\mathrm{P} \neq \mathrm{NP}$, which leads to the relationships among P, NP, and NPC shown in Figure 34.6. For all we know, however, someone may yet come up with a polynomial-time algorithm for an NP-complete problem, thus proving that $\mathrm{P} = \mathrm{NP}$. Nevertheless, since no polynomial-time algorithm for any NP-complete problem has yet been discovered, 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, polynomial-time reducibility becomes 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'll informally describe a proof that relies on a basic understanding of boolean combinational circuits.



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.

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 appearing 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 used 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.

The operation of each gate, or of any boolean combinational element, is defined 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 example, the truth table for the OR gate says that when the inputs are $x = 0$ and $y = 1$, the output value is $z = 1$. The symbol $\neg$ denotes the NOT function, $\wedge$ denotes the AND function, and $\vee$ denotes the OR function. Thus, for example, $0 \vee 1 = 1$.

AND and OR gates are not limited to just 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, so that the output value of the first element becomes 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

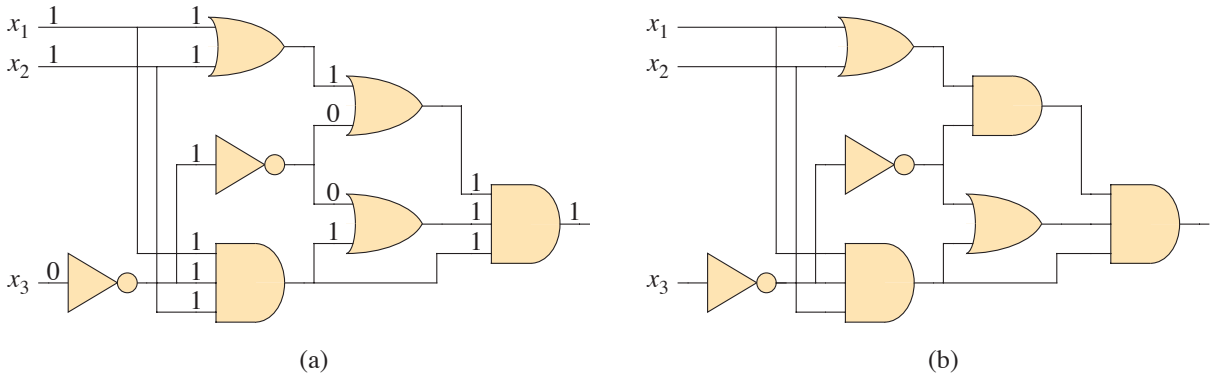


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

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, for a given combinational circuit, imagine 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, where 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.

A ***truth assignment*** for a boolean combinational circuit is a set of boolean input values. We say that a 1-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. Since it always produces 0, 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 graph-like 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 the value in having a polynomial-time algorithm for this problem.

Given a circuit C, you can determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if the circuit has k inputs, then you would have to check up to 2^k possible assignments. When the size of C is polynomial in k, checking all possible assignments to the inputs takes $\Omega(2^k)$ time, which is superpolynomial in the size of the circuit.[10] 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 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 a boolean value to each of the wires in C. (See Exercise 34.3-4 for a smaller certificate.)

The algorithm A works 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, algorithm A outputs 1, since the values assigned to the inputs of C provide a satisfying assignment. Otherwise, A outputs 0.

---

[10] 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.

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, and with a good implementation, linear time suffices. Thus, CIRCUIT-SAT is verifiable 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 *every* language in NP is polynomial-time reducible to CIRCUIT-SAT. The actual proof of this fact is full of technical intricacies, and so instead we'll sketch the proof based on some understanding of the workings of computer hardware.

A computer program is stored in the computer's 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 instruction is to be executed next. The program counter automatically increments when each instruction is fetched, thereby causing the computer to execute instructions sequentially. Certain instructions 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 while a program executes, the computer's memory holds the entire state of the computation. (Consider 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***. When an instruction executes, it transforms the configuration. Think 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'll 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 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

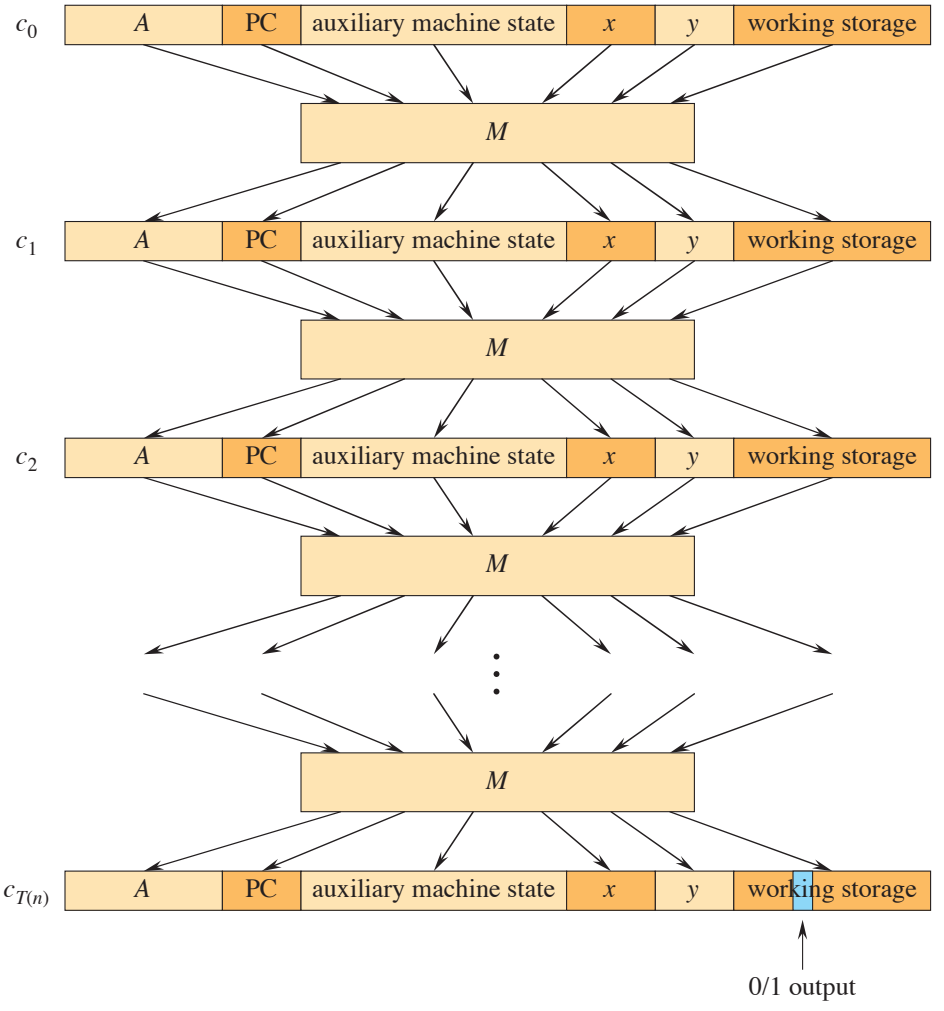


Figure 34.9 The sequence of configurations produced by an algorithm A running on an input x and certificate y. Each configuration represents the state of the computer for one step of the computation and, besides A, x, and y, includes the program counter (PC), auxiliary machine state, and working storage. 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.

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, consider each configuration as com-

prising a few parts: 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. After A halts, the output 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 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, suppose that there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$. Then, upon applying 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 to show that F runs in time polynomial in $n = |x|$. First, the number of bits required to represent a configuration is polynomial in n. Why? 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 certifi-

cate 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 implicitly 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 $O(n^k)$, and 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 $O(n^k)$ 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 an alternative proof of Lemma 34.5 can use a satisfying assignment as a certificate. 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 does the proof 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 cannot 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

The proof that the circuit-satisfiability problem is NP-complete showed directly that $L \leq_P$ CIRCUIT-SAT for every language $L \in$ NP. This section shows how to prove that languages are NP-complete without directly reducing *every* language in NP to the given language. We'll explore examples of this methodology by proving that various formula-satisfiability problems are NP-complete. Section 34.5 provides many more examples.

The following lemma provides a foundation for showing that a given 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, we have $L \in$ NP, then $L \in$ NPC.

Proof Since L' is NP-complete, for all $L'' \in$ NP, we have $L'' \leq_P L'$. By supposition, we have $L' \leq_P L$, 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 provides a method for proving that a language L is NP-complete:

1. Prove $L \in \text{NP}$.
2. Prove that L is NP-hard:
   a. Select a known NP-complete language L'.
   b. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
   c. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
   d. Prove that the algorithm computing f runs in polynomial time.

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 furnishes a starting point. Knowing that the circuit-satisfiability problem is NP-complete makes it much easier to prove that other problems are NP-complete. Moreover, as the catalog of known NP-complete problems grows, so will the choices for languages from which to reduce.

Formula satisfiability

To illustrate the reduction methodology, let's see 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, assume that there are no redundant parentheses, i.e., a formula contains at most one pair of parentheses per boolean connective.)

We can 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, which we can express in formal-language terms as

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable boolean formula}\} .$$

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 $\le_{\text{P}}$ SAT, which by Lemma 34.8 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 can be done in polynomial time. If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable. Thus, SAT belongs to NP.

To prove that SAT is NP-hard, we show that CIRCUIT-SAT $\le_{\text{P}}$ 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

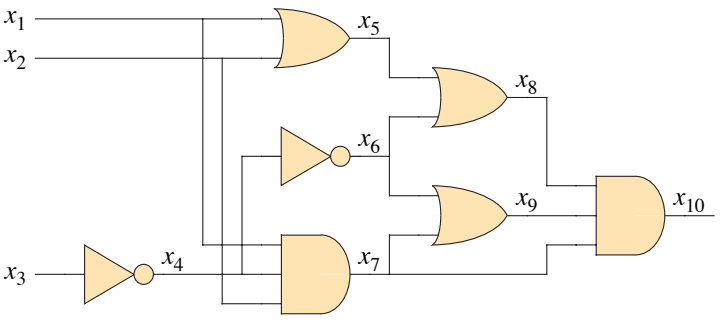


Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit and a clause for each logic gate.

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 to 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 has a variable x_i. To express how each gate operates, construct a small formula involving the variables of its incident wires. The formula has the form of an "if and only if" ($\leftrightarrow$), with the variable for the gate's output on the left and on the right a logical expression encapsulating the gate's function on its inputs. For example, the operation of the output AND gate (the rightmost gate in the figure) 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 wire values are assigned 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 $\leq_P$ SAT, which completes the proof. ■

3-CNF satisfiability

Reducing from formula satisfiability gives us an avenue to prove many problems NP-complete. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases to consider. Instead, it is usually simpler to reduce from a restricted language of boolean formulas. Of course, the restricted language must not be polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

In order to define 3-CNF satisfiability, we first need to define a few terms. A ***literal*** in a boolean formula is an occurrence of a variable (such as x_1) or its negation ($\neg x_1$). A ***clause*** is the OR of one or more literals, such as $x_1 \vee \neg x_2 \vee \neg x_3$. A boolean formula is in ***conjunctive normal form***, or ***CNF***, if it is expressed as an AND of clauses, and it's in ***3-conjunctive normal form***, or ***3-CNF***, if each clause contains 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 $\neg x_2$.

The language 3-CNF-SAT consists of encodings of boolean formulas in 3-CNF that are 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 from 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.

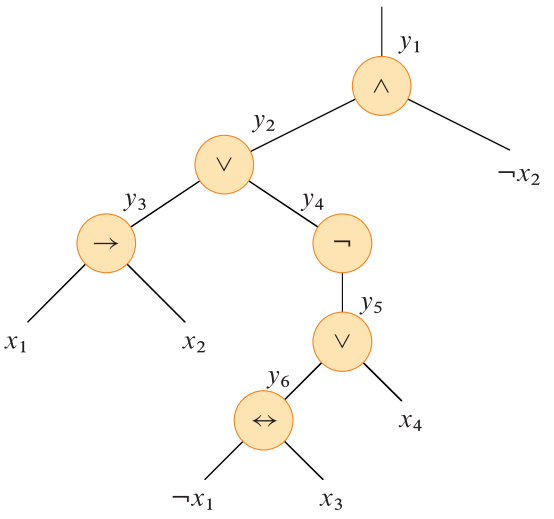


Figure 34.11 The tree corresponding to the formula $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

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, 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

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2 \; . \tag{34.3}$$

If the input formula contains a clause such as the OR of several literals, use associativity to parenthesize the expression fully so that every internal node in the resulting tree has just one or two children. The binary parse tree is like a circuit for computing the function.

Mimicking the reduction in the proof of Theorem 34.9, introduce a variable y_i for the output of each internal node. Then rewrite the original formula ϕ as the AND of the variable at the root of the parse tree and a conjunction of clauses describing the operation of each node. For the formula (34.3), the resulting expression is

$$\begin{aligned}
\phi' \; = \; & y_1 \\
& \wedge \; (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
& \wedge \; (y_2 \leftrightarrow (y_3 \vee y_4)) \\
& \wedge \; (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
& \wedge \; (y_4 \leftrightarrow \neg y_5) \\
& \wedge \; (y_5 \leftrightarrow (y_6 \vee x_4)) \\
& \wedge \; (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) \; .
\end{aligned}$$

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 $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

The formula ϕ' thus obtained is a conjunction of clauses ϕ'_i, each of which has at most three literals. These clauses are not yet ORs of three literals.

The second step of the reduction converts each clause ϕ'_i into conjunctive normal form. 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, build a formula in ***disjunctive normal form*** (or ***DNF***)—an OR of ANDs—that is equivalent to $\neg\phi'_i$. Then negate this formula and convert it into a CNF formula ϕ''_i by using ***DeMorgan's laws*** for propositional logic,

$$\neg(a \wedge b) = \neg a \vee \neg b ,$$
$$\neg(a \vee b) = \neg a \wedge \neg b ,$$

to complement all literals, change ORs into ANDs, and change ANDs into ORs.

In our example, the clause $\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ converts 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 yields 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, each clause ϕ'_i of the formula ϕ' has been converted 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 three literals.

The third and final step of the reduction further transforms the formula so that each clause has *exactly* three distinct literals. From the clauses of the CNF formula ϕ'', construct the final 3-CNF formula ϕ'''. This formula also uses two auxiliary variables, p and q. For each clause C_i of ϕ'', include the following clauses in ϕ''':

- If C_i contains three distinct literals, then simply include C_i as a clause of ϕ'''.
- If C_i contains exactly two 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''' contain exactly three 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 contains just one 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 three 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'. Then 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 one variable and one clause per connective in ϕ. Constructing ϕ'' from ϕ' can introduce at most eight clauses into ϕ'' for each clause from ϕ', since each clause of ϕ' contains at most three variables, and the truth table for each clause has at most $2^3 = 8$ rows. The construction of ϕ''' from ϕ'' introduces at most four 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 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 upon using 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
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 two 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. This section uses the reduction methodology to provide NP-completeness proofs for a variety of problems drawn from graph theory and set partitioning.

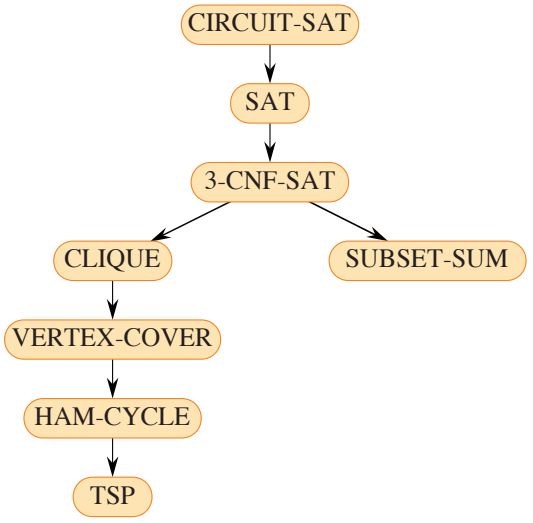


Figure 34.13 The structure of NP-completeness proofs in Sections 34.4 and 34.5. All proofs ultimately follow by reduction from the NP-completeness of CIRCUIT-SAT.

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. This section concludes with a recap of reduction strategies.

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 a graph. The corresponding decision problem asks 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 contains a clique of size k lists all k-subsets of V and checks 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 First, we show that CLIQUE $\in$ NP. For a given graph $G = (V, E)$, use the set $V' \subseteq V$ of vertices in the clique as a certificate for G. To check whether V' is a clique in polynomial time, check whether, for each pair $u, v \in V'$, the edge (u, v) belongs to E.

We next prove that 3-CNF-SAT $\leq_P$ CLIQUE, which shows that the clique problem is NP-hard. You might be surprised that the proof reduces an instance of 3-CNF-SAT to an instance of CLIQUE, 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 $r = 1, 2, \ldots, k$, each clause C_r contains exactly three distinct literals: l_1^r, l_2^r, and l_3^r. We will construct a graph G such that ϕ is satisfiable if and only if G contains a clique of size k.

We construct the undirected graph $G = (V, E)$ as follows. For each clause $C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, place a triple of vertices v_1^r, v_2^r, and v_3^r into V. Add edge (v_i^r, v_j^s) into E 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 build this graph from ϕ in polynomial time. As an example of this construction, if

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) ,$$

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 contains 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. If $v_i^r \in V'$, then assign 1 to the corresponding literal l_i^r. Since G contains no edges between inconsistent literals, no literal and its complement are both assigned 1. 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.) ■

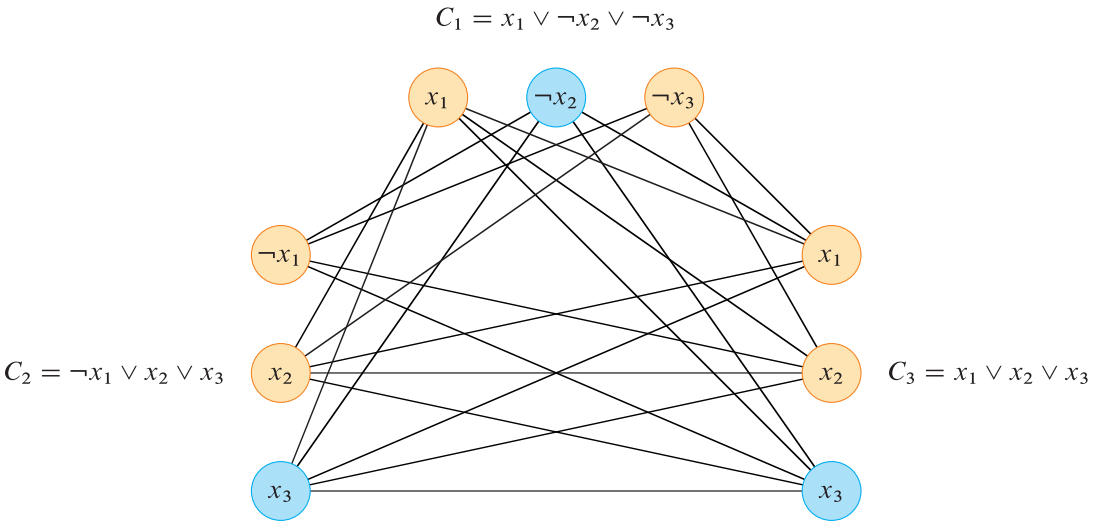


Figure 34.14 The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 set to 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 blue vertices.

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 $\neg x_1$, this satisfying assignment can set x_1 to either 0 or 1.

The proof of Theorem 34.11 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 there were a polynomial-time algorithm that solves 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—does not suffice, however. Why not? Perhaps the instances of 3-CNF-SAT that we choose to reduce from are "easy," and so we would not have reduced an NP-hard problem to CLIQUE.

Moreover, the reduction uses the instance of 3-CNF-SAT, but not the solution. We would have erred if the polynomial-time reduction had relied on knowing

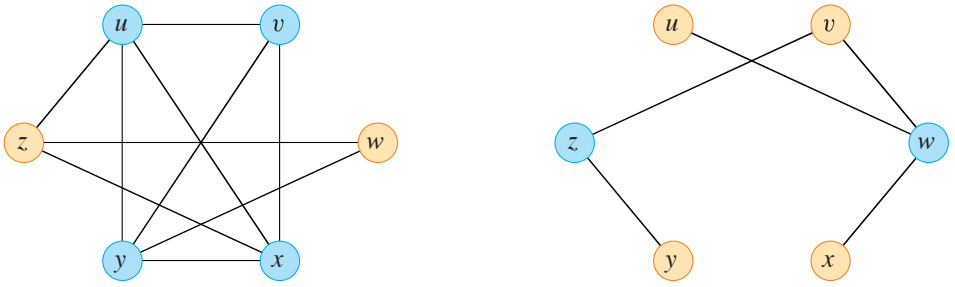


Figure 34.15 Reducing CLIQUE to VERTEX-COVER. **(a)** An undirected graph $G = (V, E)$ with clique $V' = \{u, v, x, y\}$, shown in blue. **(b)** The graph $\overline{G}$ produced by the reduction algorithm that has vertex cover $V - V' = \{w, z\}$, in blue.

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. For this optimization problem, the corresponding decision problem asks 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. Given a graph $G = (V, E)$ and an integer k, the certificate 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 $v \in V'$. It is easy to verify the certificate in polynomial time.

To prove that the vertex-cover problem is NP-hard, we reduce from the clique problem, 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 a graph $\overline{G} = (V, \overline{E})$, 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 and computes the complement $\overline{G}$ 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 contains a clique of size k if and only if the graph $\overline{G}$ has a vertex cover of size $|V| - k$.

Suppose that G contains 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 belongs to $V - V'$, 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, you shouldn't give up hope just because a problem is NP-complete. You might 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.

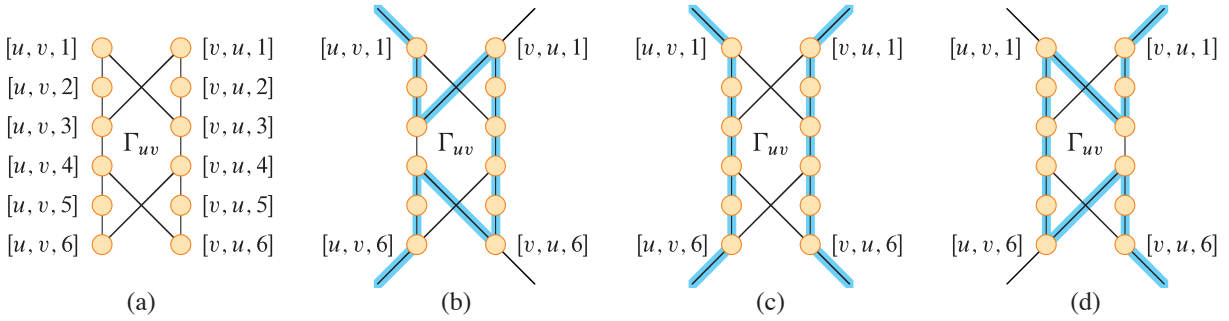


Figure 34.16 The gadget used in reducing the vertex-cover problem to the hamiltonian-cycle problem. An edge (u, v) of graph G corresponds to gadget Γ_{uv} in the graph G' created in the reduction. **(a)** The gadget, with individual vertices labeled. **(b)–(d)** The paths highlighted in blue are the only possible ones through the gadget that include all vertices, assuming that the only connections from the gadget to the remainder of G' are through vertices $[u, v, 1]$, $[u, v, 6]$, $[v, u, 1]$, and $[v, u, 6]$.

Proof We first show that HAM-CYCLE $\in$ NP. Given an undirected graph $G = (V, E)$, the 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. This certificate can be verified 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 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. We assume without loss of generality that G contains no isolated vertices (that is, every vertex in V has at least one incident edge) and that $k \leq |V|$. (If an isolated vertex belongs to a vertex cover of size k, then there also exists a vertex cover of size $k-1$, and for any graph, the entire set V is always a vertex cover.)

Our construction uses a ***gadget***, which is a piece of a graph that enforces certain properties. Figure 34.16(a) shows the gadget we use. For each edge $(u, v) \in E$, the constructed graph G' contains one copy of this gadget, which we denote by Γ_{uv}. We denote each vertex in Γ_{uv} by $[u, v, i]$ or $[v, u, i]$, where $1 \leq i \leq 6$, so that each gadget Γ_{uv} contains 12 vertices. Gadget Γ_{uv} also contains the 14 edges shown in Figure 34.16(a).

Along with the internal structure of the gadget, we enforce the properties we want by limiting the connections between the gadget 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 Γ_{uv}. Any hamiltonian cycle

of G' must traverse the edges of Γ_{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 gadget'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 gadget 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 either it visits all 12 of the gadget's vertices (Figure 34.16(d)) or it visits the six vertices $[v, u, 1]$ through $[v, u, 6]$ and reenters to visit $[u, v, 1]$ through $[u, v, 6]$ (Figure 34.16(c)). No other paths through the gadget 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 gadget's vertices.

The only other vertices in V' other than those of gadgets are ***selector vertices*** $s_1, s_2, \ldots, s_k$. We'll use edges incident on selector vertices in G' to select the k vertices of the cover in G.

In addition to the edges in gadgets, E' contains two other types of edges, which Figure 34.17 shows. First, for each vertex $u \in V$, edges join pairs of gadgets in order to form a path containing all gadgets 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. To create a path in G' through all the gadgets corresponding to edges incident on u, E' contains 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 $\langle x, y, z \rangle$, and so graph G' in part (b) of the figure includes the edges $([w, x, 6], [w, y, 1])$ and $([w, y, 6], [w, z, 1])$. The vertices adjacent to x are ordered as $\langle w, y \rangle$, so that G' includes the edge $([x, w, 6], [x, y, 1])$. For each vertex $u \in V$, these edges in G' fill in a path containing all gadgets corresponding to edges incident on u in G.

The intuition behind these edges is that if vertex $u \in V$ belongs to the vertex cover of G, then G' contains a path from $[u, u^{(1)}, 1]$ to $[u, u^{(\text{degree}(u))}, 6]$ that "covers" all gadgets corresponding to edges incident on u. That is, for each of these gadgets, say $\Gamma_{u,u^{(i)}}$, the path either includes all 12 vertices (if u belongs to the vertex cover but $u^{(i)}$ does not) or just the six vertices $[u, u^{(i)}, 1]$ through $[u, u^{(i)}, 6]$ (if both u and $u^{(i)}$ belong to 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, E' includes the edges

$$\{(s_j, [u, u^{(1)}, 1]) : u \in V \text{ and } 1 \le j \le k\} \\ \cup \{(s_j, [u, u^{(\text{degree}(u))}, 6]) : u \in V \text{ and } 1 \le j \le k\} .$$

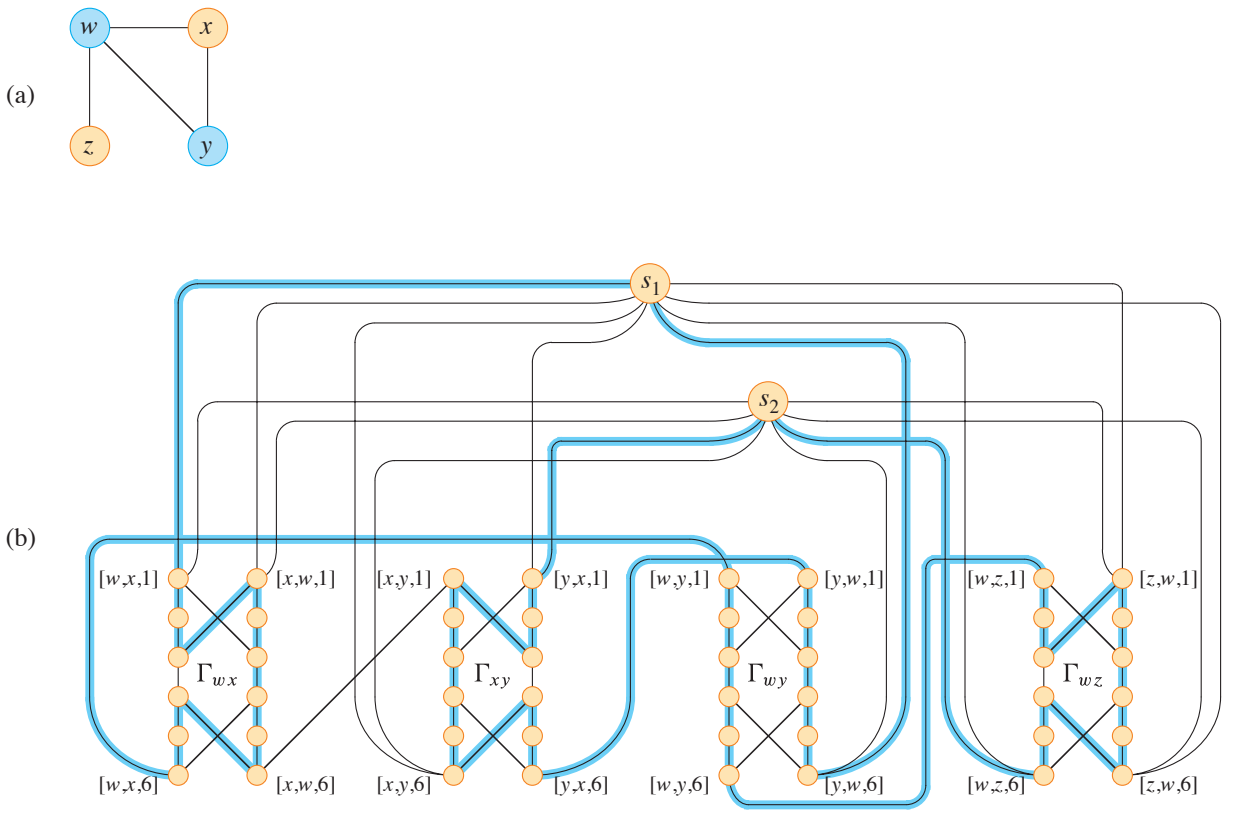


Figure 34.17 Reducing an instance of the vertex-cover problem to an instance of the hamiltonian-cycle problem. **(a)** An undirected graph G with a vertex cover of size 2, consisting of the blue vertices w and y. **(b)** The undirected graph G' produced by the reduction, with the hamiltonian cycle corresponding to the vertex cover highlighted in blue. The vertex cover $\{w, y\}$ corresponds to edges $(s_1, [w, x, 1])$ and $(s_2, [y, x, 1])$ appearing in the hamiltonian cycle.

Next we show that the size of G' is polynomial in the size of G, and hence it takes time polynomial in the size of G to construct G'. The vertices of G' are those in the gadgets, plus the selector vertices. With 12 vertices per gadget, plus $k \le |V|$ selector vertices, G' contains a total of

$$\begin{aligned} |V'| &= 12\,|E| + k \\ &\le 12\,|E| + |V| \end{aligned}$$

vertices. The edges of G' are those in the gadgets, those that go between gadgets, and those connecting selector vertices to gadgets. Each gadget contains 14 edges, totaling $14\,|E|$ in all gadgets. For each vertex $u \in V$, graph G' has degree$(u) - 1$ edges going between gadgets, so that summed over all vertices in V,

$$\sum_{u \in V} (\text{degree}(u) - 1) = 2\,|E| - |V|$$

edges go between gadgets. 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 $V^* \subseteq V$, where $|V^*| = k$. Let $V^* = \{u_1, u_2, \ldots, u_k\}$. As Figure 34.17 shows, we can construct a hamiltonian cycle in G' by including the following edges[11] for each vertex $u_j \in V^*$. Start by including 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 gadgets corresponding to edges incident on u_j. Also include the edges within these gadgets 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

$$\begin{aligned} &\{(s_j, [u_j, u_j^{(1)}, 1]) : 1 \le j \le k\} \\ &\quad \cup \{(s_{j+1}, [u_j, u_j^{(\text{degree}(u_j))}, 6]) : 1 \le j \le k - 1\} \\ &\quad \cup \{(s_1, [u_k, u_k^{(\text{degree}(u_k))}, 6])\} \ . \end{aligned}$$

By inspecting Figure 34.17, you can verify that these edges form a cycle, where $u_1 = w$ and $u_2 = y$. The cycle starts at s_1, visits all gadgets corresponding to edges incident on u_1, then visits s_2, visits all gadgets corresponding to edges incident on u_2, and so on, until it returns to s_1. The cycle visits each gadget 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 gadget of G'. Because the cycle also visits every selector vertex, it is hamiltonian.

Conversely, suppose that $G' = (V', E')$ contains a hamiltonian cycle $C \subseteq E'$. We claim that the set

$$V^* = \{u \in V : (s_j, [u, u^{(1)}, 1]) \in C \text{ for some } 1 \le j \le k\} \tag{34.4}$$

---

[11] Technically, a cycle is defined as a sequence of vertices rather than edges (see Section B.4). In the interest of clarity, we abuse notation here and define the hamiltonian cycle by its edges.

is a vertex cover for G.

We first argue that the set V^* is well defined, that is, for each selector vertex s_j, exactly one of the incident edges in the hamiltonian cycle C is of the form $(s_j, [u, u^{(1)}, 1])$ for some vertex $u \in V$. To see why, partition the hamiltonian cycle C into maximal paths that start at some selector vertex s_i, visit one or more gadgets, and end at some selector vertex s_j without passing through any other selector vertex. Let's call each of these maximal paths a "cover path." Let P be one such cover path, and orient it going from s_i to s_j. If P contains the edge $(s_i, [u, u^{(1)}, 1])$ for some vertex $u \in V$, then we have shown that one edge incident on s_i has the required form. Assume, then, that P contains the edge $(s_i, [v, v^{(\text{degree}(v))}, 6])$ for some vertex $v \in V$. This path enters a gadget from the bottom, as drawn in Figures 34.16 and 34.17, and it leaves from the top. It might go through several gadgets, but it always enters from the bottom of a gadget and leaves from the top. The only edges incident on vertices at the top of a gadget either go to the bottoms of other gadgets or to selector vertices. Therefore, after the last gadget in the series of gadgets visited by P, the edge taken must go to a selector vertex s_j, so that P contains an edge of the form $(s_j, [u, u^{(1)}, 1])$, where $[u, u^{(1)}, 1]$ is a vertex at the top of some gadget. To see that not both edges incident on s_j have this form, simply reverse the direction of traversing P in the above argument.

Having established that the set V^* is well defined, let's see why it is a vertex cover for G. We have already established that each cover path starts at some s_i, takes the edge $(s_i, [u, u^{(1)}, 1])$ for some vertex $u \in V$, passes through all the gadgets corresponding to edges in E incident on u, and then ends at some selector vertex s_j. (This orientation is the reverse of the orientation in the paragraph above.) Let's call this cover path P_u, and by equation (34.4), the vertex cover V^* includes u. Each gadget visited by P_u must be Γ_{uv} or Γ_{vu} for some $v \in V$. For each gadget 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 gadget, 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. Because each vertex in each gadget 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-salesperson problem

In the ***traveling-salesperson problem***, which is closely related to the hamiltonian-cycle problem, a salesperson must visit n cities. Let's model the problem as a complete graph with n vertices, so that the salesperson wishes to make a ***tour***, or hamiltonian cycle, visiting each city exactly once and finishing at the starting city. The salesperson incurs a nonnegative integer cost $c(i, j)$ to travel from city i

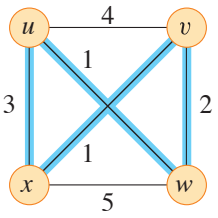


Figure 34.18 An instance of the traveling-salesperson problem. Edges highlighted in blue represent a minimum-cost tour, with cost 7.

to city j. In the optimization version of the problem, the salesperson 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

$$\begin{aligned}\text{TSP} = \{\langle G, c, k\rangle : \ & G = (V, E) \text{ is a complete graph,} \\ & c \text{ is a function from } V \times V \to \mathbb{N}, \\ & k \in \mathbb{N}, \text{ and} \\ & G \text{ has a traveling-salesperson tour with cost at most } k\}\ .\end{aligned}$$

The following theorem shows that a fast algorithm for the traveling-salesperson problem is unlikely to exist.

Theorem 34.14
The traveling-salesperson problem is NP-complete.

Proof We first show that TSP $\in$ NP. Given an instance of the problem, the certificate is 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 that 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 $\le_{\text{P}}$ TSP. Given an instance $G = (V, E)$ of HAM-CYCLE, construct an instance of TSP by forming the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \ne j\}$, with the cost function c defined as

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E\ , \\ 1 & \text{if } (i, j) \notin E\ . \end{cases}$$

(Because G is undirected, it contains 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 can be created 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. The ***subset-sum problem*** takes as inputs a finite set S of positive integers and an integer ***target*** $t > 0$. It asks whether there exists a subset $S' \subseteq S$ whose elements sum to exactly 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 express the problem as a language:

$$\text{SUBSET-SUM} = \left\{\langle S, t\rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\right\} .$$

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 $\in$ NP, for an instance $\langle S, t\rangle$ of the problem, 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. The numbers will be represented in base 10, with each number containing $n + k$ digits and each digit corresponding to either one variable

		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 shaded blue, and it contains v'_1, v'_2, and v_3, corresponding to the satisfying assignment. Subset S' 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.

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. 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 $\ell \neq i$, 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.

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 the s_j and s'_j values). Interpreting these numbers in base 10, therefore, no carries can occur from lower digits to higher digits.[12]

The reduction can be performed in polynomial time. The set S consists of $2n + 2k$ 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.

Let's 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, S' includes exactly the v_i and v'_i values that correspond 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, one, two, or three 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.

---

[12] In fact, any base $b \geq 7$ works. 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.

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. To achieve the target of 4 in each digit labeled by clause C_j, include 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 S' matches the target in all digits of the sum, and no carries can occur, the values of S' sum to t.

Now suppose that some subset $S' \subseteq S$ 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 $v_i \in S'$, then set $x_i = 1$. Otherwise, $v'_i \in S'$, and 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 $x_i = 1$ 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 $x_i = 0$ when $v'_i \in S'$, clause C_j is again satisfied. Thus, all clauses of ϕ are satisfied, which completes the proof. ■

34.5.6 Reduction strategies

From the reductions in this section, you can see that no single strategy applies to all NP-complete problems. Some reductions are straightforward, such as reducing the hamiltonian-cycle problem to the traveling-salesperson problem. Others are considerably more complicated. Here are a few things to keep in mind and some strategies that you can often bring to bear.

Pitfalls

Make sure that you don't get the reduction backward. That is, in trying to show that problem Y is NP-complete, you might take a known NP-complete problem X and give a polynomial-time reduction from Y to X. That is the wrong direction. The reduction should be from X to Y, so that a solution to Y gives a solution to X.

Remember also that reducing a known NP-complete problem X to a problem Y does not in itself prove that Y is NP-complete. It proves that Y is NP-hard. In order to show that Y is NP-complete, you additionally need to prove that it's in NP by showing how to verify a certificate for Y in polynomial time.

Go from general to specific

When reducing problem X to problem Y, you always have to start with an arbitrary input to problem X. But you are allowed to restrict the input to problem Y as much as you like. For example, when reducing 3-CNF satisfiability to the subset-sum problem, the reduction had to be able to handle *any* 3-CNF formula as its input, but the input to the subset-sum problem that it produced had a particular structure: $2n + 2k$ integers in the set, and each integer was formed in a particular way. The reduction did not need to produce *every* possible input to the subset-sum problem. The point is that one way to solve the 3-CNF satisfiability problem transforms the input into an input to the subset-sum problem and then uses the answer to the subset-sum problem as the answer to the 3-CNF satisfiability problem.

Take advantage of structure in the problem you are reducing from

When you are choosing a problem to reduce from, you might consider two problems in the same domain, but one problem has more structure than the other. For example, it's almost always much easier to reduce from 3-CNF satisfiability than to reduce from formula satisfiability. Boolean formulas can be arbitrarily complicated, but you can exploit the structure of 3-CNF formulas when reducing.

Likewise, it is usually more straightforward to reduce from the hamiltonian-cycle problem than from the traveling-salesperson problem, even though they are so similar. That's because you can view the hamiltonian-cycle problem as taking a complete graph but with edge weights of just 0 or 1, as they would appear in the adjacency matrix. In that sense, the hamiltonian-cycle problem has more structure than the traveling-salesperson problem, in which edge weights are unrestricted.

Look for special cases

Several NP-complete problems are just special cases of other NP-complete problems. For example, consider the decision version of the 0-1 knapsack problem: given a set of n items, each with a weight and a value, does there exist a subset of items whose total weight is at most a given weight W and whose total value is at least a given value V? You can view the set-partition problem in Exercise 34.5-5 as a special case of the 0-1 knapsack problem: let the value of each item equal its weight, and set both W and V to half the total weight. If problem X is NP-hard and it is a special case of problem Y, then problem Y must be NP-hard as well. That is because a polynomial-time solution for problem Y automatically gives a polynomial-time solution for problem X. More intuitively, problem Y, being more general than problem X, is at least as hard.

Select an appropriate problem to reduce from

It's often a good strategy to reduce from a problem in a domain that is the same as, or at least related to, the domain of the problem that you're trying to prove NP-complete. For example, we saw that the vertex-cover problem—a graph problem—was NP-hard by reducing from the clique problem—also a graph problem. From the vertex-cover problem, we reduced to the hamiltonian-cycle problem, and from the hamiltonian-cycle problem, we reduced to the traveling-salesperson problem. All of these problems take undirected graphs as inputs.

Sometimes, however, you will find that is it better to cross over from one domain to another, such as when we reduced from 3-CNF satisfiability to the clique problem or to the subset-sum problem. 3-CNF satisfiability often turns out to be a good choice as a problem to reduce from when crossing domains.

Within graph problems, if you need to select a portion of the graph, without regard to ordering, then the vertex-cover problem is often a good place to start. If ordering matters, then consider starting from the hamiltonian-cycle or hamiltonian-path problem (see Exercise 34.5-6).

Make big rewards and big penalties

The strategy for reducing the hamiltonian-cycle problem with a graph G to the traveling-salesperson problem encouraged using edges present in G when choosing edges for the traveling-salesperson tour. The reduction did so by giving these edges a low weight: 0. In other words, we gave a big reward for using these edges.

Alternatively, the reduction could have given the edges in G a finite weight and given edges not in G infinite weight, thereby exacting a hefty penalty for using edges not in G. With this approach, if each edge in G has weight W, then the target weight of the traveling-salesperson tour becomes $W \cdot |V|$. You can sometimes think of the penalties as a way to enforce requirements. For example, if the traveling-salesperson tour includes an edge with infinite weight, then it violates the requirement that the tour should include only edges belonging to G.

Design gadgets

The reduction from the vertex-cover problem to the hamiltonian-cycle problem uses the gadget shown in Figure 34.16. This gadget is a subgraph that is connected to other parts of the constructed graph in order to restrict the ways that a cycle can visit each vertex in the gadget once. More generally, a gadget is a component that enforces certain properties. Gadgets can be complicated, as in the reduction to the hamiltonian-cycle problem. Or they can be simple: in the reduction of 3-CNF satisfiability to the subset-sum problem, you can view the slack variables s_j and s'_j

as gadgets enabling each clause-labeled digit position to achieve the target value of 4.

Exercises

34.5-1

The ***subgraph-isomorphism problem*** takes two undirected graphs G_1 and G_2, and 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 elements 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, the input is a 3-CNF formula ϕ with n variables and m clauses, where m is even. The question is 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.

34.5-9

The proof that VERTEX-COVER $\leq_P$ HAM-CYCLE assumes that the graph G given as input to the vertex-cover problem has no isolated vertices. Show how the reduction in the proof can break down if G has an isolated vertex.

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. 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:* First prove that in a bipartite graph, the size of the maximum independent set plus the size of the maximum matching is equal to $|V|$. Then use a maximum-matching algorithm (see Section 25.1) as a first step in an algorithm to find an independent set.)

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 to divide the money or prove that the problem of dividing the money in the manner described 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 two 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 exact power of 2, so that 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, subject to the restriction that if two countries share a border, they must have different colors. You 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.

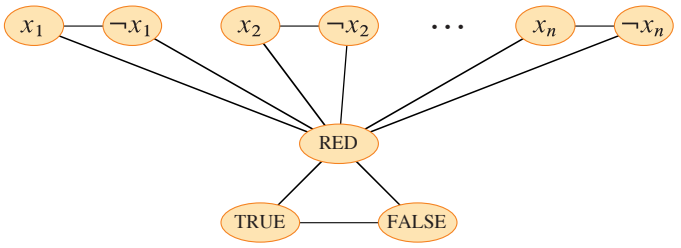


Figure 34.20 The subgraph of G in Problem 34-3 formed by the literal edges. The special vertices TRUE, FALSE, and RED form a triangle, and for each variable x_i, the vertices x_i, $\neg x_i$, and RED form a triangle.

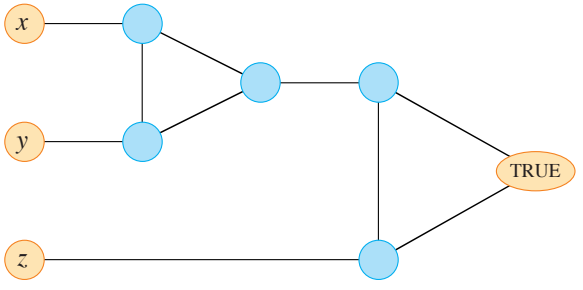


Figure 34.21 The gadget corresponding to a clause $(x \vee y \vee z)$, used in Problem 34-3.

To prove that 3-COLOR is NP-complete, you can reduce from 3-CNF-SAT. Given a formula ϕ of m clauses on n variables $x_1, x_2, \ldots, x_n$, 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, five vertices for each clause, and three 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. As Figure 34.20 shows, the literal edges form a triangle on the three special vertices TRUE, FALSE, and RED, and they also form a triangle on x_i, $\neg x_i$, and RED for $i = 1, 2, \ldots, n$.

d. Consider a graph containing the literal edges. Argue that in any 3-coloring c of such a graph, exactly one of a variable and its negation is colored $c(\text{TRUE})$ and the other is colored $c(\text{FALSE})$. Then argue that for any truth assignment for ϕ, there exists a 3-coloring of the graph containing just the literal edges.

The gadget shown in Figure 34.21 helps to enforce the condition corresponding to a clause $(x \vee y \vee z)$, where x, y, and z are literals. Each clause requires a unique copy of the five blue vertices 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 gadget 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.

34-4 Scheduling with profits and deadlines

You have one computer and a set of n tasks $\{a_1, a_2, \ldots, a_n\}$ requiring time on the computer. Each task a_j requires t_j time units on the computer (its processing time), yields a profit of p_j, and has a deadline d_j. The computer can process only one task at a time, and task a_j must run without interruption for t_j consecutive time units. If task a_j completes by its deadline d_j, you receive a profit p_j. If instead task a_j completes after its deadline, you receive no profit. As an optimization problem, given the processing times, profits, and deadlines for a set of n tasks, you 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 [176] 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. Fortnow's book [152] gives a history of NP-completeness, along with societal implications. Hopcroft, Motwani, and Ullman [225], Lewis and Papadimitriou [299], Papadimitriou [352], and Sipser [413] 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 [107], Johnsonbaugh and

Schaefer [239], and Kleinberg and Tardos [257]. The book by Hromkovič [229] studies various methods for solving hard problems.

The class P was introduced in 1964 by Cobham [96] and, independently, in 1965 by Edmonds [130], who also introduced the class NP and conjectured that $P \neq NP$. The notion of NP-completeness was proposed in 1971 by Cook [100], who gave the first NP-completeness proofs for formula satisfiability and 3-CNF satisfiability. Levin [297] independently discovered the notion, giving an NP-completeness proof for a tiling problem. Karp [248] 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.

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-salesperson problem with the triangle inequality, and many others, computing good approximate solutions (see Chapter 35) is NP-hard and hence no easier than computing optimal solutions. An introduction to this area can be found in Arora's thesis [21], a chapter by Arora and Lund in Hochbaum [221], a survey article by Arora [22], a book edited by Mayr, Prömel, and Steger [319], a survey article by Johnson [237], and a chapter in the textbook by Arora and Barak [24].

35 Approximation Algorithms

Many problems of practical significance are NP-complete, yet they are too important to abandon merely because nobody knows how to find an optimal solution in polynomial time. Even if a problem is NP-complete, there may be hope. You have at least three options to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time might be fast enough. Second, you might be able to isolate important special cases that you can solve in polynomial time. Third, you can try to devise an approach to find a *near-optimal* solution 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 you are working on an optimization problem in which each potential solution has a positive cost, and you want to find a near-optimal solution. Depending on the problem, you could define an optimal solution as one with maximum possible cost or as one with minimum possible cost, which is to say that the problem might 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 approximation ratio and $\rho(n)$-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 know of 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 polynomial-time approximation algorithms can achieve increasingly better approximation ratios by using more and more computation time. For such problems, you 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 gives a fully polynomial-time approximation scheme. We begin in Section 35.1 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 looks at a version of the traveling-salesperson problem in which the cost function satisfies the triangle inequality and presents an approximation algorithm with an approximation ratio of 2. The section also shows that without the triangle inequality, for any constant $\rho \geq 1$, a ρ-approximation algorithm cannot exist unless P $=$ NP. Section 35.3 applies 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 uses randomization and linear programming to develop two more approximation algorithms. The section first defines the optimization version of 3-CNF satisfiability and gives a simple randomized algorithm that produces a solution with an expected approximation ratio of $8/7$. Then Section 35.4 examines a weighted variant of the vertex-cover problem and exhibits 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 nobody knows how to find an optimal vertex cover in a graph G in polynomial time, there is an efficient algorithm to find a vertex cover that is near-optimal. The approximation algorithm APPROX-VERTEX-COVER on the facing page 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 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 **while** loop of lines 3–6 repeatedly picks an edge (u, v) from E', adds

```
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' edge (u, v) and every edge incident on either u or v
7  return C
```

its endpoints u and v into C, and deletes all edges in E' that u or v covers. 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^*, meaning that for every vertex in C^*, there is at most one edge in A, giving 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) yields

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| \ , \end{aligned}$$

thereby proving the theorem. ■

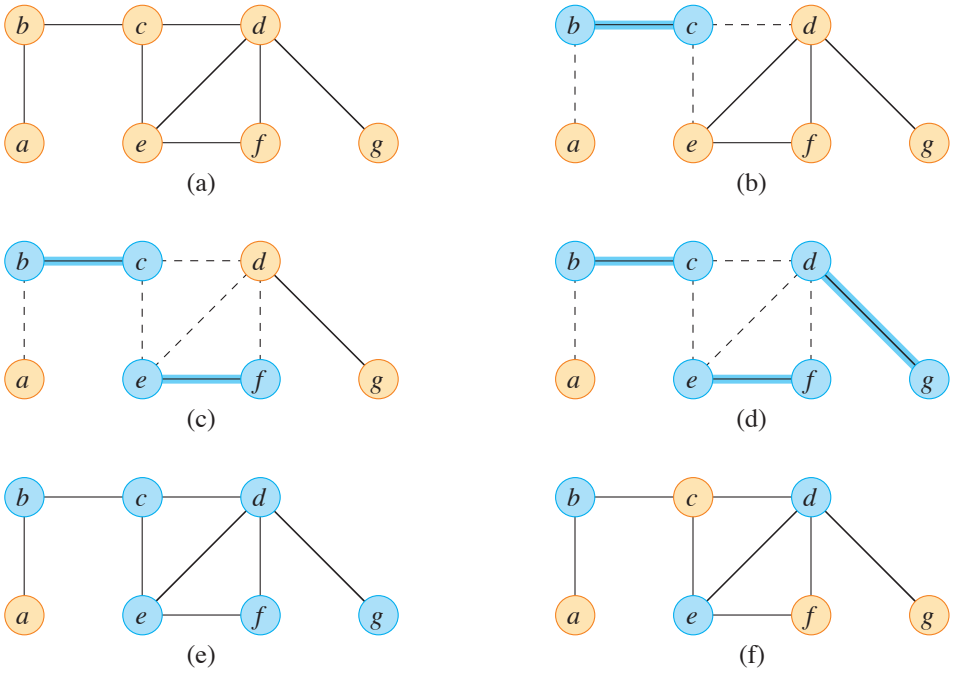


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The highlighted edge (b, c) is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, in blue, are added to the set C containing the vertex cover being created. Dashed edges (a, b), (c, e), and (c, d) are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen, and vertices e and f are added to C. **(d)** Edge (d, g) is chosen, and 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.

Let us reflect on this proof. At first, you might wonder how you 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 you don't even know the size of an optimal vertex cover. Instead of requiring that you know the exact size of an optimal vertex cover, you find 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 to which no edges can be added and still have a 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. The approximation ratio comes from relating the size of the solution returned to the lower bound. 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***
Consider 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 this heuristic does not provide 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
The proof of Theorem 34.12 on page 1084 illustrates 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-salesperson problem

The input to the traveling-salesperson problem, introduced in Section 34.5.4, is a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$. The goal is to 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. Such a cost function c satisfies the ***triangle inequality***: for all vertices $u, v, w \in V$,

$$c(u, w) \leq c(u, v) + c(v, w) .$$

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-salesperson problem is NP-complete even if you require the cost function to satisfy the triangle inequality. Thus, you should not expect to find a polynomial-time algorithm for solving this problem exactly. Your time would be better spent looking for good approximation algorithms.

In Section 35.2.1, we examine a 2-approximation algorithm for the traveling-salesperson 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-salesperson problem with the triangle inequality

Applying the methodology of the previous section, start by computing a structure —a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesperson tour. 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 procedure APPROX-TSP-TOUR on the next page implements this approach, calling the minimum-spanning-tree algorithm MST-PRIM on page 596 as a subroutine. The parameter G is a complete undirected graph, and the cost function c satisfies the triangle inequality.

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.

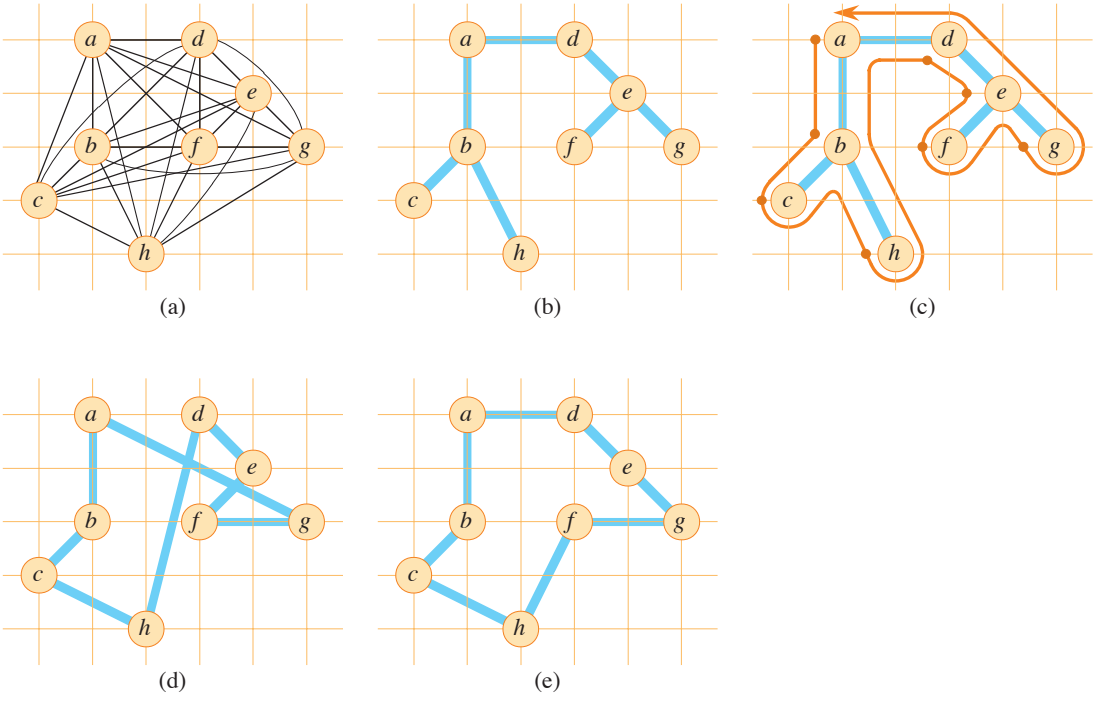


Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning 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.

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

By Exercise 21.2-2, 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-salesperson problem satisfies the triangle inequality, then APPROX-TSP-TOUR returns a tour whose cost is at most twice the cost of an optimal tour.

Theorem 35.2
When the triangle inequality holds, APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesperson problem.

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. Deleting any edge from a tour yields a spanning tree, 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:

$$c(T) \le c(H^*) . \tag{35.4}$$

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's 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, by extending the definition of the cost c in the natural manner to handle multisets of edges, we have

$$c(W) = 2c(T) . \tag{35.5}$$

Inequality (35.4) and equation (35.5) imply that

$$c(W) \le 2c(H^*) , \tag{35.6}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.

Of course, the full walk W is not a tour, since it visits some vertices more than once. By the triangle inequality, however, deleting a visit to any vertex from W does not increase the cost. (When a vertex v is deleted from W between visits to u and w, the resulting ordering specifies going directly from u to w.) Repeatedly apply this operation on each visit to a vertex after the first time it's visited in W, so that W is left with only the first visit to each vertex. In our example, this process 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. ■

Despite the small 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-salesperson problem

When the cost function c does not satisfy the triangle inequality, there is no way to find good approximate tours in polynomial time unless P = NP.

Theorem 35.3
If P $\neq$ NP, then for any constant $\rho \ge 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesperson 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, assume that ρ is an integer, by rounding it up if necessary. We will 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 on page 1085 says that the hamiltonian-cycle problem is NP-complete, Theorem 34.4 on page 1063 implies that if it has a polynomial-time algorithm, then P = NP.

Let $G = (V, E)$ be an instance of the hamiltonian-cycle problem. We will show how to determine efficiently whether G contains a hamiltonian cycle by making use of the hypothesized approximation algorithm A. Convert G into an instance of the traveling-salesperson problem as follows. Let $G' = (V, E')$ be the complete graph on V, that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\} .$$

Assign an integer cost to each edge in E' as follows:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E , \\ \rho |V| + 1 & \text{otherwise .} \end{cases}$$

Given a representation of G, it takes time polynomial in $|V|$ and $|E|$ to create representations of G' and c.

Now consider the traveling-salesperson 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.

What happens upon applying the approximation algorithm A to the traveling-salesperson 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, using A solves the hamiltonian-cycle problem in polynomial time. ■

The proof of Theorem 35.3 serves as an example of a general technique to prove that no good approximation algorithm exists for a particular problem. Given an NP-hard decision problem X, 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. This technique shows that, unless P = NP, there is no polynomial-time ρ-approximation algorithm for problem Y.

Exercises

35.2-1

Let $G = (V, E)$ be a complete undirected graph containing at least 3 vertices, and let c be a cost function 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 to transform one instance of the traveling-salesperson 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-salesperson 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

A solution to the ***bottleneck traveling-salesperson problem*** is 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 how to visit all the nodes in a bottleneck spanning tree, as discussed in Problem 21-4 on page 601, 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 bounded from above by the cost of the costliest edge in a bottleneck hamiltonian cycle.)

35.2-5

Suppose that the vertices for an instance of the traveling-salesperson 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.

35.2-6

Adapt the proof of Theorem 35.3 to show that for any constant $c \geq 0$, there is no polynomial-time approximation algorithm with approximation ratio $|V|^c$ for the general traveling-salesperson problem.

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. Instead, this section investigates 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.

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 subfamily $\mathcal{C} \subseteq \mathcal{F}$ ***covers*** a set of elements U if

$$U \subseteq \bigcup_{S \in \mathcal{C}} S \ .$$

The problem is to find a minimum-size subfamily $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S \ .$$

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 subfamily $\mathcal{C}$ 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 you have a given set of people available to work on the problem. You 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. The decision version of the set-covering problem asks 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 in the procedure GREEDY-SET-COVER on the facing page works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered. 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.

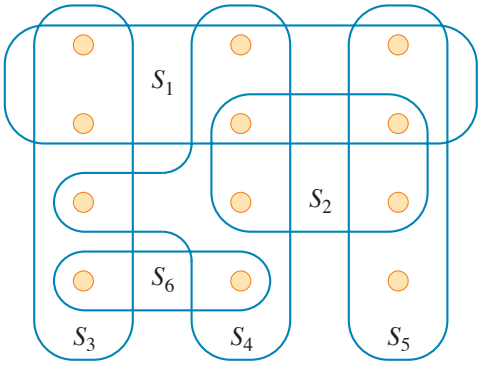


Figure 35.3 An instance $(X, \mathcal{F})$ of the set-covering problem, where X consists of the 12 tan points and $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. Each set $S_i \in \mathcal{F}$ is outlined in blue. A minimum-size set cover is $\mathcal{C} = \{S_3, S_4, S_5\}$, with size 3. The greedy 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.

```
GREEDY-SET-COVER(X, F)
1  U_0 = X
2  C = ∅
3  i = 0
4  while U_i ≠ ∅
5      select S ∈ F that maximizes |S ∩ U_i|
6      U_{i+1} = U_i − S
7      C = C ∪ {S}
8      i = i + 1
9  return C
```

The greedy algorithm works as follows. At the start of each iteration, U_i is a subset of X containing the remaining uncovered elements, with the initial subset U_0 containing all the elements in X. The set $\mathcal{C}$ contains the subfamily being constructed. Line 5 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 6 updates the set of remaining uncovered elements, denoting it by U_{i+1}, and line 7 places S into $\mathcal{C}$. When the algorithm terminates, $\mathcal{C}$ is a subfamily of $\mathcal{F}$ that covers X.

Analysis

We now show that the greedy algorithm returns a set cover that is not too much larger than an optimal set cover.

Theorem 35.4
The procedure GREEDY-SET-COVER run on a set X and family of subsets $\mathcal{F}$ is a polynomial-time $O(\lg X)$-approximation algorithm.

Proof Let's first show that the algorithm runs in time that is polynomial in $|X|$ and $|\mathcal{F}|$. The number of iterations of the loop in lines 4–7 is bounded above by $\min\{|X|, |\mathcal{F}|\} = O(|X| + |\mathcal{F}|)$. The loop body can be implemented to run in $O(|X| \cdot |\mathcal{F}|)$ time. Thus the algorithm runs in $O(|X| \cdot |\mathcal{F}| \cdot (|X| + |\mathcal{F}|))$ time, which is polynomial in the input size. (Exercise 35.3-3 asks for a linear-time algorithm.)

To prove the approximation bound, let $\mathcal{C}^*$ be an optimal set cover for the original instance $(X, \mathcal{F})$, and let $k = |\mathcal{C}^*|$. Since $\mathcal{C}^*$ is also a set cover of each subset U_i of X constructed by the algorithm, we know that any subset U_i constructed by the algorithm can be covered by k sets. Therefore, if $(U_i, \mathcal{F})$ is an instance of the set-covering problem, its optimal set cover has size at most k.

If an optimal set cover for an instance $(U_i, \mathcal{F})$ has size at most k, at least one of the sets in $\mathcal{C}$ covers at least $|U_i|/k$ new elements. Thus, line 5 of GREEDY-SET-COVER, which chooses a set with the maximum number of uncovered elements, must choose a set in which the number of newly covered elements is at least $|U_i|/k$. These elements are removed when constructing U_{i+1}, giving

$$\begin{aligned} |U_{i+1}| &\le |U_i| - |U_i|/k \\ &= |U_i|(1 - 1/k) . \end{aligned} \tag{35.8}$$

Iterating inequality (35.8) gives

$$\begin{aligned} |U_0| &= |X| , \\ |U_1| &\le |U_0|(1 - 1/k) , \\ |U_2| &\le |U_1|(1 - 1/k) = |U|(1 - 1/k)^2 , \end{aligned}$$

and in general

$$|U_i| \le |U_0|(1 - 1/k)^i = |X|(1 - 1/k)^i . \tag{35.9}$$

The algorithm stops when $U_i = \emptyset$, which means that $|U_i| < 1$. Thus an upper bound on the number of iterations of the algorithm is the smallest value of i for which $|U_i| < 1$.

Since $1 + x \le e^x$ for all real x (see inequality (3.14) on page 66), by letting $x = -1/k$, we have $1 - 1/k \le e^{-1/k}$, so that $(1 - 1/k)^k \le (e^{-1/k})^k = 1/e$. Denoting the number i of iterations by ck for some nonnegative integer c, we want c such that

$$|X|(1 - 1/k)^{ck} \le |X| e^{-c} < 1 . \tag{35.10}$$

Multiplying both sides by e^c and then taking the natural logarithm of both sides gives $c \ge \ln |X|$, so we can choose for c any integer that is at least $\ln |X|$. We

choose $c = \lceil \ln |X| \rceil$. Since $i = ck$ is an upper bound on the number of iterations, which equals the size of $\mathcal{C}$, and $k = |\mathcal{C}^*|$, we have $|\mathcal{C}| \le i = ck = c\,|\mathcal{C}^*| = |\mathcal{C}^*| \lceil \ln |X| \rceil$, and the theorem follows. ■

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 you 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 the vertex-cover problem to it.

35.3-3
Show how to implement GREEDY-SET-COVER to run in $O\left(\sum_{S \in \mathcal{F}} |S|\right)$ time.

35.3-4
The proof of Theorem 35.4 says that when GREEDY-SET-COVER, run on the instance $(X, \mathcal{F})$, returns the subfamily $\mathcal{C}$, then $|\mathcal{C}| \le |\mathcal{C}^*| \lceil \ln X \rceil$. Show that the following weaker bound 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 it breaks ties in line 5. Give a procedure BAD-SET-COVER-INSTANCE(n) that returns an n-element instance of the set-covering problem for which, depending on how line 5 breaks ties, GREEDY-SET-COVER can return a number of different solutions that is exponential in n.

35.4 Randomization and linear programming

This section studies two useful techniques for designing approximation algorithms: randomization and linear programming. It starts with a simple randomized algorithm for an optimization version of 3-CNF satisfiability, and then it shows how to design an approximation algorithm for a weighted version of the vertex-cover problem based on linear programming. 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:

$$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \le \rho(n). \tag{35.11}$$

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, you might instead want to know how "close" to satisfiable it is, that is, 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. You might be surprised 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, but we're about to see why. Recall that the definition of 3-CNF satisfiability from Section 34.4 requires each clause to consist of exactly three distinct literals. We now further assume that no clause contains both a variable and its negation. Exercise 35.4-1 asks you to remove this last assumption.

Theorem 35.5

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 each variable is independently set to 1 with probability $1/2$ and to 0 with probability $1/2$. Define, for $i = 1, 2, \ldots, m$, the indicator random variable

$$Y_i = \mathrm{I}\{\text{clause } i \text{ is satisfied}\} ,$$

so that $Y_i = 1$ as long as at least one of the literals in the ith clause is set to 1. Since no literal appears more than once in the same clause, and since we assume 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 Lemma 5.1 on page 130 gives $\mathrm{E}[Y_i] = 7/8$. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + \cdots + Y_m$. Then, we have

$$\begin{aligned}
\mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\
&= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\
&= \sum_{i=1}^{m} 7/8 \\
&= 7m/8 .
\end{aligned}$$

Since m is an upper bound on the number of satisfied clauses, the approximation ratio is at most $m/(7m/8) = 8/7$. ■

Approximating weighted vertex cover using linear programming

The ***minimum-weight vertex-cover problem*** takes as input an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. The weight $w(V')$ of a vertex cover $V' \subseteq V$ is the sum of the weights of its vertices: $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight.

The approximation algorithm for unweighted vertex cover from Section 35.1 won't work here, because the solution it returns could be far from optimal for the weighted problem. Instead, we'll first compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. Then we'll "round" this solution and use it to obtain a vertex cover.

Start by associating a variable $x(v)$ with each vertex $v \in V$, and require that $x(v)$ equals either 0 or 1 for each $v \in V$. The vertex cover includes v if and only if $x(v) = 1$. Then the constraint that for any edge (u, v), at least one of u and v must belong to the vertex cover can be expressed 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:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.12}$$

subject to

$$x(u) + x(v) \geq 1 \qquad \text{for each } (u, v) \in E \tag{35.13}$$

$$x(v) \in \{0, 1\} \quad \text{for each } v \in V\,. \tag{35.14}$$

In the special case in which all the weights $w(v)$ equal 1, this formulation is the optimization version of the NP-hard vertex-cover problem. Let's remove the constraint that $x(v) \in \{0, 1\}$ and replace it by $0 \leq x(v) \leq 1$, resulting in the following linear program:

$$\text{minimize} \quad \sum_{v \in V} w(v)\, x(v) \tag{35.15}$$

subject to

$$x(u) + x(v) \geq 1 \quad \text{for each } (u, v) \in E \tag{35.16}$$

$$x(v) \leq 1 \quad \text{for each } v \in V \tag{35.17}$$

$$x(v) \geq 0 \quad \text{for each } v \in V\,. \tag{35.18}$$

We refer to this linear program as the ***linear-programming relaxation***. Any feasible solution to the 0-1 integer program in lines (35.12)–(35.14) is also a feasible solution to its linear-programming relaxation in lines (35.15)–(35.18). Therefore, the value of an optimal solution to the linear-programming relaxation provides 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 procedure APPROX-MIN-WEIGHT-VC on the facing page starts with a solution to the linear-programming relaxation and uses it to construct an approximate solution to the minimum-weight vertex-cover problem. The procedure works as follows. Line 1 initializes the vertex cover to be empty. Line 2 formulates the linear-programming relaxation in lines (35.15)–(35.18) and then solves this linear program. An optimal solution gives each vertex v an associated value $\bar{x}(v)$, where $0 \leq \bar{x}(v) \leq 1$. The procedure uses this value to guide the choice of which vertices to add to the vertex cover C in lines 3–5: the vertex cover C includes vertex v if and only if $\bar{x}(v) \geq 1/2$. In effect, the procedure "rounds" each fractional variable in the solution to the linear-programming relaxation to either 0 or 1 in order to obtain a solution to the 0-1 integer program in lines (35.12)–(35.14). Finally, line 6 returns the vertex cover C.

Theorem 35.6

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time2-approximation algorithm for the minimum-weight vertex-cover problem.

```
APPROX-MIN-WEIGHT-VC(G, w)
1  C = ∅
2  compute x̄, an optimal solution to the linear-programming relaxation
       in lines (35.15)–(35.18)
3  for each vertex v ∈ V
4      if x̄(v) ≥ 1/2
5          C = C ∪ {v}
6  return C
```

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.

It remains to show that APPROX-MIN-WEIGHT-VC is a2-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-programming relaxation in lines (35.15)–(35.18). Since an optimal vertex cover is a feasible solution to the linear-programming relaxation, z^* must be a lower bound on $w(C^*)$, that is,

$$z^* \le w(C^*) . \tag{35.19}$$

Next, we claim that rounding the fractional values of the variables $\bar{x}(v)$ in lines 3–5 produces a set C that is a vertex cover and satisfies $w(C) \le 2z^*$. To see that C is a vertex cover, consider any edge $(u, v) \in E$. By constraint (35.16), we know that $x(u) + x(v) \ge 1$, 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

$$\begin{aligned}
z^* &= \sum_{v \in V} w(v)\,\bar{x}(v) \\
&\ge \sum_{v \in V : \bar{x}(v) \ge 1/2} w(v)\,\bar{x}(v) \\
&\ge \sum_{v \in V : \bar{x}(v) \ge 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C) .
\end{aligned} \tag{35.20}$$

Combining inequalities (35.19) and (35.20) gives

$$w(C) \le 2z^* \le 2w(C^*) ,$$

and hence APPROX-MIN-WEIGHT-VC is a2-approximation algorithm. ■

Exercises

35.4-1
Show that even if a clause is allowed 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 three literals. Give a randomized 2-approximation algorithm for the MAX-CNF satisfiability problem.

35.4-3
In the MAX-CUT problem, the input is an unweighted undirected graph $G = (V, E)$. We define a cut $(S, V - S)$ as in Chapter 21 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 each vertex v is randomly and independently placed into S with probability $1/2$ and into $V - S$ with probability $1/2$. Show that this algorithm is a randomized 2-approximation algorithm.

35.4-4
Show that the constraints in line (35.17) are redundant in the sense that removing them from the linear-programming relaxation in lines (35.15)–(35.18) yields a linear program for which any optimal solution x must satisfy $x(v) \le 1$ 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 given by 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. The optimization problem seeks a subset of $\{x_1, x_2, \ldots, x_n\}$

whose sum is as large as possible but not larger than t. For example, consider a truck that can carry no more than t pounds, which is to be loaded with up to n different boxes, the ith of which weighs x_i pounds. How heavy a load can the truck take without exceeding the t-pound weight limit?

We start this section with an exponential-time algorithm to compute the optimal value for this optimization problem. 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 you compute, for each subset S' of S, the sum of the elements in S', and then you select, among the subsets whose sum does not exceed t, the one whose sum is closest to t. This algorithm returns the optimal solution, but it might take exponential time. To implement this algorithm, you can 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, you would realize that once a particular subset S' has a sum exceeding t, there is no reason to maintain it, since no superset of S' can be an optimal solution. Let's see how to implement this strategy.

The procedure EXACT-SUBSET-SUM takes an input set $S = \{x_1, x_2, \ldots, x_n\}$, the size $n = |S|$, and a target value t. This procedure iteratively 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 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 $L + 2 = \langle 3, 4, 5, 7, 11 \rangle$. This notation extends to sets, so that

$$S + x = \{s + x : s \in S\} \ .$$

```
EXACT-SUBSET-SUM(S, n, t)
1  L_0 = ⟨0⟩
2  for i = 1 to n
3       L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
4       remove from L_i every element that is greater than t
5  return the largest element in L_n
```

EXACT-SUBSET-SUM invokes 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 on page 36, MERGE-LISTS runs in $O(|L| + |L'|)$ time. We omit the pseudocode for MERGE-LISTS.

To see how EXACT-SUBSET-SUM works, let P_i denote the set of values obtained by selecting each (possibly empty) subset of $\{x_1, x_2, \ldots, x_i\}$ and summing its members. For example, if $S = \{1, 4, 5\}$, then

$$P_1 = \{0, 1\} ,$$
$$P_2 = \{0, 1, 4, 5\} ,$$
$$P_3 = \{0, 1, 4, 5, 6, 9, 10\} .$$

Given the identity

$$P_i = P_{i-1} \cup (P_{i-1} + x_i) , \tag{35.21}$$

you 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

The key to devising a fully polynomial-time approximation scheme for the subset-sum problem is to "trim" each list L_i after it is created. Here's the idea behind trimming: if two values in L are close to each other, then since the goal is just an approximate solution, there is no need to maintain both of them explicitly. More precisely, use a trimming parameter δ such that $0 < \delta < 1$. When ***trimming*** a list L by δ, 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, some element z still in L' approximates y. For z to approximate y, it must be no greater than y and also within a factor of $1 + \delta$ of y, so that

$$\frac{y}{1 + \delta} \le z \le y . \tag{35.22}$$

You 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 (35.22). For example, suppose that $\delta = 0.1$ and

$$L = \langle 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 \rangle .$$

Then trimming L results in

$$L' = \langle 10, 12, 15, 20, 23, 29 \rangle ,$$

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 procedure TRIM trims list $L = \langle y_1, y_2, \ldots, y_m \rangle$ in $\Theta(m)$ time, given L and the trimming parameter δ. It assumes that L is sorted into monotonically increasing order. The output of the procedure is a trimmed, sorted list. 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'.

```
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'
```

Given the procedure TRIM, the procedure APPROX-SUBSET-SUM on the following page implements the approximation scheme. This procedure takes as input a set $S = \{x_1, x_2, \ldots, x_n\}$ of n integers (in arbitrary order), the size $n = |S|$, the target integer t, and an approximation parameter ϵ, where

$$0 < \epsilon < 1 \,. \tag{35.23}$$

It returns a value z^* whose value is within a factor of $1 + \epsilon$ of the optimal solution.

The APPROX-SUBSET-SUM procedure works as follows. Line 1 initializes the list L_0 to be the list containing just the element 0. The **for** loop in lines 2–5 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 the procedure creates L_i from L_{i-1}, it must ensure that the repeated trimming doesn't introduce too much compounded inaccuracy. That's why instead of the trimming parameter being ϵ in the call to TRIM, it has the smaller value $\epsilon/2n$. We'll soon see that APPROX-SUBSET-SUM returns a correct approximation if one exists.

```
APPROX-SUBSET-SUM(S, n, t, ε)
1  L_0 = ⟨0⟩
2  for i = 1 to n
3       L_i = MERGE-LISTS(L_{i-1}, L_{i-1} + x_i)
4       L_i = TRIM(L_i, ε/2n)
5       remove from L_i every element that is greater than t
6  let z* be the largest value in L_n
7  return z*
```

As an example, suppose that APPROX-SUBSET-SUM is given

$$S = \langle 104, 102, 201, 101 \rangle$$

with $t = 308$ and $\epsilon = 0.40$. The trimming parameter δ is $\epsilon/2n = 0.40/8 = 0.05$. The procedure computes the following values on the indicated lines:

line 1: $L_0 = \langle 0 \rangle$,

line 3: $L_1 = \langle 0, 104 \rangle$,
line 4: $L_1 = \langle 0, 104 \rangle$,
line 5: $L_1 = \langle 0, 104 \rangle$,

line 3: $L_2 = \langle 0, 102, 104, 206 \rangle$,
line 4: $L_2 = \langle 0, 102, 206 \rangle$,
line 5: $L_2 = \langle 0, 102, 206 \rangle$,

line 3: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$,
line 4: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$,
line 5: $L_3 = \langle 0, 102, 201, 303 \rangle$,

line 3: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,
line 4: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$,
line 5: $L_4 = \langle 0, 101, 201, 302 \rangle$.

The procedure 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 35.7
APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.

Proof The operations of trimming L_i in line 4 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 7 is indeed the sum of some subset of S, that is, $z^* \in P_n$. Let $y^* \in P_n$ denote an optimal solution to the subset-sum problem, so that it is the greatest value in P_n that is less than or equal to t. Because line 5 ensures that $z^* \le t$, 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

$$\frac{y}{(1+\epsilon/2n)^i} \le z \le y \; . \tag{35.24}$$

Inequality (35.24) 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

$$\frac{y^*}{z} \le \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.25}$$

Since there exists an element $z \in L_n$ fulfilling inequality (35.25), the inequality must hold for z^*, which is the largest value in L_n, which is to say

$$\frac{y^*}{z^*} \le \left(1 + \frac{\epsilon}{2n}\right)^n \; . \tag{35.26}$$

Now we show that $y^*/z^* \le 1+\epsilon$. We do so by showing that $(1+\epsilon/2n)^n \le 1+\epsilon$. First, inequality (35.23), $0 < \epsilon < 1$, implies that

$$(\epsilon/2)^2 \le \epsilon/2 \tag{35.27}$$

Next, from equation (3.16) on page 66, we have $\lim_{n\to\infty}(1 + \epsilon/2n)^n = e^{\epsilon/2}$. Exercise 35.5-3 asks you to show that

$$\frac{d}{dn}\left(1 + \frac{\epsilon}{2n}\right)^n > 0 \; . \tag{35.28}$$

Therefore, the function $(1 + \epsilon/2n)^n$ increases with n as it approaches its limit of $e^{\epsilon/2}$, and we have

$$\begin{aligned}
\left(1 + \frac{\epsilon}{2n}\right)^n &\le e^{\epsilon/2} \\
&\le 1 + \epsilon/2 + (\epsilon/2)^2 && \text{(by inequality (3.15) on page 66)} \\
&\le 1 + \epsilon && \text{(by inequality (35.27))} \; .
\end{aligned} \tag{35.29}$$

Combining inequalities (35.26) and (35.29) 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

$$\begin{aligned}
\log_{1+\epsilon/2n} t + 2 &= \frac{\ln t}{\ln(1+\epsilon/2n)} + 2 \\
&\le \frac{2n(1+\epsilon/2n)\ln t}{\epsilon} + 2 \quad \text{(by inequality (3.23) on page 67)} \\
&< \frac{3n \ln t}{\epsilon} + 2 \quad \text{(by inequality (35.23), } 0 < \epsilon < 1\text{)} .
\end{aligned}$$

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 in turn is polynomial in n—and in $1/\epsilon$. Since the running time of APPROX-SUBSET-SUM is polynomial in the lengths of the lists L_i, we conclude that APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme. ■

Exercises

35.5-1

Prove equation (35.21). Then show that after executing line 4 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 (35.24).

35.5-3

Prove inequality (35.28).

35.5-4

How can 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

You are given a set of n objects, where the size s_i of the ith object satisfies $0 < s_i < 1$. Your goal is 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, as follows. It maintains an ordered list of bins. Let b denote the number of bins in the list, where b increases over the course of the algorithm, and let $\langle B_1, \ldots, B_b \rangle$ be the list of bins. Initially $b = 0$ and the list is empty. The algorithm takes each object i in turn and places it in the lowest-numbered bin that can still accommodate it. If no bin can accommodate object i, then b is incremented and a new bin B_b is opened, containing object i. 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 at most half full.

d. Prove that the number of bins used by the first-fit heuristic never exceeds $\lceil 2S \rceil$.

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 sets have weights in the set-covering problem, so that each set S_i in the family $\mathcal{F}$ has an associated weight w_i. The weight of a cover $\mathcal{C}$ is $\sum_{S_i \in \mathcal{C}} w_i$. The goal is 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. Letting d be the maximum size of any set S_i, show that your heuristic has an approximation ratio of $H(d) = \sum_{i=1}^{d} 1/i$.

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. Section 25.1 showed how to find a maximum matching in a bipartite graph, that is, a matching such that no other matching in G contains more edges. This problem examines matchings in undirected graphs that are not required to be bipartite.

a. 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 a connected, undirected graph $G = (V, E)$. Give an $O(E)$-time greedy algorithm to find a maximal matching in G.

This problem concentrates 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***, the input has two parts: n jobs, $J_1, J_2, \ldots, J_n$, where each job J_k has an associated nonnegative processing time of p_k, and 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, define $C_{\max} = \max\{C_j : 1 \le j \le n\}$ to be the ***makespan*** of the schedule. The goal is to find a schedule whose makespan is minimum.

For example, consider an input with two machines M_1 and M_2, and four jobs J_1, J_2, J_3, and 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, 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 job J_2 on machine M_1 and jobs J_1, J_3, and J_4 on machine M_2. For this schedule, we have $C_1 = 2$, $C_2 = 12$, $C_3 = 6$, and $C_4 = 11$, and so $C_{\max} = 12$.

Given the input to a parallel-machine-scheduling problem, 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,

$$C^*_{\max} \ge \max\{p_k : 1 \le k \le n\} .$$

b. Show that the optimal makespan is at least as large as the average machine load, that is,

$$C^*_{\max} \ge \frac{1}{m}\sum_{k=1}^{n} p_k .$$

Consider 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

$$C_{\max} \le \frac{1}{m}\sum_{k=1}^{n} p_k + \max\{p_k : 1 \le k \le n\} .$$

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 $v \in V$, denote by $\max(v)$ 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 $w(S_G) \geq w(T_G)/2$ 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 15.2. The input includes n items, where the ith item is worth v_i dollars and weighs w_i pounds. The input also includes the capacity of a knapsack, which is 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: $v_1 \geq v_2 \geq \cdots \geq v_n$.

In the 0-1 knapsack problem, the goal is 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 a fraction of each item may be put into the knapsack, rather than either all or none of each item. If a fraction x_i of item i goes into the knapsack, where $0 \leq x_i \leq 1$, it contributes $x_i w_i$ to the weight of the knapsack and adds value $x_i v_i$. The goal of this problem is to develop a polynomial-time 2-approximation algorithm for the 0-1 knapsack problem.

In order to design a polynomial-time algorithm, let's consider restricted instances of the 0-1 knapsack problem. Given an instance I of the knapsack problem, 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 to find an optimal solution Q_j to the fractional problem for instance I_j, you can include item j and then use the greedy algorithm in which each step takes as much as possible of the unchosen item with the maximum value per pound v_i/w_i in the set $\{j+1, j+2, \ldots, n\}$.

c. Prove that there is always 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, either all of the item or none of the item goes into 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 [221] credits Garey, Graham, and Ullman [175] and Johnson [236] with formalizing the concept of a polynomial-time approximation algorithm. The first such algorithm is often credited to Graham [197].

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. Texts by Ausiello et al. [29], Hochbaum [221], Vazirani [446], and Williamson and Shmoys [459] deal exclusively with approximation algorithms, as do surveys by Shmoys [409] and Klein and Young [256]. Several other texts, such as Garey and Johnson [176] and Papadimitriou and Steiglitz [353], have significant coverage of approximation algorithms as well. Books edited by Lawler, Lenstra, Rinnooy Kan, and Shmoys [277] and by Gutin and Punnen [204] provide extensive treatments of approximation algorithms and heuristics for the traveling-salesperson 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 [221] 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 [384]. Christofides improved on this algorithm and gave a $3/2$-approximation algorithm for the traveling-salesperson problem with the triangle inequality. Arora [23] and Mitchell [330] have shown that if the points lie in the euclidean plane, there is a polynomial-time approximation scheme. Theorem 35.3 is due to Sahni and Gonzalez [392].

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 [234].

Problem 35-7 is a combinatorial version of a more general result on approximating knapsack-type integer programs by Bienstock and McClosky [55].

The randomized algorithm for MAX-3-CNF satisfiability is implicit in the work of Johnson [236]. The weighted vertex-cover algorithm is by Hochbaum [220]. 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 [374], and it has had many subsequent uses. (See Motwani, Naor, and Raghavan [335] for a survey.) Several other notable ideas in the field of approximation algorithms include the primal-dual method (see Goemans and Williamson [184] for a survey), finding sparse cuts for use in divide-and-conquer algorithms [288], and the use of semidefinite programming [183].

As mentioned in the chapter notes for Chapter 34, 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 [26] contains a good description of the relationship between probabilistically checkable proofs and the hardness of approximating various problems.