15 Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. (“Programming” in this context refers to a tabular method, not to writing computer code.) As we saw in Chapters 2 and 4, divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

We typically apply dynamic programming to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an* optimal solution to the problem, as opposed to *the* optimal solution, since there may be several solutions that achieve the optimal value.

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem. If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 15.1 examines the problem of cutting a rod into rods of smaller length in way that maximizes their total value. Section 15.2 asks how we can multiply a chain of matrices while performing the fewest total scalar multiplications. Given these examples of dynamic programming, Section 15.3 discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. Section 15.4 then shows how to find the longest common subsequence of two sequences via dynamic programming. Finally, Section 15.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

15.1 Rod cutting

Our first example uses dynamic programming to solve a simple problem in deciding where to cut steel rods. Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of Serling Enterprises wants to know the best way to cut up the rods.

We assume that we know, for $i = 1, 2, \ldots$, the price p_i in dollars that Serling Enterprises charges for a rod of length i inches. Rod lengths are always an integral number of inches. Figure 15.1 gives a sample price table.

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

Consider the case when $n = 4$. Figure 15.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. We see that cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end,

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

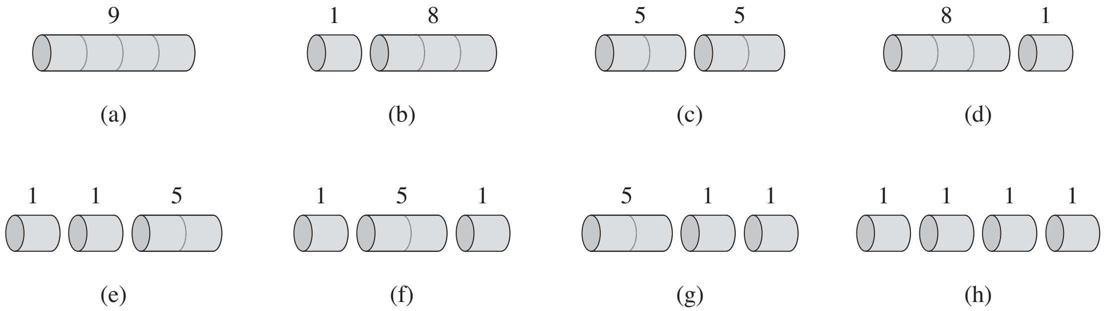


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

for $i = 1, 2, \ldots, n-1$.[1] We denote a decomposition into pieces using ordinary additive notation, so that $7 = 2 + 2 + 3$ indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3. If an optimal solution cuts the rod into k pieces, for some $1 \le k \le n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths i_1, i_2, ..., i_k provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} .$$

For our sample problem, we can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

[1] If we required the pieces to be cut in order of nondecreasing size, there would be fewer ways to consider. For $n = 4$, we would consider only 5 such ways: parts (a), (b), (c), (e), and (h) in Figure 15.2. The number of ways is called the ***partition function***; it is approximately equal to $e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$. This quantity is less than 2^{n-1}, but still much greater than any polynomial in n. We shall not pursue this line of inquiry further, however.

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)} \; , \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)} \; , \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)} \; , \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 \; , \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 \; , \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)} \; , \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \ \text{ or } \ 7 = 2 + 2 + 3 \; , \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 \; , \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 \; , \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)} \; .
\end{aligned}
$$

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) \; . \tag{15.1}$$

The first argument, p_n, corresponds to making no cuts at all and selling the rod of length n as is. The other $n - 1$ arguments to max correspond to the maximum revenue obtained by making an initial cut of the rod into two pieces of size i and $n - i$, for each $i = 1, 2, \ldots, n - 1$, and then optimally cutting up those pieces further, obtaining revenues r_i and r_{n-i} from those two pieces. Since we don't know ahead of time which value of i optimizes revenue, we have to consider all possible values for i and pick the one that maximizes revenue. We also have the option of picking no i at all if we can obtain more revenue by selling the rod uncut.

Note that to solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes. Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting problem exhibits ***optimal substructure***: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, we view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. We may view every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. When doing so, we can couch the solution with no cuts at all as saying that the first piece has size $i = n$ and revenue p_n and that the remainder has size 0 with corresponding revenue $r_0 = 0$. We thus obtain the following simpler version of equation (15.1):

$$r_n = \max_{1 \leq i \leq n} \left(p_i + r_{n-i}\right) \; . \tag{15.2}$$

In this formulation, an optimal solution embodies the solution to only *one* related subproblem—the remainder—rather than two.

Recursive top-down implementation

The following procedure implements the computation implicit in equation (15.2) in a straightforward, top-down, recursive manner.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = −∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

Procedure CUT-ROD takes as input an array $p[1 \,..\, n]$ of prices and an integer n, and it returns the maximum revenue possible for a rod of length n. If $n = 0$, no revenue is possible, and so CUT-ROD returns 0 in line 2. Line 3 initializes the maximum revenue q to $-\infty$, so that the **for** loop in lines 4–5 correctly computes $q = \max_{1 \le i \le n}(p_i + \text{CUT-ROD}(p, n-i))$; line 6 then returns this value. A simple induction on n proves that this answer is equal to the desired answer r_n, using equation (15.2).

If you were to code up CUT-ROD in your favorite programming language and run it on your computer, you would find that once the input size becomes moderately large, your program would take a long time to run. For $n = 40$, you would find that your program takes at least several minutes, and most likely more than an hour. In fact, you would find that each time you increase n by 1, your program's running time would approximately double.

Why is CUT-ROD so inefficient? The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values; it solves the same subproblems repeatedly. Figure 15.3 illustrates what happens for $n = 4$: CUT-ROD(p, n) calls CUT-ROD$(p, n-i)$ for $i = 1, 2, \ldots, n$. Equivalently, CUT-ROD(p, n) calls CUT-ROD(p, j) for each $j = 0, 1, \ldots, n-1$. When this process unfolds recursively, the amount of work done, as a function of n, grows explosively.

To analyze the running time of CUT-ROD, let $T(n)$ denote the total number of calls made to CUT-ROD when called with its second parameter equal to n. This expression equals the number of nodes in a subtree whose root is labeled n in the recursion tree. The count includes the initial call at its root. Thus, $T(0) = 1$ and

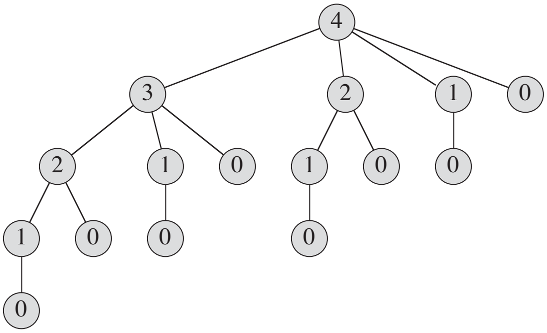


Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) . \qquad (15.3)$$

The initial 1 is for the call at the root, and the term $T(j)$ counts the number of calls (including recursive calls) due to the call CUT-ROD$(p, n - i)$, where $j = n - i$. As Exercise 15.1-1 asks you to show,

$$T(n) = 2^n , \qquad (15.4)$$

and so the running time of CUT-ROD is exponential in n.

In retrospect, this exponential running time is not so surprising. CUT-ROD explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n. The tree of recursive calls has 2^{n-1} leaves, one for each possible way of cutting up the rod. The labels on the simple path from the root to a leaf give the sizes of each remaining right-hand piece before making each cut. That is, the labels give the corresponding cut points, measured from the right-hand end of the rod.

Using dynamic programming for optimal rod cutting

We now show how to convert CUT-ROD into an efficient algorithm, using dynamic programming.

The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only *once*, saving its solution. If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it. Dynamic programming thus uses additional memory to save computation time; it serves an example of a ***time-memory trade-off***. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution. A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. We shall illustrate both of them with our rod-cutting example.

The first approach is ***top-down with memoization***.[2] In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. We say that the recursive procedure has been ***memoized***; it "remembers" what results it has computed previously.

The second approach is the ***bottom-up method***. This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls.

Here is the the pseudocode for the top-down CUT-ROD procedure, with memoization added:

```
MEMOIZED-CUT-ROD($p, n$)
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX($p, n, r$)
```

[2] This is not a misspelling. The word really is *memoization*, not *memorization*. *Memoization* comes from *memo*, since the technique consists of recording a value so that we can look it up later.

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
1  if $r[n] \geq 0$
2      return $r[n]$
3  if $n == 0$
4      $q = 0$
5  else $q = -\infty$
6      for $i = 1$ to $n$
7          $q = \max(q, p[i] +$ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
8  $r[n] = q$
9  return $q$
```

Here, the main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0 \,..\, n]$ with the value $-\infty$, a convenient choice with which to denote "unknown." (Known revenue values are always nonnegative.) It then calls its helper routine, MEMOIZED-CUT-ROD-AUX.

The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of our previous procedure, CUT-ROD. It first checks in line 1 to see whether the desired value is already known and, if it is, then line 2 returns it. Otherwise, lines 3–7 compute the desired value q in the usual manner, line 8 saves it in $r[n]$, and line 9 returns it.

The bottom-up version is even simpler:

```
BOTTOM-UP-CUT-ROD$(p, n)$
1  let $r[0 \,..\, n]$ be a new array
2  $r[0] = 0$
3  for $j = 1$ to $n$
4      $q = -\infty$
5      for $i = 1$ to $j$
6          $q = \max(q, p[i] + r[j - i])$
7      $r[j] = q$
8  return $r[n]$
```

For the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD uses the natural ordering of the subproblems: a problem of size i is "smaller" than a subproblem of size j if $i < j$. Thus, the procedure solves subproblems of sizes $j = 0, 1, \ldots, n$, in that order.

Line 1 of procedure BOTTOM-UP-CUT-ROD creates a new array $r[0 \,..\, n]$ in which to save the results of the subproblems, and line 2 initializes $r[0]$ to 0, since a rod of length 0 earns no revenue. Lines 3–6 solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now

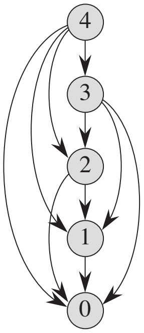


Figure 15.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that we need a solution to subproblem y when solving subproblem x. This graph is a reduced version of the tree of Figure 15.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem of size $j - i$. Line 7 saves in $r[j]$ the solution to the subproblem of size j. Finally, line 8 returns $r[n]$, which equals the optimal value r_n.

The bottom-up and top-down versions have the same asymptotic running time. The running time of procedure BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly-nested loop structure. The number of iterations of its inner **for** loop, in lines 5–6, forms an arithmetic series. The running time of its top-down counterpart, MEMOIZED-CUT-ROD, is also $\Theta(n^2)$, although this running time may be a little harder to see. Because a recursive call to solve a previously solved subproblem returns immediately, MEMOIZED-CUT-ROD solves each subproblem just once. It solves subproblems for sizes $0, 1, \ldots, n$. To solve a subproblem of size n, the **for** loop of lines 6–7 iterates n times. Thus, the total number of iterations of this **for** loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series, giving a total of $\Theta(n^2)$ iterations, just like the inner **for** loop of BOTTOM-UP-CUT-ROD. (We actually are using a form of aggregate analysis here. We shall see aggregate analysis in detail in Section 17.1.)

Subproblem graphs

When we think about a dynamic-programming problem, we should understand the set of subproblems involved and how subproblems depend on one another.

The ***subproblem graph*** for the problem embodies exactly this information. Figure 15.4 shows the subproblem graph for the rod-cutting problem with $n = 4$. It is a directed graph, containing one vertex for each distinct subproblem. The subproblem graph has a directed edge from the vertex for subproblem x to the vertex for subproblem y if determining an optimal solution for subproblem x involves directly considering an optimal solution for subproblem y. For example, the subproblem graph contains an edge from x to y if a top-down recursive procedure for solving x directly calls itself to solve y. We can think of the subproblem graph as a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method, in which we coalesce all nodes for the same subproblem into a single vertex and direct all edges from parent to child.

The bottom-up method for dynamic programming considers the vertices of the subproblem graph in such an order that we solve the subproblems y adjacent to a given subproblem x before we solve subproblem x. (Recall from Section B.4 that the adjacency relation is not necessarily symmetric.) Using the terminology from Chapter 22, in a bottom-up dynamic-programming algorithm, we consider the vertices of the subproblem graph in an order that is a "reverse topological sort," or a "topological sort of the transpose" (see Section 22.4) of the subproblem graph. In other words, no subproblem is considered until all of the subproblems it depends upon have been solved. Similarly, using notions from the same chapter, we can view the top-down method (with memoization) for dynamic programming as a "depth-first search" of the subproblem graph (see Section 22.3).

The size of the subproblem graph $G = (V, E)$ can help us determine the running time of the dynamic programming algorithm. Since we solve each subproblem just once, the running time is the sum of the times needed to solve each subproblem. Typically, the time to compute the solution to a subproblem is proportional to the degree (number of outgoing edges) of the corresponding vertex in the subproblem graph, and the number of subproblems is equal to the number of vertices in the subproblem graph. In this common case, the running time of dynamic programming is linear in the number of vertices and edges.

Reconstructing a solution

Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes. We can extend the dynamic-programming approach to record not only the optimal *value* computed for each subproblem, but also a *choice* that led to the optimal value. With this information, we can readily print an optimal solution.

Here is an extended version of BOTTOM-UP-CUT-ROD that computes, for each rod size j, not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off:

```
EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
 1  let $r[0..n]$ and $s[0..n]$ be new arrays
 2  $r[0] = 0$
 3  for $j = 1$ to $n$
 4      $q = -\infty$
 5      for $i = 1$ to $j$
 6          if $q < p[i] + r[j - i]$
 7              $q = p[i] + r[j - i]$
 8              $s[j] = i$
 9      $r[j] = q$
10  return $r$ and $s$
```

This procedure is similar to BOTTOM-UP-CUT-ROD, except that it creates the array s in line 1, and it updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j.

The following procedure takes a price table p and a rod size n, and it calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1..n]$ of optimal first-piece sizes and then prints out the complete list of piece sizes in an optimal decomposition of a rod of length n:

```
PRINT-CUT-ROD-SOLUTION$(p, n)$
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD$(p, n)$
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

A call to PRINT-CUT-ROD-SOLUTION$(p, 10)$ would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for r_7 given earlier.

Exercises

15.1-1

Show that equation (15.4) follows from equation (15.3) and the initial condition $T(0) = 1$.

15.1-2

Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

15.1-3

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

15.1-4

Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution, too.

15.1-5

The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

15.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \cdots A_n \; . \tag{15.5}$$

We can evaluate the expression (15.5) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is ***fully parenthesized*** if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then we can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$$
\begin{array}{l}
(A_1(A_2(A_3A_4)))\ , \\
(A_1((A_2A_3)A_4))\ , \\
((A_1A_2)(A_3A_4))\ , \\
((A_1(A_2A_3))A_4)\ , \\
(((A_1A_2)A_3)A_4)\ .
\end{array}
$$

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode, which generalizes the Square-Matrix-Multiply procedure from Section 4.2. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

```
Matrix-Multiply$(A, B)$
1  if $A.columns \neq B.rows$
2      error "incompatible dimensions"
3  else let $C$ be a new $A.rows \times B.columns$ matrix
4      for $i = 1$ to $A.rows$
5          for $j = 1$ to $B.columns$
6              $c_{ij} = 0$
7              for $k = 1$ to $A.columns$
8                  $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
9      return $C$
```

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr. In what follows, we shall express costs in terms of the number of scalar multiplications.

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. If we multiply according to the parenthesization $((A_1A_2)A_3)$, we perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product A_1A_2, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1(A_2A_3))$, we perform $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product A_2A_3, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension

$p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and $(k+1)$st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \, , \\ \displaystyle\sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 \, . \end{cases} \qquad (15.6)$$

Problem 12-4 asked you to show that the solution to a similar recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n / n^{3/2})$. A simpler exercise (see Exercise 15.2-3) is to show that the solution to the recurrence (15.6) is $\Omega(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence that we stated at the beginning of this chapter:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.

4. Construct an optimal solution from computed information.

We shall go through these steps in order, demonstrating clearly how we apply each step to the problem.

Step 1: The structure of an optimal parenthesization

For our first step in the dynamic-programming paradigm, we find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. In the matrix-chain multiplication problem, we can perform this step as follows. For convenience, let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is, for some value of k, we first compute the matrices $A_{i..k}$ and $A_{k+1..j}$ and then multiply them together to produce the final product $A_{i..j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i..k}$, plus the cost of computing $A_{k+1..j}$, plus the cost of multiplying them together.

The optimal substructure of this problem is as follows. Suppose that to optimally parenthesize $A_i A_{i+1} \cdots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$. Why? If there were a less costly way to parenthesize $A_i A_{i+1} \cdots A_k$, then we could substitute that parenthesization in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ to produce another way to parenthesize $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for how we parenthesize the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. We have seen that any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.

Step 2: A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Let $m[i,j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$; for the full problem, the lowest-cost way to compute $A_{1..n}$ would thus be $m[1,n]$.

We can define $m[i,j]$ recursively as follows. If $i = j$, the problem is trivial; the chain consists of just one matrix $A_{i..i} = A_i$, so that no scalar multiplications are necessary to compute the product. Thus, $m[i,i] = 0$ for $i = 1, 2, \ldots, n$. To compute $m[i,j]$ when $i < j$, we take advantage of the structure of an optimal solution from step 1. Let us assume that to optimally parenthesize, we split the product $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, where $i \le k < j$. Then, $m[i,j]$ equals the minimum cost for computing the subproducts $A_{i..k}$ and $A_{k+1..j}$, plus the cost of multiplying these two matrices together. Recalling that each matrix A_i is $p_{i-1} \times p_i$, we see that computing the matrix product $A_{i..k} A_{k+1..j}$ takes $p_{i-1} p_k p_j$ scalar multiplications. Thus, we obtain

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j \; .$$

This recursive equation assumes that we know the value of k, which we do not. There are only $j - i$ possible values for k, however, namely $k = i, i+1, \ldots, j-1$. Since the optimal parenthesization must use one of these values for k, we need only check them all to find the best. Thus, our recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \; , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \; . \end{cases} \tag{15.7}$$

The $m[i,j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i,j]$ to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization. That is, $s[i,j]$ equals a value k such that $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

Step 3: Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (15.7) to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \cdots A_n$. As we saw for the rod-cutting problem, and as we shall see in Section 15.3, this recursive algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.

Observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta(n^2)$ in all. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure).

Instead of computing the solution to recurrence (15.7) recursively, we compute the optimal cost by using a tabular, bottom-up approach. (We present the corresponding top-down approach using memoization in Section 15.3.)

We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears below. This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. Its input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $p.length = n + 1$. The procedure uses an auxiliary table $m[1 \mathinner{..} n, 1 \mathinner{..} n]$ for storing the $m[i, j]$ costs and another auxiliary table $s[1 \mathinner{..} n-1, 2 \mathinner{..} n]$ that records which index of k achieved the optimal cost in computing $m[i, j]$. We shall use the table s to construct an optimal solution.

In order to implement the bottom-up approach, we must determine which entries of the table we refer to when computing $m[i, j]$. Equation (15.7) shows that the cost $m[i, j]$ of computing a matrix-chain product of $j-i+1$ matrices depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices. That is, for $k = i, i+1, \ldots, j-1$, the matrix $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length. For the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, we consider the subproblem size to be the length $j - i + 1$ of the chain.

```
MATRIX-CHAIN-ORDER($p$)
$n = p.length - 1$
let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
for $i = 1$ to $n$
    $m[i,i] = 0$
for $l = 2$ to $n$                  // $l$ is the chain length
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$
            if $q < m[i,j]$
                $m[i,j] = q$
                $s[i,j] = k$
return $m$ and $s$
```

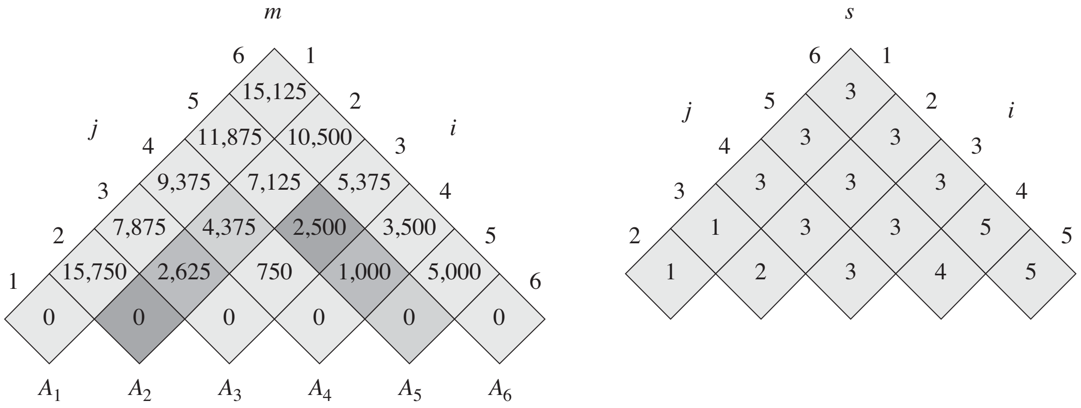


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

The algorithm first computes $m[i,i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 3–4. It then uses recurrence (15.7) to compute $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ (the minimum costs for chains of length $l = 2$) during the first execution of the **for** loop in lines 5–13. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i,j]$ cost computed in lines 10–13 depends only on table entries $m[i,k]$ and $m[k+1,j]$ already computed.

Figure 15.5 illustrates this procedure on a chain of $n = 6$ matrices. Since we have defined $m[i,j]$ only for $i \le j$, only the portion of the table m strictly above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, we can find the minimum cost $m[i,j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices at the intersection of lines running northeast from A_i and northwest from A_j. Each horizontal row in the table contains the entries for matrix chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i, j]$ using the products $p_{i-1} p_k p_j$ for $k = i, i+1, \ldots, j-1$ and all entries southwest and southeast from $m[i, j]$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n-1$ values. Exercise 15.2-5 asks you to show that the running time of this algorithm is in fact also $\Omega(n^3)$. The algorithm requires $\Theta(n^2)$ space to store the m and s tables. Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Step 4: Constructing an optimal solution

Although MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1..n-1, 2..n]$ gives us the information we need to do so. Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$. We can determine the earlier matrix multiplications recursively, since $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1..s[1,n]}$ and $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1..n}$. The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$.

```
PRINT-OPTIMAL-PARENS$(s, i, j)$
if $i == j$
    print "A"$_i$
else print "("
    PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
    PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
    print ")"
```

In the example of Figure 15.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Exercises

15.2-1

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

15.2-2

Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call would be MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.)

15.2-3

Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$.

15.2-4

Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

15.2-5

Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3} .$$

(*Hint:* You may find equation (A.3) useful.)

15.2-6

Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

15.3 Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should we look for a dynamic-programming solution to a problem? In this section, we examine the two key ingredients that an optithe number of scalar multiplications. Does this problem exhibit optimal substructure?

15.3-4

As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) *before* solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

15.3-5

Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

15.3-6

Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1, 2, \ldots, n$, where you start with currency 1 and wish to wind up with currency n. You are given, for each pair of currencies i and j, an exchange rate r_{ij}, meaning that if you start with d units of currency i, you can trade for dr_{ij} units of currency j. A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that, if $c_k = 0$ for all $k = 1, 2, \ldots, n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure. Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

15.4 Longest common subsequence

Biological applications often need to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called

bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. (See Appendix C for the definition of a string.) For example, the DNA of one organism may be $S_1 =$ ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be $S_2 =$ GTCGTTCGGAATGCCGTTGCTCTGTAAA. One reason to compare two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are. We can, and do, define similarity in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. (Chapter 32 explores algorithms to solve this problem.) In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 15-5 looks at this notion.) Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is GTCGTCGGAAGCCGGCCGAA.

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. The sequence $\langle B, C, A \rangle$ is not a *longest* common subsequence (LCS) of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater.

In the ***longest-common-subsequence problem***, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y. This section shows how to efficiently solve the LCS problem using dynamic programming.

Step 1: Characterizing a longest common subsequence

In a brute-force approach to solving the LCS problem, we would enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence we find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

The LCS problem has an optimal-substructure property, however, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be precise, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the ith ***prefix*** of X, for $i = 0, 1, \ldots, m$, as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k-1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2). ■

The way that Theorem 15.1 characterizes longest common subsequences tells us that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property, as we shall see in a moment.

Step 2: A recursive solution

Theorem 15.1 implies that we should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then we must solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whichever of these two LCSs is longer is an LCS of X and Y. Because these cases exhaust all possibilities, we know that one of the optimal subproblem solutions must appear within an LCS of X and Y.

We can readily see the overlapping-subproblems property in the LCS problem. To find an LCS of X and Y, we may need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

As in the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a recurrence for the value of an optimal solution. Let us define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \,. \end{cases} \tag{15.9}$$

Observe that in this recursive formulation, a condition in the problem restricts which subproblems we may consider. When $x_i = y_j$, we can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}. Otherwise, we instead consider the two subproblems of finding an LCS of X_i and Y_{j-1} and of X_{i-1} and Y_j. In the previous dynamic-programming algorithms we have examined—for rod cutting and matrix-chain multiplication—we ruled out no subproblems due to conditions in the problem. Finding an LCS is not the only dynamic-programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem (see Problem 15-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (15.9), we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since the LCS problem has only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

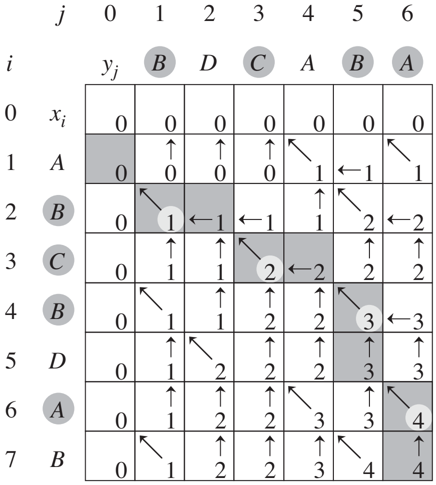
Procedure LCS-LENGTH takes two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as inputs. It stores the $c[i, j]$ values in a table $c[0 \mathinner{..} m, 0 \mathinner{..} n]$, and it computes the entries in ***row-major*** order. (That is, the procedure fills in the first row of c from left to right, then the second row, and so on.) The procedure also maintains the table $b[1 \mathinner{..} m, 1 \mathinner{..} n]$ to help us construct an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. The procedure returns the b and c tables; $c[m, n]$ contains the length of an LCS of X and Y.

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

Figure 15.8 shows the tables produced by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The running time of the procedure is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute.

Step 4: Constructing an LCS

The b table returned by LCS-LENGTH enables us to quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. We simply begin at $b[m, n]$ and trace through the table by following the arrows. Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS$(b, X, X.length, Y.length)$.

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

For the b table in Figure 15.8, this procedure prints $BCBA$. The procedure takes time $O(m + n)$, since it decrements at least one of i and j in each recursive call.

Improving the code

Once you have developed an algorithm, you will often find that you can improve on the time or space it uses. Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance. Others can yield substantial asymptotic savings in time and space.

In the LCS algorithm, for example, we can eliminate the b table altogether. Each $c[i, j]$ entry depends on only three other c table entries: $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$. Given the value of $c[i, j]$, we can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b. Thus, we can reconstruct an LCS in $O(m+n)$ time using a procedure similar to PRINT-LCS. (Exercise 15.4-2 asks you to give the pseudocode.) Although we save $\Theta(mn)$ space by this method, the auxiliary space requirement for computing an LCS does not asymptotically decrease, since we need $\Theta(mn)$ space for the c table anyway.

We can, however, reduce the asymptotic space requirements for LCS-LENGTH, since it needs only two rows of table c at a time: the row being computed and the previous row. (In fact, as Exercise 15.4-4 asks you to show, we can use only slightly more than the space for one row of c to compute the length of an LCS.) This improvement works if we need only the length of an LCS; if we need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace our steps in $O(m+n)$ time.

Exercises

15.4-1

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m+n)$ time, without using the b table.

15.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

15.4-4

Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

15.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

15.4-6 ★

Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

15.5 Optimal binary search trees

Suppose that we are designing a program to translate text from English to French. For each occurrence of each English word in the text, we need to look up its French equivalent. We could perform these lookup operations by building a binary search tree with n English words as keys and their French equivalents as satellite data. Because we will search the tree for each individual word in the text, we want the total time spent searching to be as low as possible. We could ensure an $O(\lg n)$ search time per occurrence by using a red-black tree or any other balanced binary search tree. Words appear with different frequencies, however, and a frequently used word such as *the* may appear far from the root while a rarely used word such as *machicolation* appears near the root. Such an organization would slow down the translation, since the number of nodes visited when searching for a key in a binary search tree equals one plus the depth of the node containing the key. We want words that occur frequently in the text to be placed nearer the root.[6] Moreover, some words in the text might have no French translation,[7] and such words would not appear in the binary search tree at all. How do we organize a binary search tree so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?

What we need is known as an ***optimal binary search tree***. Formally, we are given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 < \cdots < k_n$), and we wish to build a binary search tree from these keys. For each key k_i, we have a probability p_i that a search will be for k_i. Some searches may be for values not in K, and so we also have $n + 1$ "dummy keys"

[6] If the subject of the text is castle architecture, we might want *machicolation* to appear near the root.

[7] Yes, *machicolation* has a French counterpart: *mâchicoulis*.

26 Maximum Flow

Just as we can model a road map as a directed graph in order to find the shortest path from one point to another, we can also interpret a directed graph as a "flow network" and use it to answer questions about material flows. Imagine a material coursing through a system from a source, where the material is produced, to a sink, where it is consumed. The source produces the material at some steady rate, and the sink consumes the material at the same rate. The "flow" of the material at any point in the system is intuitively the rate at which the material moves. Flow networks can model many problems, including liquids flowing through pipes, parts through assembly lines, current through electrical networks, and information through communication networks.

We can think of each directed edge in a flow network as a conduit for the material. Each conduit has a stated capacity, given as a maximum rate at which the material can flow through the conduit, such as 200 gallons of liquid per hour through a pipe or 20 amperes of electrical current through a wire. Vertices are conduit junctions, and other than the source and sink, material flows through the vertices without collecting in them. In other words, the rate at which material enters a vertex must equal the rate at which it leaves the vertex. We call this property "flow conservation," and it is equivalent to Kirchhoff's current law when the material is electrical current.

In the maximum-flow problem, we wish to compute the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints. It is one of the simplest problems concerning flow networks and, as we shall see in this chapter, this problem can be solved by efficient algorithms. Moreover, we can adapt the basic techniques used in maximum-flow algorithms to solve other network-flow problems.

This chapter presents two general methods for solving the maximum-flow problem. Section 26.1 formalizes the notions of flow networks and flows, formally defining the maximum-flow problem. Section 26.2 describes the classical method of Ford and Fulkerson for finding maximum flows. An application of this method, finding a maximum matching in an undirected bipartite graph, appears in Section 26.3. Section 26.4 presents the push-relabel method, which underlies many of the fastest algorithms for network-flow problems. Section 26.5 covers the "relabel-to-front" algorithm, a particular implementation of the push-relabel method that runs in time $O(V^3)$. Although this algorithm is not the fastest algorithm known, it illustrates some of the techniques used in the asymptotically fastest algorithms, and it is reasonably efficient in practice.

26.1 Flow networks

In this section, we give a graph-theoretic definition of flow networks, discuss their properties, and define the maximum-flow problem precisely. We also introduce some helpful notation.

Flow networks and flows

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v) \geq 0$. We further require that if E contains an edge (u, v), then there is no edge (v, u) in the reverse direction. (We shall see shortly how to work around this restriction.) If $(u, v) \notin E$, then for convenience we define $c(u, v) = 0$, and we disallow self-loops. We distinguish two vertices in a flow network: a ***source*** s and a ***sink*** t. For convenience, we assume that each vertex lies on some path from the source to the sink. That is, for each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$. The graph is therefore connected and, since each vertex other than s has at least one entering edge, $|E| \geq |V| - 1$. Figure 26.1 shows an example of a flow network.

We are now ready to define flows more formally. Let $G = (V, E)$ be a flow network with a capacity function c. Let s be the source of the network, and let t be the sink. A ***flow*** in G is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ that satisfies the following two properties:

Capacity constraint: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: For all $u \in V - \{s, t\}$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) .$$

When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$.

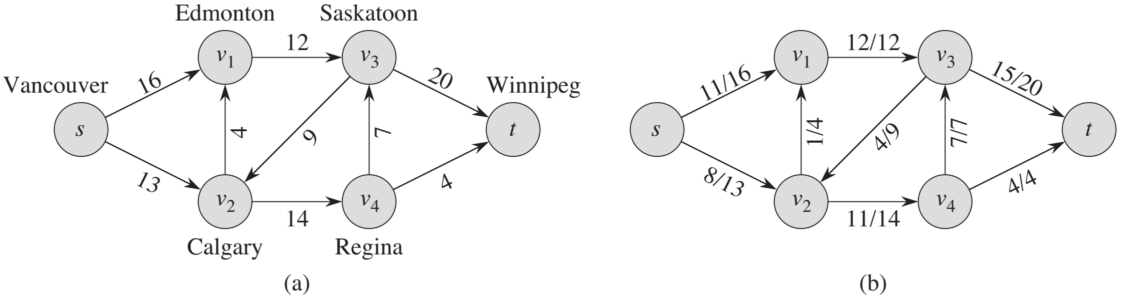


Figure 26.1 **(a)** A flow network $G = (V, E)$ for the Lucky Puck Company's trucking problem. The Vancouver factory is the source s, and the Winnipeg warehouse is the sink t. The company ships pucks through intermediate cities, but only $c(u, v)$ crates per day can go from city u to city v. Each edge is labeled with its capacity. **(b)** A flow f in G with value $|f| = 19$. Each edge (u, v) is labeled by $f(u, v)/c(u, v)$. The slash notation merely separates the flow and capacity; it does not indicate division.

We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v. The ***value*** $|f|$ of a flow f is defined as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) , \tag{26.1}$$

that is, the total flow out of the source minus the flow into the source. (Here, the $|\cdot|$ notation denotes flow value, not absolute value or cardinality.) Typically, a flow network will not have any edges into the source, and the flow into the source, given by the summation $\sum_{v \in V} f(v, s)$, will be 0. We include it, however, because when we introduce residual networks later in this chapter, the flow into the source will become significant. In the ***maximum-flow problem***, we are given a flow network G with source s and sink t, and we wish to find a flow of maximum value.

Before seeing an example of a network-flow problem, let us briefly explore the definition of flow and the two flow properties. The capacity constraint simply says that the flow from one vertex to another must be nonnegative and must not exceed the given capacity. The flow-conservation property says that the total flow into a vertex other than the source or sink must equal the total flow out of that vertex—informally, "flow in equals flow out."

An example of flow

A flow network can model the trucking problem shown in Figure 26.1(a). The Lucky Puck Company has a factory (source s) in Vancouver that manufactures hockey pucks, and it has a warehouse (sink t) in Winnipeg that stocks them. Lucky

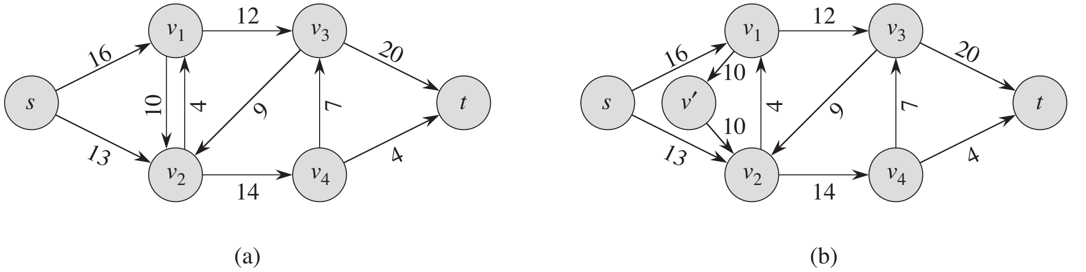


Figure 26.2 Converting a network with antiparallel edges to an equivalent one with no antiparallel edges. **(a)** A flow network containing both the edges (v_1, v_2) and (v_2, v_1). **(b)** An equivalent network with no antiparallel edges. We add the new vertex v', and we replace edge (v_1, v_2) by the pair of edges (v_1, v') and (v', v_2), both with the same capacity as (v_1, v_2).

Puck leases space on trucks from another firm to ship the pucks from the factory to the warehouse. Because the trucks travel over specified routes (edges) between cities (vertices) and have a limited capacity, Lucky Puck can ship at most $c(u, v)$ crates per day between each pair of cities u and v in Figure 26.1(a). Lucky Puck has no control over these routes and capacities, and so the company cannot alter the flow network shown in Figure 26.1(a). They need to determine the largest number p of crates per day that they can ship and then to produce this amount, since there is no point in producing more pucks than they can ship to their warehouse. Lucky Puck is not concerned with how long it takes for a given puck to get from the factory to the warehouse; they care only that p crates per day leave the factory and p crates per day arrive at the warehouse.

We can model the "flow" of shipments with a flow in this network because the number of crates shipped per day from one city to another is subject to a capacity constraint. Additionally, the model must obey flow conservation, for in a steady state, the rate at which pucks enter an intermediate city must equal the rate at which they leave. Otherwise, crates would accumulate at intermediate cities.

Modeling problems with antiparallel edges

Suppose that the trucking firm offered Lucky Puck the opportunity to lease space for 10 crates in trucks going from Edmonton to Calgary. It would seem natural to add this opportunity to our example and form the network shown in Figure 26.2(a). This network suffers from one problem, however: it violates our original assumption that if an edge $(v_1, v_2) \in E$, then $(v_2, v_1) \notin E$. We call the two edges (v_1, v_2) and (v_2, v_1) ***antiparallel***. Thus, if we wish to model a flow problem with antiparallel edges, we must transform the network into an equivalent one containing no antiparallel edges. Figure 26.2(b) displays this equivalent network. We choose one of the two antiparallel edges, in this case (v_1, v_2), and split it by adding a new vertex v' and replacing edge (v_1, v_2) with the pair of edges (v_1, v') and (v', v_2). We also set the capacity of both new edges to the capacity of the original edge. The resulting network satisfies the property that if an edge is in the network, the reverse edge is not. Exercise 26.1-1 asks you to prove that the resulting network is equivalent to the original one.

Thus, we see that a real-world flow problem might be most naturally modeled by a network with antiparallel edges. It will be convenient to disallow antiparallel edges, however, and so we have a straightforward way to convert a network containing antiparallel edges into an equivalent one with no antiparallel edges.

Networks with multiple sources and sinks

A maximum-flow problem may have several sources and sinks, rather than just one of each. The Lucky Puck Company, for example, might actually have a set of m factories $\{s_1, s_2, \ldots, s_m\}$ and a set of n warehouses $\{t_1, t_2, \ldots, t_n\}$, as shown in Figure 26.3(a). Fortunately, this problem is no harder than ordinary maximum flow.

We can reduce the problem of determining a maximum flow in a network with multiple sources and multiple sinks to an ordinary maximum-flow problem. Figure 26.3(b) shows how to convert the network from (a) to an ordinary flow network with only a single source and a single sink. We add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, \ldots, m$. We also create a new ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, \ldots, n$. Intuitively, any flow in the network in (a) corresponds to a flow in the network in (b), and vice versa. The single source s simply provides as much flow as desired for the multiple sources s_i, and the single sink t likewise consumes as much flow as desired for the multiple sinks t_i. Exercise 26.1-2 asks you to prove formally that the two problems are equivalent.

Exercises

26.1-1

Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v), and we create a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G.

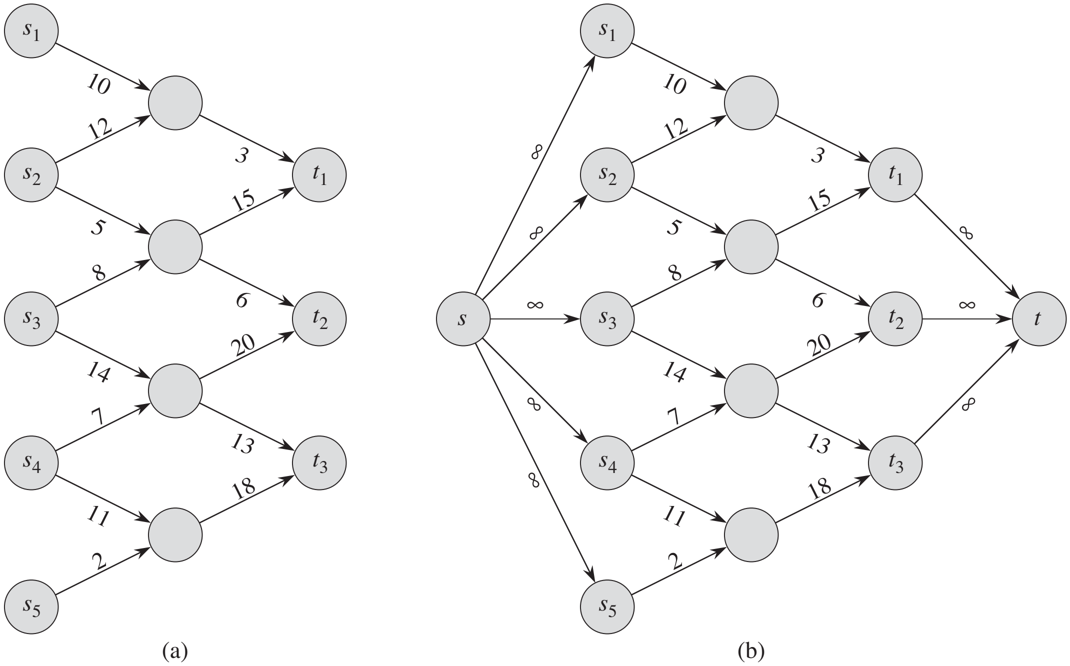


Figure 26.3 Converting a multiple-source, multiple-sink maximum-flow problem into a problem with a single source and a single sink. **(a)** A flow network with five sources $S = \{s_1, s_2, s_3, s_4, s_5\}$ and three sinks $T = \{t_1, t_2, t_3\}$. **(b)** An equivalent single-source, single-sink flow network. We add a supersource s and an edge with infinite capacity from s to each of the multiple sources. We also add a supersink t and an edge with infinite capacity from each of the multiple sinks to t.

26.1-2

Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

26.1-3

Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \leadsto v \leadsto t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \leadsto u \leadsto t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

26.1-4

Let f be a flow in a network, and let α be a real number. The ***scalar flow product***, denoted αf, is a function from $V \times V$ to $\mathbb{R}$ defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v) .$$

Prove that the flows in a network form a ***convex set***. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \le \alpha \le 1$.

26.1-5

State the maximum-flow problem as a linear-programming problem.

26.1-6

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

26.1-7

Suppose that, in addition to edge capacities, a flow network has ***vertex capacities***. That is each vertex v has a limit $l(v)$ on how much flow can pass though v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

26.2 The Ford-Fulkerson method

This section presents the Ford-Fulkerson method for solving the maximum-flow problem. We call it a "method" rather than an "algorithm" because it encompasses several implementations with differing running times. The Ford-Fulkerson method depends on three important ideas that transcend the method and are relevant to many flow algorithms and problems: residual networks, augmenting paths, and cuts. These ideas are essential to the important max-flow min-cut theorem (Theorem 26.6), which characterizes the value of a maximum flow in terms of cuts of the flow network. We end this section by presenting one specific implementation of the Ford-Fulkerson method and analyzing its running time.

The Ford-Fulkerson method iteratively increases the value of the flow. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value in G by finding an "augmenting path" in an associated "residual network" G_f. Once we know the edges of an augmenting path in G_f, we can easily identify specific edges in G for which we can change the flow so that we increase the value of the flow. Although each iteration of the Ford-Fulkerson method increases the value of the flow, we shall see that the flow on any particular edge of G may increase or decrease; decreasing the flow on some edges may be necessary in order to enable an algorithm to send more flow from the source to the sink. We repeatedly augment the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

```
FORD-FULKERSON-METHOD($G, s, t$)
initialize flow $f$ to 0
while there exists an augmenting path $p$ in the residual network $G_f$
    augment flow $f$ along $p$
return $f$
```

In order to implement and analyze the Ford-Fulkerson method, we need to introduce several additional concepts.

Residual networks

Intuitively, given a flow network G and a flow f, the residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G. An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If that value is positive, we place that edge into G_f with a "residual capacity" of $c_f(u, v) = c(u, v) - f(u, v)$. The only edges of G that are in G_f are those that can admit more flow; those edges (u, v) whose flow equals their capacity have $c_f(u, v) = 0$, and they are not in G_f.

The residual network G_f may also contain edges that are not in G, however. As an algorithm manipulates the flow, with the goal of increasing the total flow, it might need to decrease the flow on a particular edge. In order to represent a possible decrease of a positive flow $f(u, v)$ on an edge in G, we place an edge (v, u) into G_f with residual capacity $c_f(v, u) = f(u, v)$—that is, an edge that can admit flow in the opposite direction to (u, v), at most canceling out the flow on (u, v). These reverse edges in the residual network allow an algorithm to send back flow it has already sent along an edge. Sending flow back along an edge is equivalent to *decreasing* the flow on the edge, which is a necessary operation in many algorithms.

More formally, suppose that we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. We define the ***residual capacity*** $c_f(u, v)$ by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \ , \\ f(v,u) & \text{if } (v,u) \in E \ , \\ 0 & \text{otherwise} \ . \end{cases} \qquad (26.2)$$

Because of our assumption that $(u, v) \in E$ implies $(v, u) \notin E$, exactly one case in equation (26.2) applies to each ordered pair of vertices.

As an example of equation (26.2), if $c(u, v) = 16$ and $f(u, v) = 11$, then we can increase $f(u, v)$ by up to $c_f(u, v) = 5$ units before we exceed the capacity constraint on edge (u, v). We also wish to allow an algorithm to return up to 11 units of flow from v to u, and hence $c_f(v, u) = 11$.

Given a flow network $G = (V, E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\} \ . \qquad (26.3)$$

That is, as promised above, each edge of the residual network, or ***residual edge***, can admit a flow that is greater than 0. Figure 26.4(a) repeats the flow network G and flow f of Figure 26.1(b), and Figure 26.4(b) shows the corresponding residual network G_f. The edges in E_f are either edges in E or their reversals, and thus

$$|E_f| \le 2 \, |E| \ .$$

Observe that the residual network G_f is similar to a flow network with capacities given by c_f. It does not satisfy our definition of a flow network because it may contain both an edge (u, v) and its reversal (v, u). Other than this difference, a residual network has the same properties as a flow network, and we can define a flow in the residual network as one that satisfies the definition of a flow, but with respect to capacities c_f in the network G_f.

A flow in a residual network provides a roadmap for adding flow to the original flow network. If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the ***augmentation*** of flow f by f', to be a function from $V \times V$ to $\mathbb{R}$, defined by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \text{if } (u,v) \in E \ , \\ 0 & \text{otherwise} \ . \end{cases} \qquad (26.4)$$

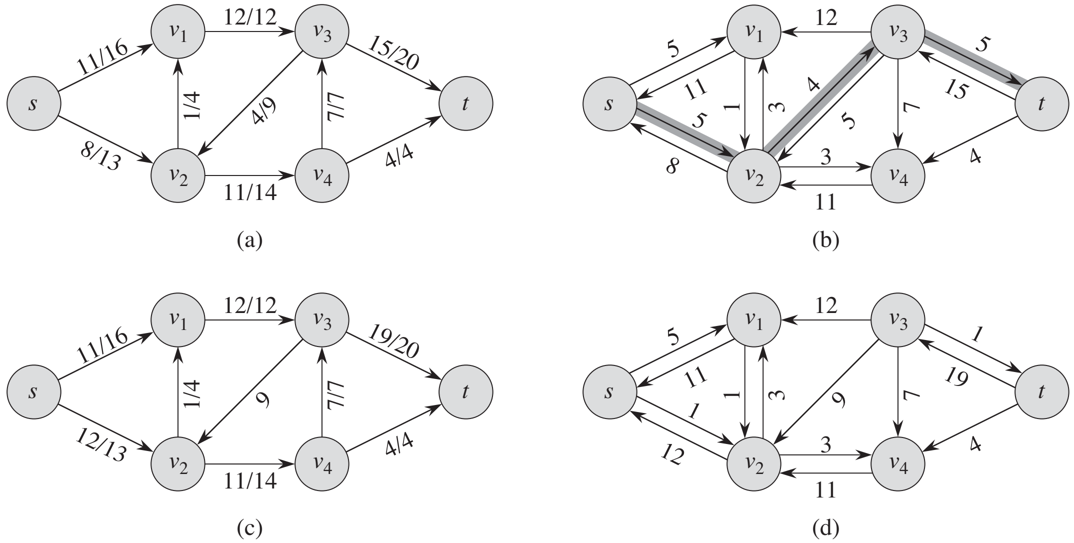


Figure 26.4 **(a)** The flow network G and flow f of Figure 26.1(b). **(b)** The residual network G_f with augmenting path p shaded; its residual capacity is $c_f(p) = c_f(v_2, v_3) = 4$. Edges with residual capacity equal to 0, such as (v_1, v_3), are not shown, a convention we follow in the remainder of this section. **(c)** The flow in G that results from augmenting along path p by its residual capacity 4. Edges carrying no flow, such as (v_3, v_2), are labeled only by their capacity, another convention we follow throughout. **(d)** The residual network induced by the flow in (c).

The intuition behind this definition follows the definition of the residual network. We increase the flow on (u, v) by $f'(u, v)$ but decrease it by $f'(v, u)$ because pushing flow on the reverse edge in the residual network signifies decreasing the flow in the original network. Pushing flow on the reverse edge in the residual network is also known as ***cancellation***. For example, if we send 5 crates of hockey pucks from u to v and send 2 crates from v to u, we could equivalently (from the perspective of the final result) just send 3 creates from u to v and none from v to u. Cancellation of this type is crucial for any maximum-flow algorithm.

Lemma 26.1

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ defined in equation (26.4) is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Proof We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex in $V - \{s, t\}$.

For the capacity constraint, first observe that if $(u, v) \in E$, then $c_f(v, u) = f(u, v)$. Therefore, we have $f'(v, u) \le c_f(v, u) = f(u, v)$, and hence

$$\begin{aligned}
(f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (26.4))} \\
&\ge f(u, v) + f'(u, v) - f(u, v) && \text{(because } f'(v, u) \le f(u, v)\text{)} \\
&= f'(u, v) \\
&\ge 0 \, .
\end{aligned}$$

In addition,

$$\begin{aligned}
&(f \uparrow f')(u, v) \\
&\quad = f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (26.4))} \\
&\quad \le f(u, v) + f'(u, v) && \text{(because flows are nonnegative)} \\
&\quad \le f(u, v) + c_f(u, v) && \text{(capacity constraint)} \\
&\quad = f(u, v) + c(u, v) - f(u, v) && \text{(definition of } c_f\text{)} \\
&\quad = c(u, v) \, .
\end{aligned}$$

For flow conservation, because both f and f' obey flow conservation, we have that for all $u \in V - \{s, t\}$,

$$\begin{aligned}
\sum_{v \in V} (f \uparrow f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v) - f'(v, u)) \\
&= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u) \\
&= \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(v, u) - \sum_{v \in V} f'(u, v) \\
&= \sum_{v \in V} (f(v, u) + f'(v, u) - f'(u, v)) \\
&= \sum_{v \in V} (f \uparrow f')(v, u) \, ,
\end{aligned}$$

where the third line follows from the second by flow conservation.

Finally, we compute the value of $f \uparrow f'$. Recall that we disallow antiparallel edges in G (but not in G_f), and hence for each vertex $v \in V$, we know that there can be an edge (s, v) or (v, s), but never both. We define $V_1 = \{v : (s, v) \in E\}$ to be the set of vertices with edges from s, and $V_2 = \{v : (v, s) \in E\}$ to be the set of vertices with edges to s. We have $V_1 \cup V_2 \subseteq V$ and, because we disallow antiparallel edges, $V_1 \cap V_2 = \emptyset$. We now compute

$$\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')\,(s, v) - \sum_{v \in V} (f \uparrow f')\,(v, s) \\
&= \sum_{v \in V_1} (f \uparrow f')\,(s, v) - \sum_{v \in V_2} (f \uparrow f')\,(v, s) \, , && (26.5)
\end{aligned}$$

where the second line follows because $(f \uparrow f')(w, x)$ is 0 if $(w, x) \notin E$. We now apply the definition of $f \uparrow f'$ to equation (26.5), and then reorder and group terms to obtain

$$\begin{aligned}
|f \uparrow f'| \\
&= \sum_{v \in V_1} (f(s, v) + f'(s, v) - f'(v, s)) - \sum_{v \in V_2} (f(v, s) + f'(v, s) - f'(s, v)) \\
&= \sum_{v \in V_1} f(s, v) + \sum_{v \in V_1} f'(s, v) - \sum_{v \in V_1} f'(v, s) \\
&\qquad - \sum_{v \in V_2} f(v, s) - \sum_{v \in V_2} f'(v, s) + \sum_{v \in V_2} f'(s, v) \\
&= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) \\
&\qquad + \sum_{v \in V_1} f'(s, v) + \sum_{v \in V_2} f'(s, v) - \sum_{v \in V_1} f'(v, s) - \sum_{v \in V_2} f'(v, s) \\
&= \sum_{v \in V_1} f(s, v) - \sum_{v \in V_2} f(v, s) + \sum_{v \in V_1 \cup V_2} f'(s, v) - \sum_{v \in V_1 \cup V_2} f'(v, s) . \qquad (26.6)
\end{aligned}$$

In equation (26.6), we can extend all four summations to sum over V, since each additional term has value 0. (Exercise 26.2-1 asks you to prove this formally.) We thus have

$$\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) \qquad (26.7) \\
&= |f| + |f'| . \qquad \blacksquare
\end{aligned}$$

Augmenting paths

Given a flow network $G = (V, E)$ and a flow f, an ***augmenting path*** p is a simple path from s to t in the residual network G_f. By the definition of the residual network, we may increase the flow on an edge (u, v) of an augmenting path by up to $c_f(u, v)$ without violating the capacity constraint on whichever of (u, v) and (v, u) is in the original flow network G.

The shaded path in Figure 26.4(b) is an augmenting path. Treating the residual network G_f in the figure as a flow network, we can increase the flow through each edge of this path by up to 4 units without violating a capacity constraint, since the smallest residual capacity on this path is $c_f(v_2, v_3) = 4$. We call the maximum amount by which we can increase the flow on each edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ is on } p\} .$$

The following lemma, whose proof we leave as Exercise 26.2-7, makes the above argument more precise.

Lemma 26.2
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p , \\ 0 & \text{otherwise .} \end{cases} \tag{26.8}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. ■

The following corollary shows that if we augment f by f_p, we get another flow in G whose value is closer to the maximum. Figure 26.4(c) shows the result of augmenting the flow f from Figure 26.4(a) by the flow f_p in Figure 26.4(b), and Figure 26.4(d) shows the ensuing residual network.

Corollary 26.3
Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (26.8), and suppose that we augment f by f_p. Then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Proof Immediate from Lemmas 26.1 and 26.2. ■

Cuts of flow networks

The Ford-Fulkerson method repeatedly augments the flow along augmenting paths until it has found a maximum flow. How do we know that when the algorithm terminates, we have actually found a maximum flow? The max-flow min-cut theorem, which we shall prove shortly, tells us that a flow is maximum if and only if its residual network contains no augmenting path. To prove this theorem, though, we must first explore the notion of a cut of a flow network.

A ***cut*** (S, T) of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. (This definition is similar to the definition of "cut" that we used for minimum spanning trees in Chapter 23, except that here we are cutting a directed graph rather than an undirected graph, and we insist that $s \in S$ and $t \in T$.) If f is a flow, then the ***net flow*** $f(S, T)$ across the cut (S, T) is defined to be

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u) . \tag{26.9}$$

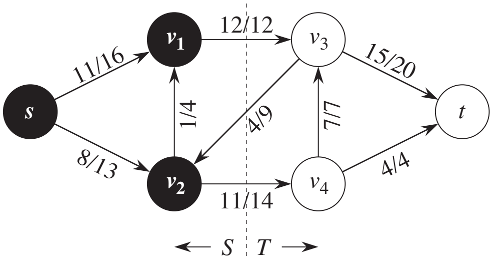


Figure 26.5 A cut (S, T) in the flow network of Figure 26.1(b), where $S = \{s, v_1, v_2\}$ and $T = \{v_3, v_4, t\}$. The vertices in S are black, and the vertices in T are white. The net flow across (S, T) is $f(S, T) = 19$, and the capacity is $c(S, T) = 26$.

The ***capacity*** of the cut (S, T) is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v) . \tag{26.10}$$

A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

The asymmetry between the definitions of flow and capacity of a cut is intentional and important. For capacity, we count only the capacities of edges going from S to T, ignoring edges in the reverse direction. For flow, we consider the flow going from S to T minus the flow going in the reverse direction from T to S. The reason for this difference will become clear later in this section.

Figure 26.5 shows the cut $(\{s, v_1, v_2\}, \{v_3, v_4, t\})$ in the flow network of Figure 26.1(b). The net flow across this cut is

$$\begin{aligned} f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2) &= 12 + 11 - 4 \\ &= 19 , \end{aligned}$$

and the capacity of this cut is

$$\begin{aligned} c(v_1, v_3) + c(v_2, v_4) &= 12 + 14 \\ &= 26 . \end{aligned}$$

The following lemma shows that, for a given flow f, the net flow across any cut is the same, and it equals $|f|$, the value of the flow.

Lemma 26.4

Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.

Proof We can rewrite the flow-conservation condition for any node $u \in V - \{s,t\}$ as

$$\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) = 0 \,. \tag{26.11}$$

Taking the definition of $|f|$ from equation (26.1) and adding the left-hand side of equation (26.11), which equals 0, summed over all vertices in $S - \{s\}$, gives

$$|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \left(\sum_{v \in V} f(u,v) - \sum_{v \in V} f(v,u) \right) .$$

Expanding the right-hand summation and regrouping terms yields

$$\begin{aligned}
|f| &= \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) + \sum_{u \in S - \{s\}} \sum_{v \in V} f(u,v) - \sum_{u \in S - \{s\}} \sum_{v \in V} f(v,u) \\
&= \sum_{v \in V} \left(f(s,v) + \sum_{u \in S - \{s\}} f(u,v) \right) - \sum_{v \in V} \left(f(v,s) + \sum_{u \in S - \{s\}} f(v,u) \right) \\
&= \sum_{v \in V} \sum_{u \in S} f(u,v) - \sum_{v \in V} \sum_{u \in S} f(v,u) \,.
\end{aligned}$$

Because $V = S \cup T$ and $S \cap T = \emptyset$, we can split each summation over V into summations over S and T to obtain

$$\begin{aligned}
|f| &= \sum_{v \in S} \sum_{u \in S} f(u,v) + \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&= \sum_{v \in T} \sum_{u \in S} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&\qquad + \left(\sum_{v \in S} \sum_{u \in S} f(u,v) - \sum_{v \in S} \sum_{u \in S} f(v,u) \right) .
\end{aligned}$$

The two summations within the parentheses are actually the same, since for all vertices $x, y \in V$, the term $f(x,y)$ appears once in each summation. Hence, these summations cancel, and we have

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&= f(S,T) \,.
\end{aligned}$$

■

A corollary to Lemma 26.4 shows how we can use cut capacities to bound the value of a flow.

Corollary 26.5

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Proof Let (S, T) be any cut of G and let f be any flow. By Lemma 26.4 and the capacity constraint,

$$\begin{aligned}
|f| &= f(S,T) \\
&= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&\leq \sum_{u \in S} \sum_{v \in T} f(u,v) \\
&\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\
&= c(S,T) \, .
\end{aligned}$$

■

Corollary 26.5 yields the immediate consequence that the value of a maximum flow in a network is bounded from above by the capacity of a minimum cut of the network. The important max-flow min-cut theorem, which we now state and prove, says that the value of a maximum flow is in fact equal to the capacity of a minimum cut.

Theorem 26.6 (Max-flow min-cut theorem)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G.
2. The residual network G_f contains no augmenting paths.
3. $|f| = c(S, T)$ for some cut (S, T) of G.

Proof (1) ⇒ (2): Suppose for the sake of contradiction that f is a maximum flow in G but that G_f has an augmenting path p. Then, by Corollary 26.3, the flow found by augmenting f by f_p, where f_p is given by equation (26.8), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

(2) ⇒ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$

and $T = V - S$. The partition (S, T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f. Now consider a pair of vertices

$u \in S$ and $v \in T$. If $(u,v) \in E$, we must have $f(u,v) = c(u,v)$, since otherwise $(u,v) \in E_f$, which would place v in set S. If $(v,u) \in E$, we must have $f(v,u) = 0$, because otherwise $c_f(u,v) = f(v,u)$ would be positive and we would have $(u,v) \in E_f$, which would place v in S. Of course, if neither (u,v) nor (v,u) is in E, then $f(u,v) = f(v,u) = 0$. We thus have

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{v \in T} \sum_{u \in S} 0 \\ &= c(S,T) \,. \end{aligned}$$

By Lemma 26.4, therefore, $|f| = f(S,T) = c(S,T)$.

(3) ⇒ (1): By Corollary 26.5, $|f| \le c(S,T)$ for all cuts (S,T). The condition $|f| = c(S,T)$ thus implies that f is a maximum flow. ■

The basic Ford-Fulkerson algorithm

In each iteration of the Ford-Fulkerson method, we find *some* augmenting path p and use p to modify the flow f. As Lemma 26.2 and Corollary 26.3 suggest, we replace f by $f \uparrow f_p$, obtaining a new flow whose value is $|f| + |f_p|$. The following implementation of the method computes the maximum flow in a flow network $G = (V,E)$ by updating the flow attribute $(u,v).f$ for each edge $(u,v) \in E$.[1] If $(u,v) \notin E$, we assume implicitly that $(u,v).f = 0$. We also assume that we are given the capacities $c(u,v)$ along with the flow network, and $c(u,v) = 0$ if $(u,v) \notin E$. We compute the residual capacity $c_f(u,v)$ in accordance with the formula (26.2). The expression $c_f(p)$ in the code is just a temporary variable that stores the residual capacity of the path p.

```
FORD-FULKERSON($G, s, t$)
for each edge $(u,v) \in G.E$
    $(u,v).f = 0$
while there exists a path $p$ from $s$ to $t$ in the residual network $G_f$
    $c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is in } p\}$
    for each edge $(u,v)$ in $p$
        if $(u,v) \in E$
            $(u,v).f = (u,v).f + c_f(p)$
        else $(v,u).f = (v,u).f - c_f(p)$
```

[1] Recall from Section 22.1 that we represent an attribute f for edge (u,v) with the same style of notation—$(u,v).f$—that we use for an attribute of any other object.

The FORD-FULKERSON algorithm simply expands on the FORD-FULKERSON-METHOD pseudocode given earlier. Figure 26.6 shows the result of each iteration in a sample run. Lines 1–2 initialize the flow f to 0. The **while** loop of lines 3–8 repeatedly finds an augmenting path p in G_f and augments flow f along p by the residual capacity $c_f(p)$. Each residual edge in path p is either an edge in the original network or the reversal of an edge in the original network. Lines 6–8 update the flow in each case appropriately, adding flow when the residual edge is an original edge and subtracting it otherwise. When no augmenting paths exist, the flow f is a maximum flow.

Analysis of Ford-Fulkerson

The running time of FORD-FULKERSON depends on how we find the augmenting path p in line 3. If we choose it poorly, the algorithm might not even terminate: the value of the flow will increase with successive augmentations, but it need not even converge to the maximum flow value.[2] If we find the augmenting path by using a breadth-first search (which we saw in Section 22.2), however, the algorithm runs in polynomial time. Before proving this result, we obtain a simple bound for the case in which we choose the augmenting path arbitrarily and all capacities are integers.

In practice, the maximum-flow problem often arises with integral capacities. If the capacities are rational numbers, we can apply an appropriate scaling transformation to make them all integral. If f^* denotes a maximum flow in the transformed network, then a straightforward implementation of FORD-FULKERSON executes the **while** loop of lines 3–8 at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration.

We can perform the work done within the **while** loop efficiently if we implement the flow network $G = (V, E)$ with the right data structure and find an augmenting path by a linear-time algorithm. Let us assume that we keep a data structure corresponding to a directed graph $G' = (V, E')$, where $E' = \{(u, v) : (u, v) \in E$ or $(v, u) \in E\}$. Edges in the network G are also edges in G', and therefore we can easily maintain capacities and flows in this data structure. Given a flow f on G, the edges in the residual network G_f consist of all edges (u, v) of G' such that $c_f(u, v) > 0$, where c_f conforms to equation (26.2). The time to find a path in a residual network is therefore $O(V + E') = O(E)$ if we use either depth-first search or breadth-first search. Each iteration of the **while** loop thus takes $O(E)$ time, as does the initialization in lines 1–2, making the total running time of the FORD-FULKERSON algorithm $O(E\,|f^*|)$.

[2] The Ford-Fulkerson method might fail to terminate only if edge capacities are irrational numbers.

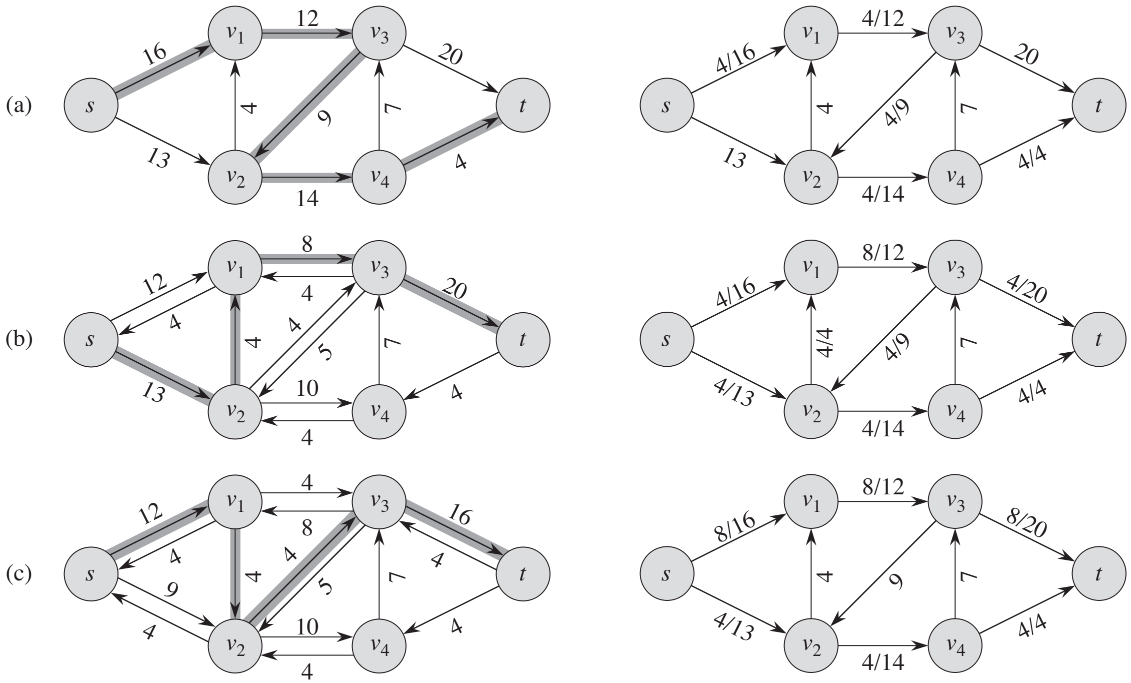


Figure 26.6 The execution of the basic Ford-Fulkerson algorithm. **(a)–(e)** Successive iterations of the **while** loop. The left side of each part shows the residual network G_f from line 3 with a shaded augmenting path p. The right side of each part shows the new flow f that results from augmenting f by f_p. The residual network in (a) is the input network G.

When the capacities are integral and the optimal flow value $|f^*|$ is small, the running time of the Ford-Fulkerson algorithm is good. Figure 26.7(a) shows an example of what can happen on a simple flow network for which $|f^*|$ is large. A maximum flow in this network has value 2,000,000: 1,000,000 units of flow traverse the path $s \to u \to t$, and another 1,000,000 units traverse the path $s \to v \to t$. If the first augmenting path found by FORD-FULKERSON is $s \to u \to v \to t$, shown in Figure 26.7(a), the flow has value 1 after the first iteration. The resulting residual network appears in Figure 26.7(b). If the second iteration finds the augmenting path $s \to v \to u \to t$, as shown in Figure 26.7(b), the flow then has value 2. Figure 26.7(c) shows the resulting residual network. We can continue, choosing the augmenting path $s \to u \to v \to t$ in the odd-numbered iterations and the augmenting path $s \to v \to u \to t$ in the even-numbered iterations. We would perform a total of 2,000,000 augmentations, increasing the flow value by only 1 unit in each.

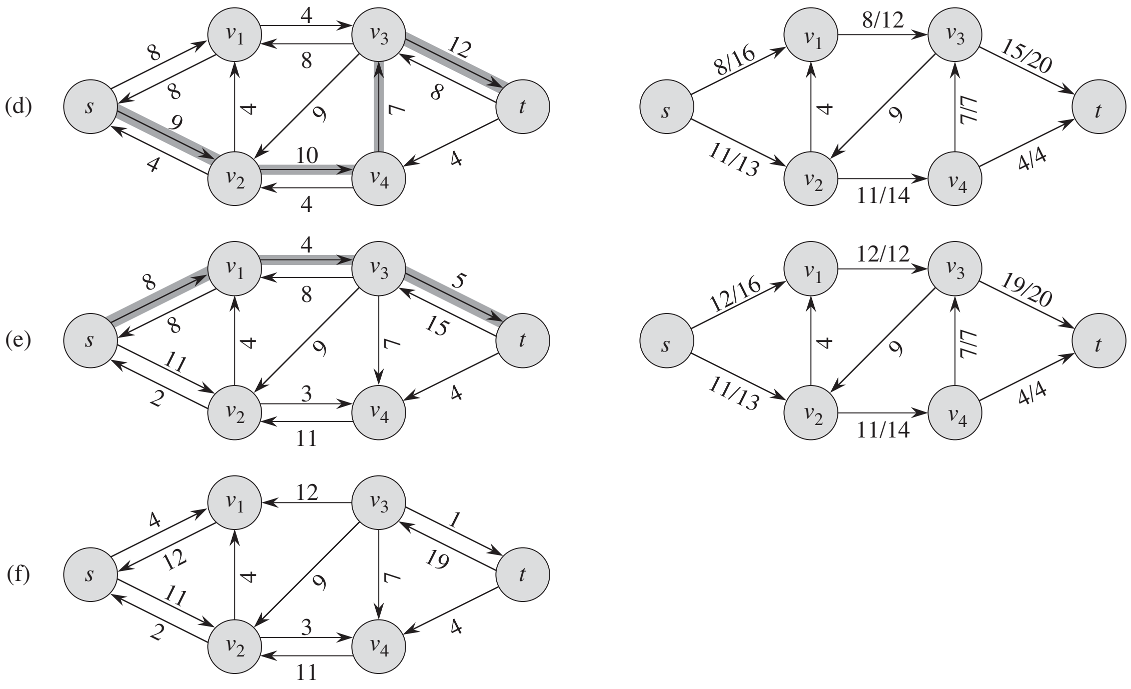


Figure 26.6, continued **(f)** The residual network at the last **while** loop test. It has no augmenting paths, and the flow f shown in (e) is therefore a maximum flow. The value of the maximum flow found is 23.

The Edmonds-Karp algorithm

We can improve the bound on FORD-FULKERSON by finding the augmenting path p in line 3 with a breadth-first search. That is, we choose the augmenting path as a *shortest* path from s to t in the residual network, where each edge has unit distance (weight). We call the Ford-Fulkerson method so implemented the ***Edmonds-Karp algorithm***. We now prove that the Edmonds-Karp algorithm runs in $O(VE^2)$ time.

The analysis depends on the distances to vertices in the residual network G_f. The following lemma uses the notation $\delta_f(u, v)$ for the shortest-path distance from u to v in G_f, where each edge has unit distance.

Lemma 26.7

If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

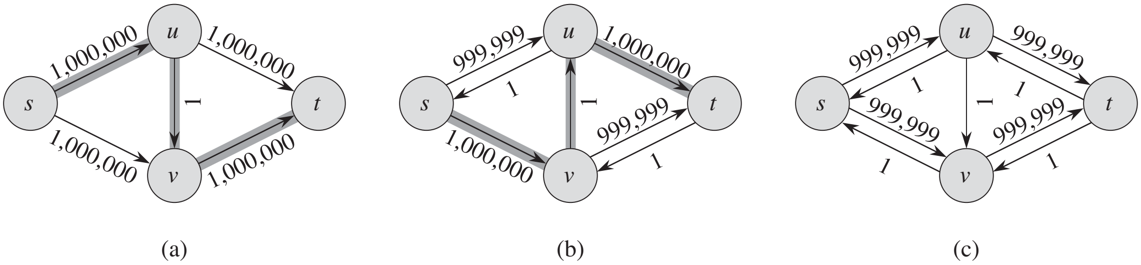


Figure 26.7 **(a)** A flow network for which FORD-FULKERSON can take $\Theta(E\,|f^*|)$ time, where f^* is a maximum flow, shown here with $|f^*| = 2{,}000{,}000$. The shaded path is an augmenting path with residual capacity 1. **(b)** The resulting residual network, with another augmenting path whose residual capacity is 1. **(c)** The resulting residual network.

Proof We will suppose that for some vertex $v \in V - \{s,t\}$, there is a flow augmentation that causes the shortest-path distance from s to v to decrease, and then we will derive a contradiction. Let f be the flow just before the first augmentation that decreases some shortest-path distance, and let f' be the flow just afterward. Let v be the vertex with the minimum $\delta_{f'}(s,v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s,v) < \delta_f(s,v)$. Let $p = s \rightsquigarrow u \to v$ be a shortest path from s to v in $G_{f'}$, so that $(u,v) \in E_{f'}$ and

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1 \,. \tag{26.12}$$

Because of how we chose v, we know that the distance of vertex u from the source s did not decrease, i.e.,

$$\delta_{f'}(s,u) \ge \delta_f(s,u) \,. \tag{26.13}$$

We claim that $(u,v) \notin E_f$. Why? If we had $(u,v) \in E_f$, then we would also have

$$\begin{aligned}
\delta_f(s,v) &\le \delta_f(s,u) + 1 && \text{(by Lemma 24.10, the triangle inequality)} \\
&\le \delta_{f'}(s,u) + 1 && \text{(by inequality (26.13))} \\
&= \delta_{f'}(s,v) && \text{(by equation (26.12))} \,,
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$.

How can we have $(u,v) \notin E_f$ and $(u,v) \in E_{f'}$? The augmentation must have increased the flow from v to u. The Edmonds-Karp algorithm always augments flow along shortest paths, and therefore the shortest path from s to u in G_f has (v,u) as its last edge. Therefore,

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u) - 1 \\
&\le \delta_{f'}(s,u) - 1 && \text{(by inequality (26.13))} \\
&= \delta_{f'}(s,v) - 2 && \text{(by equation (26.12))} \,,
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$. We conclude that our assumption that such a vertex v exists is incorrect. ■

The next theorem bounds the number of iterations of the Edmonds-Karp algorithm.

Theorem 26.8
If the Edmonds-Karp algorithm is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Proof We say that an edge (u, v) in a residual network G_f is ***critical*** on an augmenting path p if the residual capacity of p is the residual capacity of (u, v), that is, if $c_f(p) = c_f(u, v)$. After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We will show that each of the $|E|$ edges can become critical at most $|V|/2$ times.

Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u, v) is critical for the first time, we have

$$\delta_f(s, v) = \delta_f(s, u) + 1 .$$

Once the flow is augmented, the edge (u, v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v, u) appears on an augmenting path. If f' is the flow in G when this event occurs, then we have

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 .$$

Since $\delta_f(s, v) \le \delta_{f'}(s, v)$ by Lemma 26.7, we have

$$\begin{aligned} \delta_{f'}(s, u) &= \delta_{f'}(s, v) + 1 \\ &\ge \delta_f(s, v) + 1 \\ &= \delta_f(s, u) + 2 . \end{aligned}$$

Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2. The distance of u from the source is initially at least 0. The intermediate vertices on a shortest path from s to u cannot contain s, u, or t (since (u, v) on an augmenting path implies that $u \ne t$). Therefore, until u becomes unreachable from the source, if ever, its distance is at most $|V| - 2$. Thus, after the first time that (u, v) becomes critical, it can become critical at most $(|V| - 2)/2 = |V|/2 - 1$ times more, for a total of at most $|V|/2$ times. Since there are $O(E)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$. Each augmenting path has at least one critical edge, and hence the theorem follows. ■

Because we can implement each iteration of FORD-FULKERSON in $O(E)$ time when we find the augmenting path by breadth-first search, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$. We shall see that push-relabel algorithms can yield even better bounds. The algorithm of Section 26.4 gives a method for achieving an $O(V^2E)$ running time, which forms the basis for the $O(V^3)$-time algorithm of Section 26.5.

Exercises

26.2-1
Prove that the summations in equation (26.6) equal the summations in equation (26.7).

26.2-2
In Figure 26.1(b), what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

26.2-3
Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 26.1(a).

26.2-4
In the example of Figure 26.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

26.2-5
Recall that the construction in Section 26.1 that converts a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

26.2-6
Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys

these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

26.2-7
Prove Lemma 26.2.

26.2-8
Suppose that we redefine the residual network to disallow edges into s. Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

26.2-9
Suppose that both f and f' are flows in a network G and we compute flow $f \uparrow f'$. Does the augmented flow satisfy the flow conservation property? Does it satisfy the capacity constraint?

26.2-10
Show how to find a maximum flow in a network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (*Hint:* Determine the paths *after* finding the maximum flow.)

26.2-11
The ***edge connectivity*** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

26.2-12
Suppose that you are given a flow network G, and G has edges entering the source s. Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$-time algorithm to compute f', given f, and assuming that all edge capacities are integers.

26.2-13
Suppose that you wish to find, among all minimum cuts in a flow network G with integral capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G.

26.3 Maximum bipartite matching

Some combinatorial problems can easily be cast as maximum-flow problems. The multiple-source, multiple-sink maximum-flow problem from Section 26.1 gave us one example. Some other combinatorial problems seem on the surface to have little to do with flow networks, but can in fact be reduced to maximum-flow problems. This section presents one such problem: finding a maximum matching in a bipartite graph. In order to solve this problem, we shall take advantage of an integrality property provided by the Ford-Fulkerson method. We shall also see how to use the Ford-Fulkerson method to solve the maximum-bipartite-matching problem on a graph $G = (V, E)$ in $O(VE)$ time.

The maximum-bipartite-matching problem

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by the matching M if some edge in M is incident on v; otherwise, v is ***unmatched***. A ***maximum matching*** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M| \geq |M'|$. In this section, we shall restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge. Figure 26.8 illustrates the notion of a matching in a bipartite graph.

The problem of finding a maximum matching in a bipartite graph has many practical applications. As an example, we might consider matching a set L of machines with a set R of tasks to be performed simultaneously. We take the presence of edge (u, v) in E to mean that a particular machine $u \in L$ is capable of performing a particular task $v \in R$. A maximum matching provides work for as many machines as possible.

Finding a maximum bipartite matching

We can use the Ford-Fulkerson method to find a maximum matching in an undirected bipartite graph $G = (V, E)$ in time polynomial in $|V|$ and $|E|$. The trick is to construct a flow network in which flows correspond to matchings, as shown in Figure 26.8(c). We define the ***corresponding flow network*** $G' = (V', E')$ for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, and we let $V' = V \cup \{s, t\}$. If the vertex partition of G is $V = L \cup R$, the

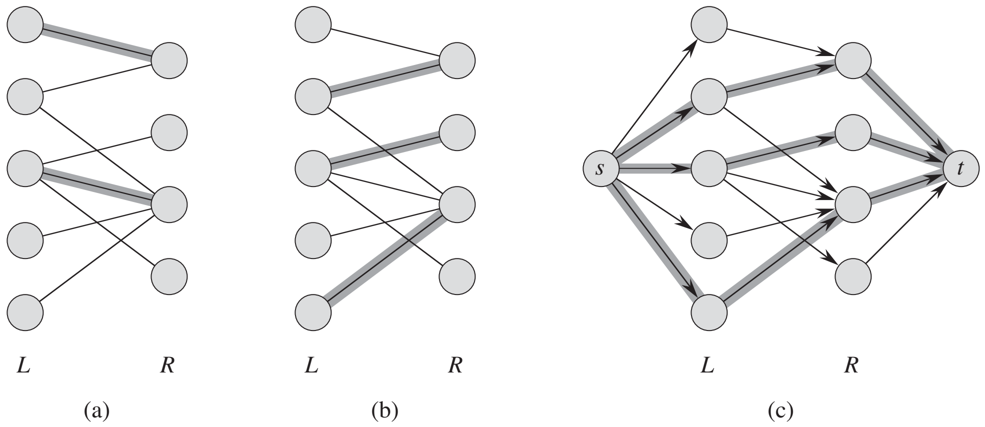


Figure 26.8 A bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$. **(a)** A matching with cardinality 2, indicated by shaded edges. **(b)** A maximum matching with cardinality 3. **(c)** The corresponding flow network G' with a maximum flow shown. Each edge has unit capacity. Shaded edges have a flow of 1, and all other edges carry no flow. The shaded edges from L to R correspond to those in the maximum matching from (b).

directed edges of G' are the edges of E, directed from L to R, along with $|V|$ new directed edges:

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\} .$$

To complete the construction, we assign unit capacity to each edge in E'. Since each vertex in V has at least one incident edge, $|E| \geq |V|/2$. Thus, $|E| \leq |E'| = |E| + |V| \leq 3|E|$, and so $|E'| = \Theta(E)$.

The following lemma shows that a matching in G corresponds directly to a flow in G's corresponding flow network G'. We say that a flow f on a flow network $G = (V, E)$ is ***integer-valued*** if $f(u, v)$ is an integer for all $(u, v) \in V \times V$.

Lemma 26.9
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is a matching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$.

Proof We first show that a matching M in G corresponds to an integer-valued flow f in G'. Define f as follows. If $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$. For all other edges $(u, v) \in E'$, we define $f(u, v) = 0$. It is simple to verify that f satisfies the capacity constraint and flow conservation.

Intuitively, each edge $(u, v) \in M$ corresponds to one unit of flow in G' that traverses the path $s \to u \to v \to t$. Moreover, the paths induced by edges in M are vertex-disjoint, except for s and t. The net flow across cut $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$; thus, by Lemma 26.4, the value of the flow is $|f| = |M|$.

To prove the converse, let f be an integer-valued flow in G', and let

$$M = \{(u, v) : u \in L,\ v \in R,\ \text{and } f(u, v) > 0\} \ .$$

Each vertex $u \in L$ has only one entering edge, namely (s, u), and its capacity is 1. Thus, each $u \in L$ has at most one unit of flow entering it, and if one unit of flow does enter, by flow conservation, one unit of flow must leave. Furthermore, since f is integer-valued, for each $u \in L$, the one unit of flow can enter on at most one edge and can leave on at most one edge. Thus, one unit of flow enters u if and only if there is exactly one vertex $v \in R$ such that $f(u, v) = 1$, and at most one edge leaving each $u \in L$ carries positive flow. A symmetric argument applies to each $v \in R$. The set M is therefore a matching.

To see that $|M| = |f|$, observe that for every matched vertex $u \in L$, we have $f(s, u) = 1$, and for every edge $(u, v) \in E - M$, we have $f(u, v) = 0$. Consequently, $f(L \cup \{s\}, R \cup \{t\})$, the net flow across cut $(L \cup \{s\}, R \cup \{t\})$, is equal to $|M|$. Applying Lemma 26.4, we have that $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$. ■

Based on Lemma 26.9, we would like to conclude that a maximum matching in a bipartite graph G corresponds to a maximum flow in its corresponding flow network G', and we can therefore compute a maximum matching in G by running a maximum-flow algorithm on G'. The only hitch in this reasoning is that the maximum-flow algorithm might return a flow in G' for which some $f(u, v)$ is not an integer, even though the flow value $|f|$ must be an integer. The following theorem shows that if we use the Ford-Fulkerson method, this difficulty cannot arise.

Theorem 26.10 (Integrality theorem)

If the capacity function c takes on only integral values, then the maximum flow f produced by the Ford-Fulkerson method has the property that $|f|$ is an integer. Moreover, for all vertices u and v, the value of $f(u, v)$ is an integer.

Proof The proof is by induction on the number of iterations. We leave it as Exercise 26.3-2. ■

We can now prove the following corollary to Lemma 26.9.

Corollary 26.11

The cardinality of a maximum matching M in a bipartite graph G equals the value of a maximum flow f in its corresponding flow network G'.

Proof We use the nomenclature from Lemma 26.9. Suppose that M is a maximum matching in G and that the corresponding flow f in G' is not maximum. Then there is a maximum flow f' in G' such that $|f'| > |f|$. Since the capacities in G' are integer-valued, by Theorem 26.10, we can assume that f' is integer-valued. Thus, f' corresponds to a matching M' in G with cardinality $|M'| = |f'| > |f| = |M|$, contradicting our assumption that M is a maximum matching. In a similar manner, we can show that if f is a maximum flow in G', its corresponding matching is a maximum matching on G. ■

Thus, given a bipartite undirected graph G, we can find a maximum matching by creating the flow network G', running the Ford-Fulkerson method, and directly obtaining a maximum matching M from the integer-valued maximum flow f found. Since any matching in a bipartite graph has cardinality at most $\min(L, R) = O(V)$, the value of the maximum flow in G' is $O(V)$. We can therefore find a maximum matching in a bipartite graph in time $O(VE') = O(VE)$, since $|E'| = \Theta(E)$.

Exercises

26.3-1

Run the Ford-Fulkerson algorithm on the flow network in Figure 26.8(c) and show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

26.3-2

Prove Theorem 26.10.

26.3-3

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

26.3-4 ★

A ***perfect matching*** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the ***neighborhood*** of X as

$$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\} ,$$

that is, the set of vertices adjacent to some member of X. Prove ***Hall's theorem***: there exists a perfect matching in G if and only if $|A| \le |N(A)|$ for every subset $A \subseteq L$.

26.3-5 ★

We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is ***d-regular*** if every vertex $v \in V$ has degree exactly d. Every d-regular bipartite graph has $|L| = |R|$. Prove that every d-regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

★ 26.4 Push-relabel algorithms

In this section, we present the "push-relabel" approach to computing maximum flows. To date, many of the asymptotically fastest maximum-flow algorithms are push-relabel algorithms, and the fastest actual implementations of maximum-flow algorithms are based on the push-relabel method. Push-relabel methods also efficiently solve other flow problems, such as the minimum-cost flow problem. This section introduces Goldberg's "generic" maximum-flow algorithm, which has a simple implementation that runs in $O(V^2E)$ time, thereby improving upon the $O(VE^2)$ bound of the Edmonds-Karp algorithm. Section 26.5 refines the generic algorithm to obtain another push-relabel algorithm that runs in $O(V^3)$ time.

Push-relabel algorithms work in a more localized manner than the Ford-Fulkerson method. Rather than examine the entire residual network to find an augmenting path, push-relabel algorithms work on one vertex at a time, looking only at the vertex's neighbors in the residual network. Furthermore, unlike the Ford-Fulkerson method, push-relabel algorithms do not maintain the flow-conservation property throughout their execution. They do, however, maintain a ***preflow***, which is a function $f : V \times V \to \mathbb{R}$ that satisfies the capacity constraint and the following relaxation of flow conservation:

$$\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \ge 0$$

for all vertices $u \in V - \{s\}$. That is, the flow into a vertex may exceed the flow out. We call the quantity

$$e(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) \tag{26.14}$$

the ***excess flow*** into vertex u. The excess at a vertex is the amount by which the flow in exceeds the flow out. We say that a vertex $u \in V - \{s, t\}$ is ***overflowing*** if $e(u) > 0$.

27 Multithreaded Algorithms

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

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

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

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

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

Dynamic multithreaded programming

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

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

Our model for dynamic multithreading offers several important advantages:

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

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

27.1 The basics of dynamic multithreading

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

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

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

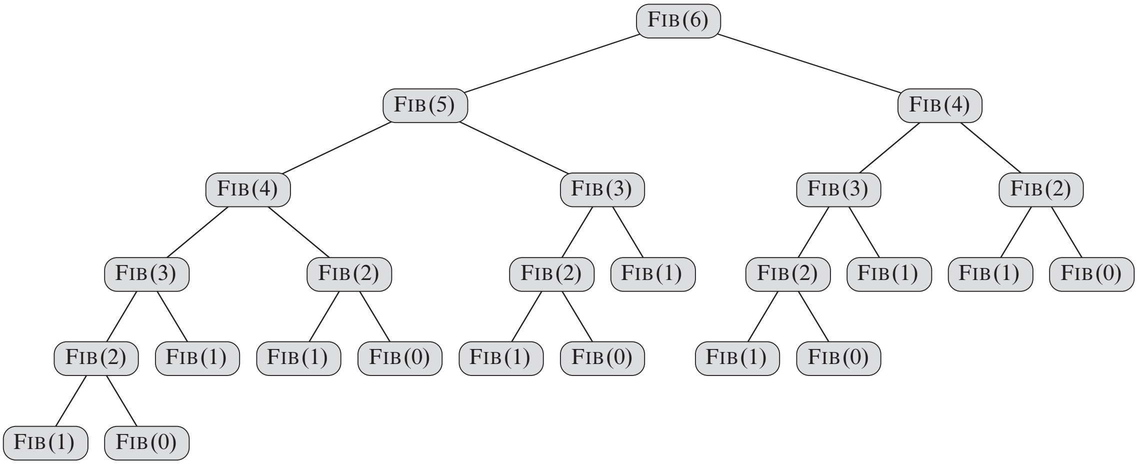


Figure 27.1 The tree of recursive procedure instances when computing Fib(6). Each instance of Fib with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

```
Fib($n$)
if $n \leq 1$
    return $n$
else $x = $ Fib($n - 1$)
    $y = $ Fib($n - 2$)
    return $x + y$
```

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

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

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

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

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

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

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

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

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

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

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

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

Nested parallelism occurs when the keyword **spawn** precedes a procedure call, as in line 3. The semantics of a spawn differs from an ordinary procedure call in that the procedure instance that executes the spawn—the ***parent***—may continue to execute in parallel with the spawned subroutine—its ***child***—instead of waiting for the child to complete, as would normally happen in a serial execution. In this case, while the spawned child is computing P-FIB$(n-1)$, the parent may go on to compute P-FIB$(n-2)$ in line 4 in parallel with the spawned child. Since the P-FIB procedure is recursive, these two subroutine calls themselves create nested parallelism, as do their children, thereby creating a potentially vast tree of subcomputations, all executing in parallel.

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

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

A model for multithreaded execution

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

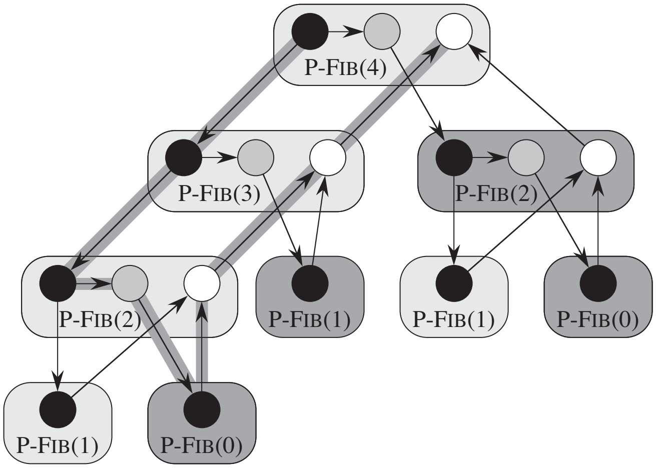


Figure 27.2 A directed acyclic graph representing the computation of P-FIB(4). Each circle represents one strand, with black circles representing either base cases or the part of the procedure (instance) up to the spawn of P-FIB$(n-1)$ in line 3, shaded circles representing the part of the procedure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

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

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

We can classify the edges of a computation dag to indicate the kind of dependencies between the various strands. A ***continuation edge*** (u, u'), drawn horizontally in Figure 27.2, connects a strand u to its successor u' within the same procedure instance. When a strand u spawns a strand v, the dag contains a ***spawn edge*** (u, v), which points downward in the figure. ***Call edges***, representing normal procedure calls, also point downward. Strand u spawning strand v differs from u calling v in that a spawn induces a horizontal continuation edge from u to the strand u' following u in its procedure, indicating that u' is free to execute at the same time as v, whereas a call induces no such edge. When a strand u returns to its calling procedure and x is the strand immediately following the next **sync** in the calling procedure, the computation dag contains ***return edge*** (u, x), which points upward. A computation starts with a single ***initial strand***—the black vertex in the procedure labeled P-FIB(4) in Figure 27.2—and ends with a single ***final strand***—the white vertex in the procedure labeled P-FIB(4).

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

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

Performance measures

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

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

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

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

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

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

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

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

The ratio T_1/T_∞ of the work to the span gives the ***parallelism*** of the multithreaded computation. We can view the parallelism from three perspectives. As a ratio, the parallelism denotes the average amount of work that can be performed in parallel for each step along the critical path. As an upper bound, the parallelism gives the maximum possible speedup that can be achieved on any number of processors. Finally, and perhaps most important, the parallelism provides a limit on the possibility of attaining perfect linear speedup. Specifically, once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup. To see this last point, suppose that $P > T_1/T_\infty$, in which case the span law implies that the speedup satisfies $T_1/T_P \le T_1/T_\infty < P$. Moreover, if the number P of processors in the ideal parallel computer greatly exceeds the parallelism—that is, if $P \gg T_1/T_\infty$—then $T_1/T_P \ll P$, so that the speedup is much less than the number of processors. In other words, the more processors we use beyond the parallelism, the less perfect the speedup.

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

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

Scheduling

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

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

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

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

Theorem 27.1

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

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

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

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

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

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

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

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

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

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

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

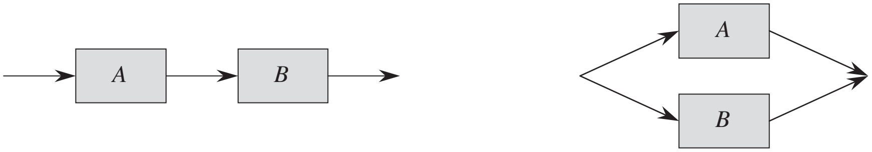
■

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

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

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

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



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

(a)

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

(b)

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

Analyzing multithreaded algorithms

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

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

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

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

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

The parallelism of P-FIB(n) is $T_1(n)/T_\infty(n) = \Theta(\phi^n/n)$, which grows dramatically as n gets large. Thus, on even the largest parallel computers, a modest value for n suffices to achieve near perfect linear speedup for P-FIB(n), because this procedure exhibits considerable parallel slackness.

Parallel loops

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

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

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

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

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

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

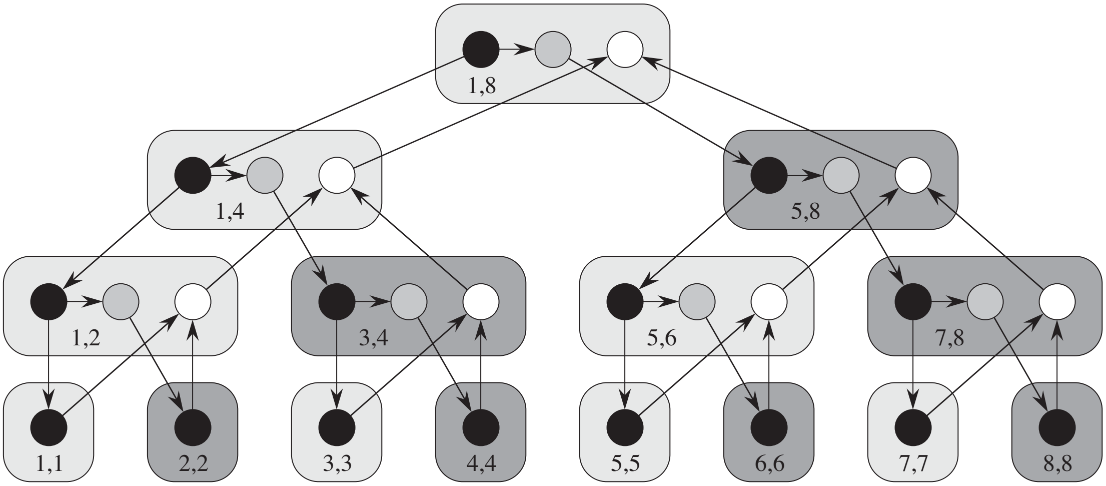


Figure 27.4 A dag representing the computation of MAT-VEC-MAIN-LOOP$(A, x, y, 8, 1, 8)$. The two numbers within each rounded rectangle give the values of the last two parameters (i and i' in the procedure header) in the invocation (spawn or call) of the procedure. The black circles represent strands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

```
MAT-VEC-MAIN-LOOP$(A, x, y, n, i, i')$
1  if $i == i'$
2      for $j = 1$ to $n$
3          $y_i = y_i + a_{ij} x_j$
4  else $mid = \lfloor (i + i')/2 \rfloor$
5      spawn MAT-VEC-MAIN-LOOP$(A, x, y, n, i, mid)$
6      MAT-VEC-MAIN-LOOP$(A, x, y, n, mid + 1, i')$
7      sync
```

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

To calculate the work $T_1(n)$ of MAT-VEC on an $n \times n$ matrix, we simply compute the running time of its serialization, which we obtain by replacing the **parallel for** loops with ordinary **for** loops. Thus, we have $T_1(n) = \Theta(n^2)$, because the quadratic running time of the doubly nested loops in lines 5–7 dominates. This analysis seems to ignore the overhead for recursive spawning in implementing the parallel loops, however. In fact, the overhead of recursive spawning does increase the work of a parallel loop compared with that of its serialization, but not asymptotically. To see why, observe that since the tree of recursive procedure instances is a full binary tree, the number of internal nodes is 1 fewer than the number of leaves (see Exercise B.5-3). Each internal node performs constant work to divide the iteration range, and each leaf corresponds to an iteration of the loop, which takes at least constant time ($\Theta(n)$ time in this case). Thus, we can amortize the overhead of recursive spawning against the work of the iterations, contributing at most a constant factor to the overall work.

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

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

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

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

Race conditions

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

Race conditions are the bane of concurrency. Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured several others, and the North American Blackout of 2003, which left over 50 million people without power. These pernicious bugs are notoriously hard to find. You can run tests in the lab for days without a failure only to discover that your software sporadically crashes in the field.

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

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

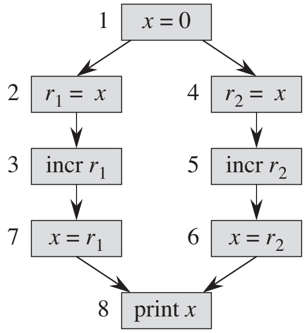
After initializing x to 0 in line 1, RACE-EXAMPLE creates two parallel strands, each of which increments x in line 3. Although it might seem that RACE-EXAMPLE should always print the value 2 (its serialization certainly does), it could instead print the value 1. Let's see how this anomaly might occur.

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

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

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

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



(a)

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

(b)

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

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

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

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

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

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

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

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

A chess lesson

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

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

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

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

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

Exercises

27.1-1

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

27.1-2

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

27.1-3

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

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

27.1-4

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

27.1-5

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

27.1-6

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

27.1-7

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

```
P-TRANSPOSE($A$)
1  $n = A.rows$
2  **parallel for** $j = 2$ **to** $n$
3      **parallel for** $i = 1$ **to** $j - 1$
4          exchange $a_{ij}$ with $a_{ji}$
```

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

27.1-8

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

27.1-9

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

27.2 Multithreaded matrix multiplication

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

Multithreaded matrix multiplication

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

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
1  $n = A.rows$
2  let $C$ be a new $n \times n$ matrix
3  parallel for $i = 1$ to $n$
4      parallel for $j = 1$ to $n$
5          $c_{ij} = 0$
6          for $k = 1$ to $n$
7              $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8  return $C$
```

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

A divide-and-conquer multithreaded algorithm for matrix multiplication

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

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

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

Then, we can write the matrix product as

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

Thus, to multiply two $n \times n$ matrices, we perform eight multiplications of $n/2 \times n/2$ matrices and one addition of $n \times n$ matrices. The following pseudocode implements this divide-and-conquer strategy using nested parallelism. Unlike the Square-Matrix-Multiply-Recursive procedure on which it is based, P-Matrix-Multiply-Recursive takes the output matrix as a parameter to avoid allocating matrices unnecessarily.

```
P-Matrix-Multiply-Recursive($C, A, B$)
 1  $n = A.rows$
 2  if $n == 1$
 3      $c_{11} = a_{11} b_{11}$
 4  else let $T$ be a new $n \times n$ matrix
 5      partition $A$, $B$, $C$, and $T$ into $n/2 \times n/2$ submatrices
            $A_{11}, A_{12}, A_{21}, A_{22}$; $B_{11}, B_{12}, B_{21}, B_{22}$; $C_{11}, C_{12}, C_{21}, C_{22}$;
            and $T_{11}, T_{12}, T_{21}, T_{22}$; respectively
 6      spawn P-Matrix-Multiply-Recursive($C_{11}, A_{11}, B_{11}$)
 7      spawn P-Matrix-Multiply-Recursive($C_{12}, A_{11}, B_{12}$)
 8      spawn P-Matrix-Multiply-Recursive($C_{21}, A_{21}, B_{11}$)
 9      spawn P-Matrix-Multiply-Recursive($C_{22}, A_{21}, B_{12}$)
10      spawn P-Matrix-Multiply-Recursive($T_{11}, A_{12}, B_{21}$)
11      spawn P-Matrix-Multiply-Recursive($T_{12}, A_{12}, B_{22}$)
12      spawn P-Matrix-Multiply-Recursive($T_{21}, A_{22}, B_{21}$)
13      P-Matrix-Multiply-Recursive($T_{22}, A_{22}, B_{22}$)
14      sync
15      parallel for $i = 1$ to $n$
16          parallel for $j = 1$ to $n$
17              $c_{ij} = c_{ij} + t_{ij}$
```

Line 3 handles the base case, where we are multiplying 1×1 matrices. We handle the recursive case in lines 4–17. We allocate a temporary matrix T in line 4, and line 5 partitions each of the matrices A, B, C, and T into $n/2 \times n/2$ submatrices. (As with Square-Matrix-Multiply-Recursive on page 77, we gloss over the minor issue of how to use index calculations to represent submatrix sections of a matrix.) The recursive call in line 6 sets the submatrix C_{11} to the submatrix product $A_{11}B_{11}$, so that C_{11} equals the first of the two terms that form its sum in equation (27.6). Similarly, lines 7–9 set C_{12}, C_{21}, and C_{22} to the first of the two terms that equal their sums in equation (27.6). Line 10 sets the submatrix T_{11} to the submatrix product $A_{12}B_{21}$, so that T_{11} equals the second of the two terms that form C_{11}'s sum. Lines 11–13 set T_{12}, T_{21}, and T_{22} to the second of the two terms that form the sums of C_{12}, C_{21}, and C_{22}, respectively. The first seven recursive calls are spawned, and the last one runs in the main strand. The **sync** statement in line 14 ensures that all the submatrix products in lines 6–13 have been computed, after which we add the products from T into C in using the doubly nested **parallel for** loops in lines 15–17.

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

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

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

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

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

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

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

Multithreading Strassen's method

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

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices, as in equation (27.6). This step takes $\Theta(1)$ work and span by index calculation.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$, each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1. We can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span by using doubly nested **parallel for** loops.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively spawn the computation of seven $n/2 \times n/2$ matrix products $P_1, P_2, \ldots, P_7$.

4. Compute the desired submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ of the result matrix C by adding and subtracting various combinations of the P_i matrices, once again using doubly nested **parallel for** loops. We can compute all four submatrices with $\Theta(n^2)$ work and $\Theta(\lg n)$ span.

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

Exercises

27.2-1

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

27.2-2

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

27.2-3

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

27.2-4

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

27.2-5

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

27.2-6

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

27.3 Multithreaded merge sort

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

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

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

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

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

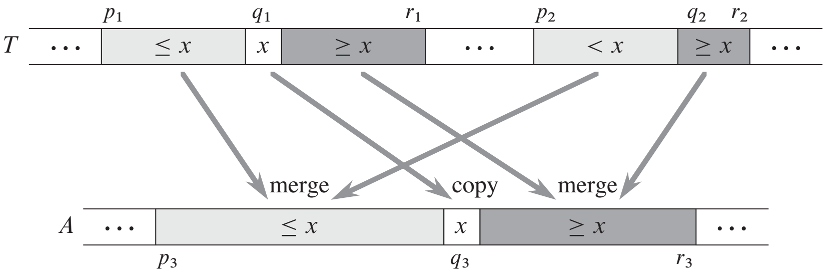


Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \mathinner{..} r_1]$ and $T[p_2 \mathinner{..} r_2]$ into the subarray $A[p_3 \mathinner{..} r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \mathinner{..} r_1]$ and q_2 be the place in $T[p_2 \mathinner{..} r_2]$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

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

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

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

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

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

We first find the middle element $x = T[q_1]$ of the subarray $T[p_1 \mathinner{..} r_1]$, where $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$. Because the subarray is sorted, x is a median of $T[p_1 \mathinner{..} r_1]$: every element in $T[p_1 \mathinner{..} q_1 - 1]$ is no more than x, and every element in $T[q_1 + 1 \mathinner{..} r_1]$ is no less than x. We then use binary search to find the index q_2 in the subarray $T[p_2 \,..\, r_2]$ so that the subarray would still be sorted if we inserted x between $T[q_2 - 1]$ and $T[q_2]$.

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

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

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

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

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

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

Here is the pseudocode:

```
BINARY-SEARCH$(x, T, p, r)$
low = $p$
high = $\max(p, r + 1)$
while low < high
    mid = $\lfloor (low + high)/2 \rfloor$
    if $x \leq T[mid]$
        high = mid
    else low = mid + 1
return high
```

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

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

```
P-MERGE($T, p_1, r_1, p_2, r_2, A, p_3$)
 1  $n_1 = r_1 - p_1 + 1$
 2  $n_2 = r_2 - p_2 + 1$
 3  if $n_1 < n_2$                // ensure that $n_1 \geq n_2$
 4      exchange $p_1$ with $p_2$
 5      exchange $r_1$ with $r_2$
 6      exchange $n_1$ with $n_2$
 7  if $n_1 == 0$                // both empty?
 8      return
 9  else $q_1 = \lfloor (p_1 + r_1)/2 \rfloor$
10      $q_2 =$ BINARY-SEARCH($T[q_1], T, p_2, r_2$)
11      $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$
12      $A[q_3] = T[q_1]$
13      spawn P-MERGE($T, p_1, q_1 - 1, p_2, q_2 - 1, A, p_3$)
14      P-MERGE($T, q_1 + 1, r_1, q_2, r_2, A, q_3 + 1$)
15      sync
```

The P-Merge procedure works as follows. Lines 1–2 compute the lengths n_1 and n_2 of the subarrays $T[p_1 \mathinner{..} r_1]$ and $T[p_2 \mathinner{..} r_2]$, respectively. Lines 3–6 enforce the assumption that $n_1 \geq n_2$. Line 7 tests for the base case, where the subarray $T[p_1 \mathinner{..} r_1]$ is empty (and hence so is $T[p_2 \mathinner{..} r_2]$), in which case we simply return. Lines 9–15 implement the divide-and-conquer strategy. Line 9 computes the midpoint of $T[p_1 \mathinner{..} r_1]$, and line 10 finds the point q_2 in $T[p_2 \mathinner{..} r_2]$ such that all elements in $T[p_2 \mathinner{..} q_2 - 1]$ are less than $T[q_1]$ (which corresponds to x) and all the elements in $T[q_2 \mathinner{..} p_2]$ are at least as large as $T[q_1]$. Line 11 computes the index q_3 of the element that divides the output subarray $A[p_3 \,..\, r_3]$ into $A[p_3 \,..\, q_3 - 1]$ and $A[q_3+1 \,..\, r_3]$, and then line 12 copies $T[q_1]$ directly into $A[q_3]$.

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

Analysis of multithreaded merging

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

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

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

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

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

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

We shall first derive a recurrence for the worst-case work. The binary search in line 10 costs $\Theta(\lg n)$ in the worst case, which dominates the other work outside of the recursive calls. For the recursive calls, observe that although the recursive calls in lines 13 and 14 might merge different numbers of elements, together the two recursive calls merge at most n elements (actually $n-1$ elements, since $T[q_1]$ does not participate in either recursive call). Moreover, as we saw in analyzing the span, a recursive call operates on at most $3n/4$ elements. We therefore obtain the recurrence

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

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

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

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

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

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

Multithreaded merge sort

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

```
P-MERGE-SORT$(A, p, r, B, s)$
 1  $n = r - p + 1$
 2  if $n == 1$
 3      $B[s] = A[p]$
 4  else let $T[1 \mathinner{.\,.} n]$ be a new array
 5      $q = \lfloor (p + r)/2 \rfloor$
 6      $q' = q - p + 1$
 7      spawn P-MERGE-SORT$(A, p, q, T, 1)$
 8      P-MERGE-SORT$(A, q + 1, r, T, q' + 1)$
 9      sync
10      P-MERGE$(T, 1, q', q' + 1, n, B, s)$
```

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

Analysis of multithreaded merge sort

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

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

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

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

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

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

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

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

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

Exercises

27.3-1

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

27.3-2

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

27.3-3

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

27.3-4

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

27.3-5 ★

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

27.3-6 ★

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

Problems

27-1 Implementing parallel loops using nested parallelism

Consider the following multithreaded algorithm for performing pairwise addition on n-element arrays $A[1 \,..\, n]$ and $B[1 \,..\, n]$, storing the sums in $C[1 \,..\, n]$:

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

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

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

```
SUM-ARRAYS'($A, B, C$)
$n = A.length$
$grain\text{-}size = ?$                // to be determined
$r = \lceil n / grain\text{-}size \rceil$
for $k = 0$ to $r - 1$
    spawn ADD-SUBARRAY($A, B, C, k \cdot grain\text{-}size + 1,$
                                    $\min((k + 1) \cdot grain\text{-}size, n)$)
sync
```

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

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

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

27-2 Saving temporary space in matrix multiplication

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

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

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

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

27-3 Multithreaded matrix algorithms

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

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

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

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

27-4 Multithreading reductions and prefix computations

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

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

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

```
REDUCE($x, i, j$)
$y = x[i]$
for $k = i + 1$ to $j$
    $y = y \otimes x[k]$
return $y$
```

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

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

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

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

```
SCAN($x$)
$n = x.length$
let $y[1 \mathinner{..} n]$ be a new array
$y[1] = x[1]$
for $i = 2$ to $n$
    $y[i] = y[i-1] \otimes x[i]$
return $y$
```

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

```
P-Scan-1($x$)
n = x.length
let $y[1..n]$ be a new array
P-Scan-1-Aux($x, y, 1, n$)
return $y$
```

```
P-Scan-1-Aux($x, y, i, j$)
parallel for $l = i$ to $j$
    $y[l] =$ P-Reduce($x, 1, l$)
```

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

By using nested parallelism, we can obtain a more efficient ⊗-prefix computation:

```
P-Scan-2($x$)
$n = x.length$
let $y[1..n]$ be a new array
P-Scan-2-Aux($x, y, 1, n$)
return $y$
```

```
P-Scan-2-Aux($x, y, i, j$)
if $i == j$
    $y[i] = x[i]$
else $k = \lfloor (i + j)/2 \rfloor$
    spawn P-Scan-2-Aux($x, y, i, k$)
    P-Scan-2-Aux($x, y, k + 1, j$)
    sync
    parallel for $l = k + 1$ to $j$
        $y[l] = y[k] \otimes y[l]$
```

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

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

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

```
P-SCAN-UP($x, t, i, j$)
1  if $i == j$
2      return $x[i]$
3  else
4      $k = \lfloor (i + j)/2 \rfloor$
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($\nu, x, t, y, i, j$)
1  if $i == j$
2      $y[i] = \nu \otimes x[i]$
3  else
4      $k = \lfloor (i + j)/2 \rfloor$
5      spawn P-SCAN-DOWN(______, $x, t, y, i, k$)      // fill in the blank
6      P-SCAN-DOWN(______, $x, t, y, k + 1, j$)        // fill in the blank
7      sync
```

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

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

27-5 Multithreading a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries does not change during the computation and is called a ***stencil***. For example, Section 15.4 presents a stencil algorithm to compute a longest common subsequence, where the value in entry $c[i, j]$ depends only on the values in $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, as well as the elements x_i and y_j within the two sequences given as inputs. The input sequences are fixed, but the algorithm fills in the two-dimensional array c so that it computes entry $c[i, j]$ after computing all three entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$.

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

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

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

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

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

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

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer $b \geq 2$. 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 $b \geq 2$. (*Hint:* For this last argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \geq 2$.)

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

27-6 Randomized multithreaded algorithms

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

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

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

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

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

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

Chapter notes

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

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

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

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

29 Linear Programming

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

A political problem

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

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

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

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

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

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

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

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

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

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

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

and

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

Any setting of the variables x_1, x_2, x_3, x_4 that satisfies inequalities (29.1)–(29.3) yields a strategy that wins a sufficient number of each type of vote. In order to keep costs as small as possible, you would like to minimize the amount spent on advertising. That is, you want to minimize the expression

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

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

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

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

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

The solution of this linear program yields your optimal strategy.

General linear programs

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

$$f(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n = \sum_{j=1}^{n} a_jx_j \ .$$

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

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

and

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

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

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

An overview of linear programming

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

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

$$\begin{array}{lrcrcrl}
\text{maximize} & x_1 & + & x_2 & & & \qquad (29.11) \\
\text{subject to} & & & & & & \\
& 4x_1 & - & x_2 & \leq & 8 & \qquad (29.12) \\
& 2x_1 & + & x_2 & \leq & 10 & \qquad (29.13) \\
& 5x_1 & - & 2x_2 & \geq & -2 & \qquad (29.14) \\
& \multicolumn{3}{c}{x_1, x_2} & \geq & 0\ . & \qquad (29.15)
\end{array}$$

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

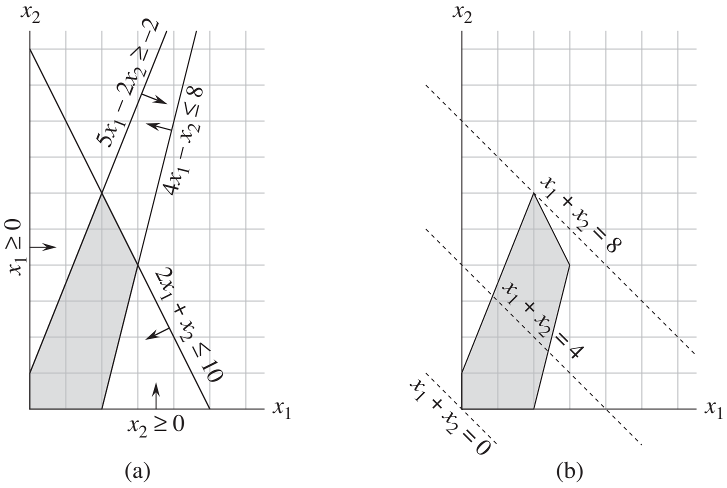


Figure 29.2 **(a)** The linear program given in (29.12)–(29.15). Each constraint is represented by a line and a direction. The intersection of the constraints, which is the feasible region, is shaded. **(b)** The dotted lines show, respectively, the points for which the objective value is 0, 4, and 8.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

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

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

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

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

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

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

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

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

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

Applications of linear programming

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

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

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

Algorithms for linear programming

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

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

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

29.1 Standard and slack forms

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

Standard form

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

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

subject to

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

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

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

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

subject to

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

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

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

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

Converting linear programs into standard form

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

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

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

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

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

$$\begin{array}{llrcrcl}
\text{minimize} & -2x_1 & + & 3x_2 & & \\
\text{subject to} & & & & & \\
& x_1 & + & x_2 & = & 7 \\
& x_1 & - & 2x_2 & \leq & 4 \\
& x_1 & & & \geq & 0 \; ,
\end{array}$$

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

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

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

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

$$
\begin{array}{lrcrcrcl}
\text{maximize} & 2x_1 & - & 3x'_2 & + & 3x''_2 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x'_2 & - & x''_2 & = & 7 \\
& x_1 & - & 2x'_2 & + & 2x''_2 & \le & 4 \\
& & & x_1, x'_2, x''_2 & & & \ge & 0 \; .
\end{array}
\qquad (29.22)
$$

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

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

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

is equivalent to

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

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

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

$$\begin{array}{lrcrcrcr}
\text{maximize} & 2x_1 & - & 3x_2' & + & 3x_2'' & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2' & - & x_2'' & \leq & 7 \\
& x_1 & + & x_2' & - & x_2'' & \geq & 7 \\
& x_1 & - & 2x_2' & + & 2x_2'' & \leq & 4 \\
& \multicolumn{5}{c}{x_1, x_2', x_2''} & \geq & 0 \ .
\end{array} \tag{29.23}$$

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

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

subject to

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

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

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

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

Converting linear programs into slack form

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

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

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

$$\begin{aligned} s &= b_i - \sum_{j=1}^{n} a_{ij} x_j \;, && (29.30) \\ s &\geq 0 \;. && (29.31) \end{aligned}$$

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

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

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

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

$$\begin{array}{lrcrcrcrcr} \text{maximize} & & & & & 2x_1 & - & 3x_2 & + & 3x_3 & (29.33) \\ \text{subject to} & & & & & & & & & & \\ & x_4 & = & 7 & - & x_1 & - & x_2 & + & x_3 & (29.34) \\ & x_5 & = & -7 & + & x_1 & + & x_2 & - & x_3 & (29.35) \\ & x_6 & = & 4 & - & x_1 & + & 2x_2 & - & 2x_3 & (29.36) \\ & \multicolumn{5}{r}{x_1, x_2, x_3, x_4, x_5, x_6} & \geq & 0 & . & & (29.37) \end{array}$$

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

For linear programs that satisfy these conditions, we shall sometimes omit the words "maximize" and "subject to," as well as the explicit nonnegativity constraints. We shall also use the variable z to denote the value of the objective function. We call the resulting format ***slack form***. If we write the linear program given in (29.33)–(29.37) in slack form, we obtain

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

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

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

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

For example, in the slack form

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

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

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

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

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

Exercises

29.1-1

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

29.1-2

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

29.1-3

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

29.1-4

Convert the following linear program into standard form:

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

29.1-5

Convert the following linear program into slack form:

$$\begin{array}{lrcrcrcl}
\text{maximize} & 2x_1 & & & - & 6x_3 & & \\
\text{subject to} & & & & & & & \\
& x_1 & + & x_2 & - & x_3 & \leq & 7 \\
& 3x_1 & - & x_2 & & & \geq & 8 \\
& -x_1 & + & 2x_2 & + & 2x_3 & \geq & 0 \\
& & & x_1, x_2, x_3 & & & \geq & 0 \; .
\end{array}$$

What are the basic and nonbasic variables?

29.1-6

Show that the following linear program is infeasible:

$$\begin{array}{lrcrcr}
\text{maximize} & 3x_1 & - & 2x_2 & & \\
\text{subject to} & & & & & \\
& x_1 & + & x_2 & \leq & 2 \\
& -2x_1 & - & 2x_2 & \leq & -10 \\
& & x_1, x_2 & & \geq & 0 \; .
\end{array}$$

29.1-7

Show that the following linear program is unbounded:

$$\begin{array}{lrcrcr}
\text{maximize} & x_1 & - & x_2 & & \\
\text{subject to} & & & & & \\
& -2x_1 & + & x_2 & \leq & -1 \\
& -x_1 & - & 2x_2 & \leq & -2 \\
& & x_1, x_2 & & \geq & 0 \; .
\end{array}$$

29.1-8

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

29.1-9

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

29.2 Formulating problems as linear programs

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

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

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

Shortest paths

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

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

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

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

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

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

Maximum flow

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

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

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

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

Minimum-cost flow

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

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

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

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

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

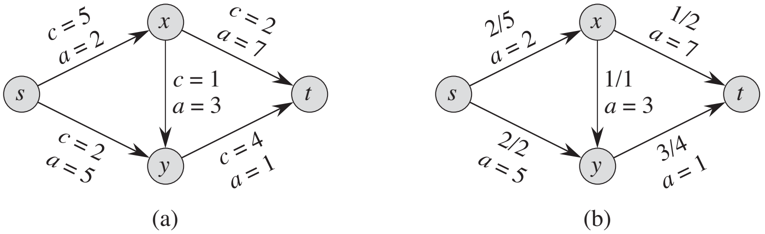


Figure 29.3 **(a)** An example of a minimum-cost-flow problem. We denote the capacities by c and the costs by a. Vertex s is the source and vertex t is the sink, and we wish to send 4 units of flow from s to t. **(b)** A solution to the minimum-cost flow problem in which 4 units of flow are sent from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

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

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

subject to

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

Multicommodity flow

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

This example is an instance of a ***multicommodity-flow problem***. In this problem, we are again given a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \ge 0$. As in the maximum-flow problem, we implicitly assume that $c(u, v) = 0$ for $(u, v) \notin E$, and that the graph has no antiparallel edges. In addition, we are given k different commodities, $K_1, K_2, \ldots, K_k$, where we specify commodity i by the triple $K_i = (s_i, t_i, d_i)$. Here, vertex s_i is the source of commodity i, vertex t_i is the sink of commodity i, and d_i is the demand for commodity i, which is the desired flow value for the commodity from s_i to t_i. We define a flow for commodity i, denoted by f_i, (so that f_{iuv} is the flow of commodity i from vertex u to vertex v) to be a real-valued function that satisfies the flow-conservation and capacity constraints. We now define f_{uv}, the ***aggregate flow***, to be the sum of the various commodity flows, so that $f_{uv} = \sum_{i=1}^{k} f_{iuv}$. The aggregate flow on edge (u, v) must be no more than the capacity of edge (u, v). We are not trying to minimize any objective function in this problem; we need only determine whether such a flow exists. Thus, we write a linear program with a "null" objective function:

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

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

Exercises

29.2-1

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

29.2-2

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

29.2-3

In the single-source shortest-paths problem, we want to find the shortest-path weights from a source vertex s to all vertices $v \in V$. Given a graph G, write a linear program for which the solution has the property that d_v is the shortest-path weight from s to v for each vertex $v \in V$.

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

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

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

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

29.3 The simplex algorithm

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

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

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

An example of the simplex algorithm

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \qquad (29.64) \\ x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \qquad (29.65) \\ x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \qquad (29.66) \\ x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2} \; . \qquad (29.67) \end{aligned}$$

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

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

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

Continuing the example, we wish to find a new variable whose value we wish to increase. We do not want to increase x_6, since as its value increases, the objective value decreases. We can attempt to increase either x_2 or x_3; let us choose x_3. How far can we increase x_3 without violating any of the constraints? Constraint (29.65) limits it to 18, constraint (29.66) limits it to $42/5$, and constraint (29.67) limits it to $3/2$. The third constraint is again the tightest one, and therefore we rewrite the third constraint so that x_3 is on the left-hand side and x_5 is on the right-hand side. We then substitute this new equation, $x_3 = 3/2 - 3x_2/8 - x_5/4 + x_6/8$, into equations (29.64)–(29.66) and obtain the new, but equivalent, system

$$
\begin{aligned}
z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} && (29.68)\\
x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} && (29.69)\\
x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} && (29.70)\\
x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16} \,. && (29.71)
\end{aligned}
$$

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

$$
\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} && (29.72)\\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} && (29.73)\\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} && (29.74)\\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2} \,. && (29.75)
\end{aligned}
$$

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

Pivoting

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

```
PIVOT$(N, B, A, b, c, \nu, l, e)$
 1  // Compute the coefficients of the equation for new basic variable $x_e$.
 2  let $\widehat{A}$ be a new $m \times n$ matrix
 3  $\widehat{b}_e = b_l / a_{le}$
 4  for each $j \in N - \{e\}$
 5      $\widehat{a}_{ej} = a_{lj} / a_{le}$
 6  $\widehat{a}_{el} = 1/a_{le}$
 7  // Compute the coefficients of the remaining constraints.
 8  for each $i \in B - \{l\}$
 9      $\widehat{b}_i = b_i - a_{ie}\widehat{b}_e$
10      for each $j \in N - \{e\}$
11          $\widehat{a}_{ij} = a_{ij} - a_{ie}\widehat{a}_{ej}$
12      $\widehat{a}_{il} = -a_{ie}\widehat{a}_{el}$
13  // Compute the objective function.
14  $\widehat{\nu} = \nu + c_e\widehat{b}_e$
15  for each $j \in N - \{e\}$
16      $\widehat{c}_j = c_j - c_e\widehat{a}_{ej}$
17  $\widehat{c}_l = -c_e\widehat{a}_{el}$
18  // Compute new sets of basic and nonbasic variables.
19  $\widehat{N} = N - \{e\} \cup \{l\}$
20  $\widehat{B} = B - \{l\} \cup \{e\}$
21  return $(\widehat{N}, \widehat{B}, \widehat{A}, \widehat{b}, \widehat{c}, \widehat{\nu})$
```

PIVOT works as follows. Lines 3–6 compute the coefficients in the new equation for x_e by rewriting the equation that has x_l on the left-hand side to instead have x_e on the left-hand side. Lines 8–12 update the remaining equations by substituting the right-hand side of this new equation for each occurrence of x_e. Lines 14–17 do the same substitution for the objective function, and lines 19 and 20 update the sets of nonbasic and basic variables. Line 21 returns the new slack form. As given, if $a_{le} = 0$, PIVOT would cause an error by dividing by 0, but as we shall see in the proofs of Lemmas 29.2 and 29.12, we call PIVOT only when $a_{le} \neq 0$.

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

Lemma 29.1

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

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

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

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

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

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

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

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

which proves the third statement. ■

The formal simplex algorithm

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

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

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

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

```
SIMPLEX$(A, b, c)$
 1  $(N, B, A, b, c, \nu) = $ INITIALIZE-SIMPLEX$(A, b, c)$
 2  let $\Delta$ be a new vector of length $n$
 3  while some index $j \in N$ has $c_j > 0$
 4      choose an index $e \in N$ for which $c_e > 0$
 5      for each index $i \in B$
 6          if $a_{ie} > 0$
 7              $\Delta_i = b_i / a_{ie}$
 8          else $\Delta_i = \infty$
 9      choose an index $l \in B$ that minimizes $\Delta_i$
10      if $\Delta_l == \infty$
11          return "unbounded"
12      else $(N, B, A, b, c, \nu) = $ PIVOT$(N, B, A, b, c, \nu, l, e)$
13  for $i = 1$ to $n$
14      if $i \in B$
15          $\bar{x}_i = b_i$
16      else $\bar{x}_i = 0$
17  return $(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$
```

The SIMPLEX procedure works as follows. In line 1, it calls the procedure INITIALIZE-SIMPLEX(A, b, c), described above, which either determines that the linear program is infeasible or returns a slack form for which the basic solution is feasible. The **while** loop of lines 3–12 forms the main part of the algorithm. If all coefficients in the objective function are negative, then the **while** loop terminates. Otherwise, line 4 selects a variable x_e, whose coefficient in the objective function is positive, as the entering variable. Although we may choose any such variable as the entering variable, we assume that we use some prespecified deterministic rule. Next, lines 5–9 check each constraint and pick the one that most severely limits the amount by which we can increase x_e without violating any of the nonnegativity constraints; the basic variable associated with this constraint is x_l. Again, we are free to choose one of several variables as the leaving variable, but we assume that we use some prespecified deterministic rule. If none of the constraints limits the amount by which the entering variable can increase, the algorithm returns "unbounded" in line 11. Otherwise, line 12 exchanges the roles of the entering and leaving variables by calling PIVOT$(N, B, A, b, c, \nu, l, e)$, as described above. Lines 13–16 compute a solution $\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$ for the original linear-programming variables by setting all the nonbasic variables to 0 and each basic variable $\bar{x}_i$ to b_i, and line 17 returns these values.

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

Lemma 29.2

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

Proof We use the following three-part loop invariant:

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

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

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

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

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

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

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

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

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

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

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

we have

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Termination

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

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

Indeed, this situation can occur. Consider the linear program

$$\begin{array}{rcrcrcrcr} z & = & & & x_1 & + & x_2 & + & x_3 \\ x_4 & = & 8 & - & x_1 & - & x_2 & & \\ x_5 & = & & & & & x_2 & - & x_3 \; . \end{array}$$

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

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

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

$$\begin{array}{rcrcrcrcr} z & = & 8 & + & x_2 & - & x_4 & - & x_5 \\ x_1 & = & 8 & - & x_2 & - & x_4 & & \\ x_3 & = & & & x_2 & & & - & x_5 \; . \end{array}$$

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

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

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

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

Lemma 29.3

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

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

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

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

A particular linear program has many different slack forms; recall that each slack form has the same set of feasible and optimal solutions as the original linear program. We now show that the slack form of a linear program is uniquely determined by the set of basic variables. That is, given the set of basic variables, a unique slack form (unique set of coefficients and right-hand sides) is associated with those basic variables.

Lemma 29.4

Let (A, b, c) be a linear program in standard form. Given a set B of basic variables, the associated slack form is uniquely determined.

Proof Assume for the purpose of contradiction that there are two different slack forms with the same set B of basic variables. The slack forms must also have identical sets $N = \{1, 2, \ldots, n + m\} - B$ of nonbasic variables. We write the first slack form as

$$z = \nu + \sum_{j \in N} c_j x_j \tag{29.79}$$

$$x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad \text{for } i \in B \ , \tag{29.80}$$

and the second as

$$z = \nu' + \sum_{j \in N} c'_j x_j \tag{29.81}$$

$$x_i = b'_i - \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B \ . \tag{29.82}$$

Consider the system of equations formed by subtracting each equation in line (29.82) from the corresponding equation in line (29.80). The resulting system is

$$0 = (b_i - b'_i) - \sum_{j \in N} (a_{ij} - a'_{ij}) x_j \quad \text{for } i \in B$$

or, equivalently,

$$\sum_{j \in N} a_{ij} x_j = (b_i - b'_i) + \sum_{j \in N} a'_{ij} x_j \quad \text{for } i \in B \; .$$

Now, for each $i \in B$, apply Lemma 29.3 with $\alpha_j = a_{ij}$, $\beta_j = a'_{ij}$, $\gamma = b_i - b'_i$, and $I = N$. Since $\alpha_i = \beta_i$, we have that $a_{ij} = a'_{ij}$ for each $j \in N$, and since $\gamma = 0$, we have that $b_i = b'_i$. Thus, for the two slack forms, A and b are identical to A' and b'. Using a similar argument, Exercise 29.3-1 shows that it must also be the case that $c = c'$ and $\nu = \nu'$, and hence that the slack forms must be identical. ■

We can now show that cycling is the only possible reason that SIMPLEX might not terminate.

Lemma 29.5
If SIMPLEX fails to terminate in at most $\binom{n+m}{m}$ iterations, then it cycles.

Proof By Lemma 29.4, the set B of basic variables uniquely determines a slack form. There are $n + m$ variables and $|B| = m$, and therefore, there are at most $\binom{n+m}{m}$ ways to choose B. Thus, there are only at most $\binom{n+m}{m}$ unique slack forms. Therefore, if SIMPLEX runs for more than $\binom{n+m}{m}$ iterations, it must cycle. ■

Cycling is theoretically possible, but extremely rare. We can prevent it by choosing the entering and leaving variables somewhat more carefully. One option is to perturb the input slightly so that it is impossible to have two solutions with the same objective value. Another option is to break ties by always choosing the variable with the smallest index, a strategy known as ***Bland's rule***. We omit the proof that these strategies avoid cycling.

Lemma 29.6
If lines 4 and 9 of SIMPLEX always break ties by choosing the variable with the smallest index, then SIMPLEX must terminate. ■

We conclude this section with the following lemma.

Lemma 29.7

Assuming that INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible, SIMPLEX either reports that a linear program is unbounded, or it terminates with a feasible solution in at most $\binom{n+m}{m}$ iterations.

Proof Lemmas 29.2 and 29.6 show that if INITIALIZE-SIMPLEX returns a slack form for which the basic solution is feasible, SIMPLEX either reports that a linear program is unbounded, or it terminates with a feasible solution. By the contrapositive of Lemma 29.5, if SIMPLEX terminates with a feasible solution, then it terminates in at most $\binom{n+m}{m}$ iterations. ■

Exercises

29.3-1

Complete the proof of Lemma 29.4 by showing that it must be the case that $c = c'$ and $\nu = \nu'$.

29.3-2

Show that the call to PIVOT in line 12 of SIMPLEX never decreases the value of ν.

29.3-3

Prove that the slack form given to the PIVOT procedure and the slack form that the procedure returns are equivalent.

29.3-4

Suppose we convert a linear program (A, b, c) in standard form to slack form. Show that the basic solution is feasible if and only if $b_i \geq 0$ for $i = 1, 2, \ldots, m$.

29.3-5

Solve the following linear program using SIMPLEX:

$$\begin{array}{lrcrcr}
\text{maximize} & 18x_1 & + & 12.5x_2 & & \\
\text{subject to} & & & & & \\
& x_1 & + & x_2 & \leq & 20 \\
& x_1 & & & \leq & 12 \\
& & & x_2 & \leq & 16 \\
& x_1, x_2 & & & \geq & 0 \; .
\end{array}$$

29.3-6

Solve the following linear program using SIMPLEX:

$$
\begin{array}{llrcrcl}
\text{maximize} & 5x_1 & - & 3x_2 & & & \\
\text{subject to} & & & & & & \\
& x_1 & - & x_2 & \leq & 1 & \\
& 2x_1 & + & x_2 & \leq & 2 & \\
& x_1, x_2 & & & \geq & 0 & .
\end{array}
$$

29.3-7

Solve the following linear program using SIMPLEX:

$$
\begin{array}{llrcrcrcl}
\text{minimize} & x_1 & + & x_2 & + & x_3 & & & \\
\text{subject to} & & & & & & & & \\
& 2x_1 & + & 7.5x_2 & + & 3x_3 & \geq & 10000 & \\
& 20x_1 & + & 5x_2 & + & 10x_3 & \geq & 30000 & \\
& x_1, x_2, x_3 & & & & & \geq & 0 & .
\end{array}
$$

29.3-8

In the proof of Lemma 29.5, we argued that there are at most $\binom{m+n}{n}$ ways to choose a set B of basic variables. Give an example of a linear program in which there are strictly fewer than $\binom{m+n}{n}$ ways to choose the set B.

29.4 Duality

We have proven that, under certain assumptions, SIMPLEX terminates. We have not yet shown that it actually finds an optimal solution to a linear program, however. In order to do so, we introduce a powerful concept called ***linear-programming duality***.

Duality enables us to prove that a solution is indeed optimal. We saw an example of duality in Chapter 26 with Theorem 26.6, the max-flow min-cut theorem. Suppose that, given an instance of a maximum-flow problem, we find a flow f with value $|f|$. How do we know whether f is a maximum flow? By the max-flow min-cut theorem, if we can find a cut whose value is also $|f|$, then we have verified that f is indeed a maximum flow. This relationship provides an example of duality: given a maximization problem, we define a related minimization problem such that the two problems have the same optimal objective values.

Given a linear program in which the objective is to maximize, we shall describe how to formulate a ***dual*** linear program in which the objective is to minimize and

32 String Matching

Text-editing programs frequently need to find all occurrences of a pattern in the text. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem—called "string matching"—can greatly aid the responsiveness of the text-editing program. Among their many other applications, string-matching algorithms search for particular patterns in DNA sequences. Internet search engines also use them to find Web pages relevant to queries.

We formalize the string-matching problem as follows. We assume that the text is an array $T[1\,..\,n]$ of length n and that the pattern is an array $P[1\,..\,m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{\texttt{0},\texttt{1}\}$ or $\Sigma = \{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$. The character arrays P and T are often called ***strings*** of characters.

Referring to Figure 32.1, we say that pattern P ***occurs with shift s*** in text T (or, equivalently, that pattern P ***occurs beginning at position s* + 1** in text T) if $0 \leq s \leq n - m$ and $T[s+1\,..\,s+m] = P[1\,..\,m]$ (that is, if $T[s+j] = P[j]$, for $1 \leq j \leq m$). If P occurs with shift s in T, then we call s a ***valid shift***; otherwise, we call s an ***invalid shift***. The ***string-matching problem*** is the problem of finding all valid shifts with which a given pattern P occurs in a given text T.

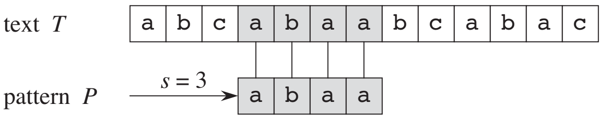


Figure 32.1 An example of the string-matching problem, where we want to find all occurrences of the pattern $P = \texttt{abaa}$ in the text $T = \texttt{abcabaabcabac}$. The pattern occurs only once in the text, at shift $s = 3$, which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

Algorithm	Preprocessing time	Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\,\lvert\Sigma\rvert)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$

Figure 32.2 The string-matching algorithms in this chapter and their preprocessing and matching times.

Except for the naive brute-force algorithm, which we review in Section 32.1, each string-matching algorithm in this chapter performs some preprocessing based on the pattern and then finds all valid shifts; we call this latter phase "matching." Figure 32.2 shows the preprocessing and matching times for each of the algorithms in this chapter. The total running time of each algorithm is the sum of the preprocessing and matching times. Section 32.2 presents an interesting string-matching algorithm, due to Rabin and Karp. Although the $\Theta((n-m+1)m)$ worst-case running time of this algorithm is no better than that of the naive method, it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems. Section 32.3 then describes a string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern P in a text. This algorithm takes $O(m\,\lvert\Sigma\rvert)$ preprocessing time, but only $\Theta(n)$ matching time. Section 32.4 presents the similar, but much cleverer, Knuth-Morris-Pratt (or KMP) algorithm; it has the same $\Theta(n)$ matching time, and it reduces the preprocessing time to only $\Theta(m)$.

Notation and terminology

We denote by Σ^* (read "sigma-star") the set of all finite-length strings formed using characters from the alphabet Σ. In this chapter, we consider only finite-length strings. The zero-length ***empty string***, denoted ε, also belongs to Σ^*. The length of a string x is denoted $|x|$. The ***concatenation*** of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.

We say that a string w is a ***prefix*** of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubset x$, then $|w| \le |x|$. Similarly, we say that a string w is a ***suffix*** of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. As with a prefix, $w \sqsupset x$ implies $|w| \le |x|$. For example, we have $\texttt{ab} \sqsubset \texttt{abcca}$ and $\texttt{cca} \sqsupset \texttt{abcca}$. The empty string ε is both a suffix and a prefix of every string. For any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$.

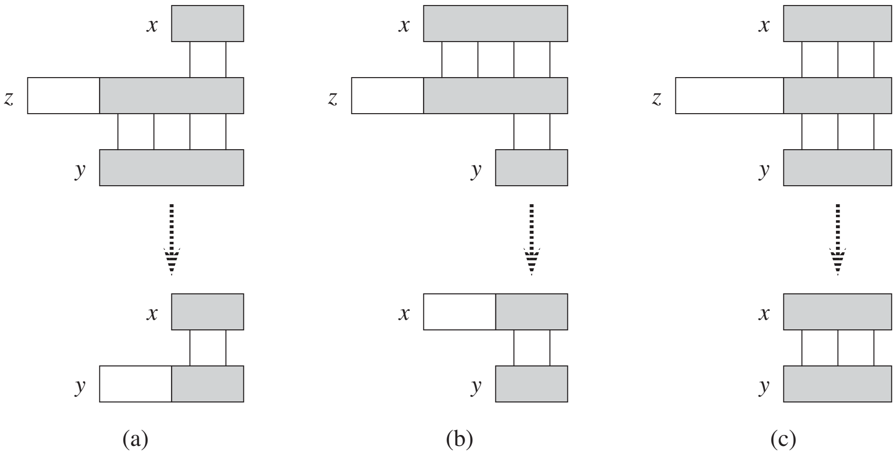


Figure 32.3 A graphical proof of Lemma 32.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. **(a)** If $|x| \le |y|$, then $x \sqsupset y$. **(b)** If $|x| \ge |y|$, then $y \sqsupset x$. **(c)** If $|x| = |y|$, then $x = y$.

Also note that $\sqsubset$ and $\sqsupset$ are transitive relations. The following lemma will be useful later.

Lemma 32.1 (Overlapping-suffix lemma)
Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof See Figure 32.3 for a graphical proof. ■

For brevity of notation, we denote the k-character prefix $P[1 \,..\, k]$ of the pattern $P[1 \,..\, m]$ by P_k. Thus, $P_0 = \varepsilon$ and $P_m = P = P[1 \,..\, m]$. Similarly, we denote the k-character prefix of the text T by T_k. Using this notation, we can state the string-matching problem as that of finding all shifts s in the range $0 \le s \le n - m$ such that $P \sqsupset T_{s+m}$.

In our pseudocode, we allow two equal-length strings to be compared for equality as a primitive operation. If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered. To be precise, the test "$x == y$" is assumed to take time $\Theta(t + 1)$, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$. (We write $\Theta(t + 1)$ rather than $\Theta(t)$ to handle the case in which $t = 0$; the first characters compared do not match, but it takes a positive amount of time to perform this comparison.)

32.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1\,..\,m] = T[s+1\,..\,s+m]$ for each of the $n-m+1$ possible values of s.

```
NAIVE-STRING-MATCHER(T, P)
1  $n = T.length$
2  $m = P.length$
3  for $s = 0$ to $n - m$
4      if $P[1\,..\,m] == T[s+1\,..\,s+m]$
5          print "Pattern occurs with shift" $s$
```

Figure 32.4 portrays the naive string-matching procedure as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text. The **for** loop of lines 3–5 considers each possible shift explicitly. The test in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

Procedure NAIVE-STRING-MATCHER takes time $O((n-m+1)m)$, and this bound is tight in the worst case. For example, consider the text string $\texttt{a}^n$ (a string of n a's) and the pattern $\texttt{a}^m$. For each of the $n-m+1$ possible values of the shift s, the implicit loop on line 4 to compare corresponding characters must execute m times to validate the shift. The worst-case running time is thus $\Theta((n-m+1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. Because it requires no preprocessing, NAIVE-STRING-MATCHER's running time equals its matching time.

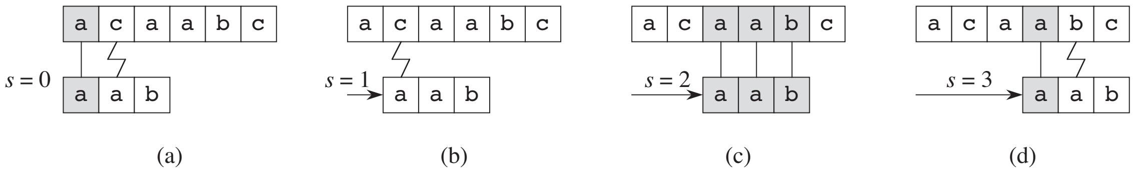


Figure 32.4 The operation of the naive string matcher for the pattern $P = \texttt{aab}$ and the text $T = \texttt{acaabc}$. We can imagine the pattern P as a template that we slide next to the text. **(a)–(d)** The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. The algorithm finds one occurrence of the pattern, at shift $s = 2$, shown in part (c).

As we shall see, NAIVE-STRING-MATCHER is not an optimal procedure for this problem. Indeed, in this chapter we shall see that the Knuth-Morris-Pratt algorithm is much better in the worst case. The naive string-matcher is inefficient because it entirely ignores information gained about the text for one value of s when it considers other values of s. Such information can be quite valuable, however. For example, if $P = \texttt{aaab}$ and we find that $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $T[4] = \texttt{b}$. In the following sections, we examine several ways to make effective use of this sort of information.

Exercises

32.1-1

Show the comparisons the naive string matcher makes for the pattern $P = \texttt{0001}$ in the text $T = \texttt{000010001010001}$.

32.1-2

Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n-character text T.

32.1-3

Suppose that pattern P and text T are *randomly* chosen strings of length m and n, respectively, from the d-ary alphabet $\Sigma_d = \{0, 1, \ldots, d-1\}$, where $d \geq 2$. Show that the *expected* number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n - m + 1)\frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$$

over all executions of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once it finds a mismatch or matches the entire pattern.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

32.1-4

Suppose we allow the pattern P to contain occurrences of a ***gap character*** $\diamond$ that can match an *arbitrary* string of characters (even one of zero length). For example, the pattern $\texttt{ab}\diamond\texttt{ba}\diamond\texttt{c}$ occurs in the text $\texttt{cabccbacbacab}$ as

$$\texttt{c}\;\underbrace{\texttt{ab}}_{\texttt{ab}}\;\underbrace{\texttt{cc}}_{\diamond}\;\underbrace{\texttt{ba}}_{\texttt{ba}}\;\underbrace{\texttt{cba}}_{\diamond}\;\underbrace{\texttt{c}}_{\texttt{c}}\;\texttt{ab}$$

and as

$$\texttt{c}\;\underbrace{\texttt{ab}}_{\texttt{ab}}\;\underbrace{\texttt{ccbac}}_{\diamond}\;\underbrace{\texttt{ba}}_{\texttt{ba}}\;\underbrace{\phantom{\texttt{ab}}}_{\diamond}\;\underbrace{\texttt{c}}_{\texttt{c}}\;\texttt{ab}\,.$$

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T, and analyze the running time of your algorithm.

32.2 The Rabin-Karp algorithm

Rabin and Karp proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$. Based on certain assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. You might want to refer to Section 31.1 for the relevant definitions.

For expository purposes, let us assume that $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}, \ldots, \texttt{9}\}$, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-d notation, where $d = |\Sigma|$.) We can then view a string of k consecutive characters as representing a length-k decimal number. The character string `31415` thus corresponds to the decimal number 31,415. Because we interpret the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

Given a pattern $P[1\,..\,m]$, let p denote its corresponding decimal value. In a similar manner, given a text $T[1\,..\,n]$, let t_s denote the decimal value of the length-m substring $T[s+1\,..\,s+m]$, for $s = 0, 1, \ldots, n-m$. Certainly, $t_s = p$ if and only if $T[s+1\,..\,s+m] = P[1\,..\,m]$; thus, s is a valid shift if and only if $t_s = p$. If we could compute p in time $\Theta(m)$ and all the t_s values in a total of $\Theta(n-m+1)$ time,[1] then we could determine all valid shifts s in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing p with each of the t_s values. (For the moment, let's not worry about the possibility that p and the t_s values might be very large numbers.)

We can compute p in time $\Theta(m)$ using Horner's rule (see Section 30.1):

$$p = P[m] + 10\,(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))\ .$$

Similarly, we can compute t_0 from $T[1\,..\,m]$ in time $\Theta(m)$.

[1] We write $\Theta(n-m+1)$ instead of $\Theta(n-m)$ because s takes on $n-m+1$ different values. The "+1" is significant in an asymptotic sense because when $m = n$, computing the lone t_s value takes $\Theta(1)$ time, not $\Theta(0)$ time.

To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time $\Theta(n-m)$, we observe that we can compute t_{s+1} from t_s in constant time, since

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1] \; . \qquad (32.1)$$

Subtracting $10^{m-1}T[s+1]$ removes the high-order digit from t_s, multiplying the result by 10 shifts the number left by one digit position, and adding $T[s+m+1]$ brings in the appropriate low-order digit. For example, if $m = 5$ and $t_s = 31415$, then we wish to remove the high-order digit $T[s+1] = 3$ and bring in the new low-order digit (suppose it is $T[s+5+1] = 2$) to obtain

$$\begin{aligned} t_{s+1} &= 10(31415 - 10000 \cdot 3) + 2 \\ &= 14152 \; . \end{aligned}$$

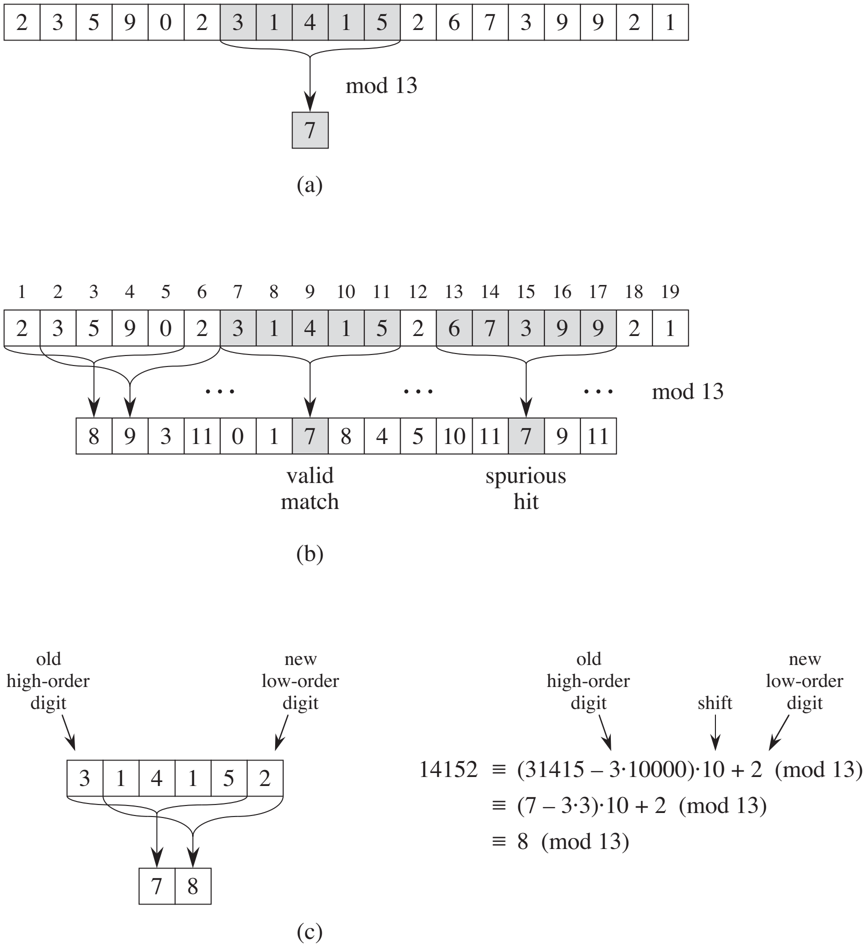
If we precompute the constant 10^{m-1} (which we can do in time $O(\lg m)$ using the techniques of Section 31.6, although for this application a straightforward $O(m)$-time method suffices), then each execution of equation (32.1) takes a constant number of arithmetic operations. Thus, we can compute p in time $\Theta(m)$, and we can compute all of $t_0, t_1, \ldots, t_{n-m}$ in time $\Theta(n-m+1)$. Therefore, we can find all occurrences of the pattern $P[1\,..\,m]$ in the text $T[1\,..\,n]$ with $\Theta(m)$ preprocessing time and $\Theta(n-m+1)$ matching time.

Until now, we have intentionally overlooked one problem: p and t_s may be too large to work with conveniently. If P contains m characters, then we cannot reasonably assume that each arithmetic operation on p (which is m digits long) takes "constant time." Fortunately, we can solve this problem easily, as Figure 32.5 shows: compute p and the t_s values modulo a suitable modulus q. We can compute p modulo q in $\Theta(m)$ time and all the t_s values modulo q in $\Theta(n-m+1)$ time. If we choose the modulus q as a prime such that $10q$ just fits within one computer word, then we can perform all the necessary computations with single-precision arithmetic. In general, with a d-ary alphabet $\{0, 1, \ldots, d-1\}$, we choose q so that dq fits within a computer word and adjust the recurrence equation (32.1) to work modulo q, so that it becomes

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q \; , \qquad (32.2)$$

where $h \equiv d^{m-1} \pmod{q}$ is the value of the digit "1" in the high-order position of an m-digit text window.

The solution of working modulo q is not perfect, however: $t_s \equiv p \pmod{q}$ does not imply that $t_s = p$. On the other hand, if $t_s \not\equiv p \pmod{q}$, then we definitely have that $t_s \neq p$, so that shift s is invalid. We can thus use the test $t_s \equiv p \pmod{q}$ as a fast heuristic test to rule out invalid shifts s. Any shift s for which $t_s \equiv p \pmod{q}$ must be tested further to see whether s is really valid or we just have a ***spurious hit***. This additional test explicitly checks the condition



$$\begin{aligned} 14152 &\equiv (31415 - 3\cdot 10000)\cdot 10 + 2 \pmod{13} \\ &\equiv (7 - 3\cdot 3)\cdot 10 + 2 \pmod{13} \\ &\equiv 8 \pmod{13} \end{aligned}$$

Figure 32.5 The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13. **(a)** A text string. A window of length 5 is shown shaded. The numerical value of the shaded number, computed modulo 13, yields the value 7. **(b)** The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern $P = 31415$, we look for windows whose value modulo 13 is 7, since $31415 \equiv 7 \pmod{13}$. The algorithm finds two such windows, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit. **(c)** How to compute the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. Because all computations are performed modulo 13, the value for the first window is 7, and the value for the new window is 8.

$P[1\,..\,m] = T[s+1\,..\,s+m]$. If q is large enough, then we hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure are the text T, the pattern P, the radix d to use (which is typically taken to be $|\Sigma|$), and the prime q to use.

```
RABIN-KARP-MATCHER($T, P, d, q$)
 1  $n = T.length$
 2  $m = P.length$
 3  $h = d^{m-1} \bmod q$
 4  $p = 0$
 5  $t_0 = 0$
 6  for $i = 1$ to $m$                    // preprocessing
 7      $p = (dp + P[i]) \bmod q$
 8      $t_0 = (dt_0 + T[i]) \bmod q$
 9  for $s = 0$ to $n - m$                // matching
10      if $p == t_s$
11          if $P[1\,..\,m] == T[s+1\,..\,s+m]$
12              print "Pattern occurs with shift" $s$
13      if $s < n - m$
14          $t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$
```

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix-d digits. The subscripts on t are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes h to the value of the high-order digit position of an m-digit window. Lines 4–8 compute p as the value of $P[1\,..\,m] \bmod q$ and t_0 as the value of $T[1\,..\,m] \bmod q$. The **for** loop of lines 9–14 iterates through all possible shifts s, maintaining the following invariant:

> Whenever line 10 is executed, $t_s = T[s+1\,..\,s+m] \bmod q$.

If $p = t_s$ in line 10 (a "hit"), then line 11 checks to see whether $P[1\,..\,m] = T[s+1\,..\,s+m]$ in order to rule out the possibility of a spurious hit. Line 12 prints out any valid shifts that are found. If $s < n - m$ (checked in line 13), then the **for** loop will execute at least one more time, and so line 14 first executes to ensure that the loop invariant holds when we get back to line 10. Line 14 computes the value of $t_{s+1} \bmod q$ from the value of $t_s \bmod q$ in constant time using equation (32.2) directly.

RABIN-KARP-MATCHER takes $\Theta(m)$ preprocessing time, and its matching time is $\Theta((n-m+1)m)$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = \mathtt{a}^m$

and $T = \mathtt{a}^n$, then verifying takes time $\Theta((n-m+1)m)$, since each of the $n-m+1$ possible shifts is valid.

In many applications, we expect few valid shifts—perhaps some constant c of them. In such applications, the expected matching time of the algorithm is only $O((n-m+1)+cm) = O(n+m)$, plus the time required to process spurious hits. We can base a heuristic analysis on the assumption that reducing values modulo q acts like a random mapping from Σ^* to $\mathbb{Z}_q$. (See the discussion on the use of division for hashing in Section 11.3.1. It is difficult to formalize and prove such an assumption, although one viable approach is to assume that q is chosen randomly from integers of the appropriate size. We shall not pursue this formalization here.) We can then expect that the number of spurious hits is $O(n/q)$, since we can estimate the chance that an arbitrary t_s will be equivalent to p, modulo q, as $1/q$. Since there are $O(n)$ positions at which the test of line 10 fails and we spend $O(m)$ time for each hit, the expected matching time taken by the Rabin-Karp algorithm is

$$O(n) + O(m(v + n/q)) ,$$

where v is the number of valid shifts. This running time is $O(n)$ if $v = O(1)$ and we choose $q \geq m$. That is, if the expected number of valid shifts is small ($O(1)$) and we choose the prime q to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to use only $O(n+m)$ matching time. Since $m \leq n$, this expected matching time is $O(n)$.

Exercises

32.2-1

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

32.2-2

How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

32.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

32.2-4

Alice has a copy of a long n-bit file $A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle$, and Bob similarly has an n-bit file $B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle$. Alice and Bob wish to know if their files are identical. To avoid transmitting all of A or B, they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer x from $\{0, 1, \ldots, q-1\}$. Then, Alice evaluates

$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q$$

and Bob similarly evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (*Hint:* See Exercise 31.4-4.)

32.3 String matching with finite automata

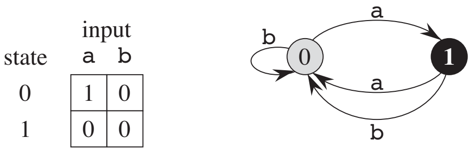
Many string-matching algorithms build a finite automaton—a simple machine for processing information—that scans the text string T for all occurrences of the pattern P. This section presents a method for building such an automaton. These string-matching automata are very efficient: they examine each text character *exactly once*, taking constant time per text character. The matching time used—after preprocessing the pattern to build the automaton—is therefore $\Theta(n)$. The time to build the automaton, however, can be large if Σ is large. Section 32.4 describes a clever way around this problem.

We begin this section with the definition of a finite automaton. We then examine a special string-matching automaton and show how to use it to find occurrences of a pattern in a text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

Finite automata

A ***finite automaton*** M, illustrated in Figure 32.6, is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.



	input	
state	a	b
0	1	0
1	0	0

(a) (b)

Figure 32.6 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, shown blackend, is the only accepting state. Directed edges represent transitions. For example, the edge from state 1 to state 0 labeled b indicates that $\delta(1, \texttt{b}) = 0$. 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 scanning the string w. Thus, M accepts a string w if and only if $\phi(w) \in A$. We define the function ϕ recursively, using the transition function:

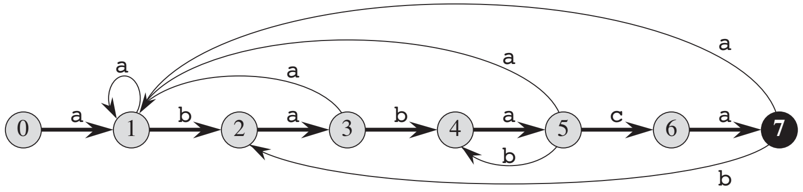
$$\begin{aligned} \phi(\varepsilon) &= q_0 , \\ \phi(wa) &= \delta(\phi(w), a) \quad \text{for } w \in \Sigma^*, a \in \Sigma . \end{aligned}$$

String-matching automata

For a given pattern P, we construct a string-matching automaton in a preprocessing step before using it to search the text string. Figure 32.7 illustrates how we construct the automaton for the pattern $P = \texttt{ababaca}$. From now on, we shall assume that P is a given fixed pattern string; for brevity, we shall not indicate the dependence upon P in our notation.

In order to specify the string-matching automaton corresponding to a given pattern $P[1\,..\,m]$, we first define an auxiliary function σ, called the ***suffix function*** corresponding to P. The function σ maps Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\} . \tag{32.3}$$



(a)

state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Figure 32.7 **(a)** A state-transition diagram for the string-matching automaton that accepts all strings ending in the string `ababaca`. State 0 is the start state, and state 7 (shown blackened) is the only accepting state. A directed edge from state i to state j labeled a represents $\delta(i,a) = j$. The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are omitted; by convention, if a state i has no outgoing edge labeled a for some $a \in \Sigma$, then $\delta(i,a) = 0$. **(b)** The corresponding transition function δ, and the pattern string $P =$ `ababaca`. The entries corresponding to successful matches between pattern and input characters are shown shaded. **(c)** The operation of the automaton on the text $T =$ `abababacaba`. Under each text character $T[i]$ appears the state $\phi(T_i)$ that the automaton is in after processing the prefix T_i. The automaton finds one occurrence of the pattern, ending in position 9.

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string. As examples, for the pattern $P =$ `ab`, we have $\sigma(\varepsilon) = 0$, $\sigma($`ccaca`$) = 1$, and $\sigma($`ccab`$) = 2$. For a pattern P of length m, we have $\sigma(x) = m$ if and only if $P \sqsupset x$. From the definition of the suffix function, $x \sqsupset y$ implies $\sigma(x) \le \sigma(y)$.

We define the string-matching automaton that corresponds to a given pattern $P[1\,..\,m]$ as follows:

- The state set Q is $\{0, 1, \ldots, m\}$. The start state q_0 is state 0, and state m is the only accepting state.
- The transition function δ is defined by the following equation, for any state q and character a:

$$\delta(q, a) = \sigma(P_q a) . \tag{32.4}$$

We define $\delta(q, a) = \sigma(P_q a)$ because we want to keep track of the longest prefix of the pattern P that has matched the text string T so far. We consider the most recently read characters of T. In order for a substring of T—let's say the substring ending at $T[i]$—to match some prefix P_j of P, this prefix P_j must be a suffix of T_i. Suppose that $q = \phi(T_i)$, so that after reading T_i, the automaton is in state q. We design the transition function δ so that this state number, q, tells us the length of the longest prefix of P that matches a suffix of T_i. That is, in state q, $P_q \sqsupset T_i$ and $q = \sigma(T_i)$. (Whenever $q = m$, all m characters of P match a suffix of T_i, and so we have found a match.) Thus, since $\phi(T_i)$ and $\sigma(T_i)$ both equal q, we shall see (in Theorem 32.4, below) that the automaton maintains the following invariant:

$$\phi(T_i) = \sigma(T_i) . \tag{32.5}$$

If the automaton is in state q and reads the next character $T[i + 1] = a$, then we want the transition to lead to the state corresponding to the longest prefix of P that is a suffix of $T_i a$, and that state is $\sigma(T_i a)$. Because P_q is the longest prefix of P that is a suffix of T_i, the longest prefix of P that is a suffix of $T_i a$ is not only $\sigma(T_i a)$, but also $\sigma(P_q a)$. (Lemma 32.3, on page 1000, proves that $\sigma(T_i a) = \sigma(P_q a)$.) Thus, when the automaton is in state q, we want the transition function on character a to take the automaton to state $\sigma(P_q a)$.

There are two cases to consider. In the first case, $a = P[q + 1]$, so that the character a continues to match the pattern; in this case, because $\delta(q, a) = q + 1$, the transition continues to go along the "spine" of the automaton (the heavy edges in Figure 32.7). In the second case, $a \neq P[q+1]$, so that a does not continue to match the pattern. Here, we must find a smaller prefix of P that is also a suffix of T_i. Because the preprocessing step matches the pattern against itself when creating the string-matching automaton, the transition function quickly identifies the longest such smaller prefix of P.

Let's look at an example. The string-matching automaton of Figure 32.7 has $\delta(5, \texttt{c}) = 6$, illustrating the first case, in which the match continues. To illustrate the second case, observe that the automaton of Figure 32.7 has $\delta(5, \texttt{b}) = 4$. We make this transition because if the automaton reads a b in state $q = 5$, then $P_q\texttt{b} = \texttt{ababab}$, and the longest prefix of P that is also a suffix of ababab is $P_4 = \texttt{abab}$.

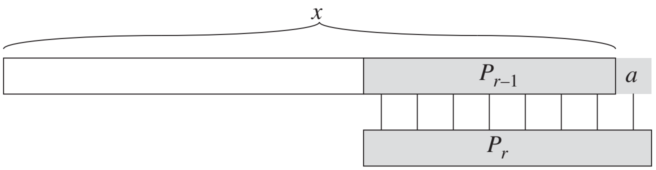


Figure 32.8 An illustration for the proof of Lemma 32.2. The figure shows that $r \le \sigma(x) + 1$, where $r = \sigma(xa)$.

To clarify the operation of a string-matching automaton, we now give a simple, efficient program for simulating the behavior of such an automaton (represented by its transition function δ) in finding occurrences of a pattern P of length m in an input text $T[1\,..\,n]$. As for any string-matching automaton for a pattern of length m, the state set Q is $\{0, 1, \ldots, m\}$, the start state is 0, and the only accepting state is state m.

```
FINITE-AUTOMATON-MATCHER$(T, \delta, m)$
$n = T.length$
$q = 0$
for $i = 1$ to $n$
    $q = \delta(q, T[i])$
    if $q == m$
        print "Pattern occurs with shift" $i - m$
```

From the simple loop structure of FINITE-AUTOMATON-MATCHER, we can easily see that its matching time on a text string of length n is $\Theta(n)$. This matching time, however, does not include the preprocessing time required to compute the transition function δ. We address this problem later, after first proving that the procedure FINITE-AUTOMATON-MATCHER operates correctly.

Consider how the automaton operates on an input text $T[1\,..\,n]$. We shall prove that the automaton is in state $\sigma(T_i)$ after scanning character $T[i]$. Since $\sigma(T_i) = m$ if and only if $P \sqsupset T_i$, the machine is in the accepting state m if and only if it has just scanned the pattern P. To prove this result, we make use of the following two lemmas about the suffix function σ.

Lemma 32.2 (Suffix-function inequality)
For any string x and character a, we have $\sigma(xa) \le \sigma(x) + 1$.

Proof Referring to Figure 32.8, let $r = \sigma(xa)$. If $r = 0$, then the conclusion $\sigma(xa) = r \le \sigma(x) + 1$ is trivially satisfied, by the nonnegativity of $\sigma(x)$. Now assume that $r > 0$. Then, $P_r \sqsupset xa$, by the definition of σ. Thus, $P_{r-1} \sqsupset x$, by

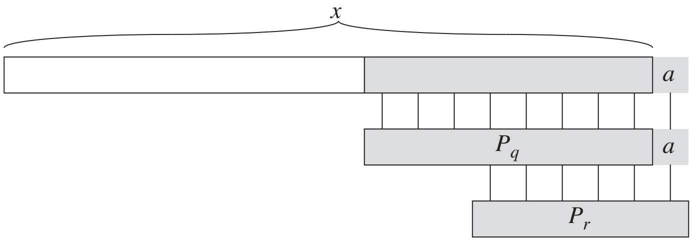


Figure 32.9 An illustration for the proof of Lemma 32.3. The figure shows that $r = \sigma(P_q a)$, where $q = \sigma(x)$ and $r = \sigma(xa)$.

dropping the a from the end of P_r and from the end of xa. Therefore, $r-1 \le \sigma(x)$, since $\sigma(x)$ is the largest k such that $P_k \sqsupset x$, and thus $\sigma(xa) = r \le \sigma(x) + 1$. ■

Lemma 32.3 (Suffix-function recursion lemma)
For any string x and character a, if $q = \sigma(x)$, then $\sigma(xa) = \sigma(P_q a)$.

Proof From the definition of σ, we have $P_q \sqsupset x$. As Figure 32.9 shows, we also have $P_q a \sqsupset xa$. If we let $r = \sigma(xa)$, then $P_r \sqsupset xa$ and, by Lemma 32.2, $r \le q + 1$. Thus, we have $|P_r| = r \le q + 1 = |P_q a|$. Since $P_q a \sqsupset xa$, $P_r \sqsupset xa$, and $|P_r| \le |P_q a|$, Lemma 32.1 implies that $P_r \sqsupset P_q a$. Therefore, $r \le \sigma(P_q a)$, that is, $\sigma(xa) \le \sigma(P_q a)$. But we also have $\sigma(P_q a) \le \sigma(xa)$, since $P_q a \sqsupset xa$. Thus, $\sigma(xa) = \sigma(P_q a)$. ■

We are now ready to prove our main theorem characterizing the behavior of a string-matching automaton on a given input text. As noted above, this theorem shows that the automaton is merely keeping track, at each step, of the longest prefix of the pattern that is a suffix of what has been read so far. In other words, the automaton maintains the invariant (32.5).

Theorem 32.4
If ϕ is the final-state function of a string-matching automaton for a given pattern P and $T[1..n]$ is an input text for the automaton, then

$$\phi(T_i) = \sigma(T_i)$$

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

Proof The proof is by induction on i. For $i = 0$, the theorem is trivially true, since $T_0 = \varepsilon$. Thus, $\phi(T_0) = 0 = \sigma(T_0)$.

Now, we assume that $\phi(T_i) = \sigma(T_i)$ and prove that $\phi(T_{i+1}) = \sigma(T_{i+1})$. Let q denote $\phi(T_i)$, and let a denote $T[i+1]$. Then,

$$\begin{aligned}
\phi(T_{i+1}) &= \phi(T_i a) && \text{(by the definitions of } T_{i+1} \text{ and } a\text{)} \\
&= \delta(\phi(T_i), a) && \text{(by the definition of } \phi\text{)} \\
&= \delta(q, a) && \text{(by the definition of } q\text{)} \\
&= \sigma(P_q a) && \text{(by the definition (32.4) of } \delta\text{)} \\
&= \sigma(T_i a) && \text{(by Lemma 32.3 and induction)} \\
&= \sigma(T_{i+1}) && \text{(by the definition of } T_{i+1}\text{)} \ . \qquad \blacksquare
\end{aligned}$$

By Theorem 32.4, if the machine enters state q on line 4, then q is the largest value such that $P_q \sqsupset T_i$. Thus, we have $q = m$ on line 5 if and only if the machine has just scanned an occurrence of the pattern P. We conclude that FINITE-AUTOMATON-MATCHER operates correctly.

Computing the transition function

The following procedure computes the transition function δ from a given pattern $P[1\,..\,m]$.

```
COMPUTE-TRANSITION-FUNCTION($P, \Sigma$)
1  $m = P.length$
2  for $q = 0$ to $m$
3      for each character $a \in \Sigma$
4          $k = \min(m+1, q+2)$
5          repeat
6              $k = k - 1$
7          until $P_k \sqsupset P_q a$
8          $\delta(q, a) = k$
9  return $\delta$
```

This procedure computes $\delta(q, a)$ in a straightforward manner according to its definition in equation (32.4). The nested loops beginning on lines 2 and 3 consider all states q and all characters a, and lines 4–8 set $\delta(q, a)$ to be the largest k such that $P_k \sqsupset P_q a$. The code starts with the largest conceivable value of k, which is $\min(m, q+1)$. It then decreases k until $P_k \sqsupset P_q a$, which must eventually occur, since $P_0 = \varepsilon$ is a suffix of every string.

The running time of COMPUTE-TRANSITION-FUNCTION is $O(m^3 |\Sigma|)$, because the outer loops contribute a factor of $m |\Sigma|$, the inner **repeat** loop can run at most $m+1$ times, and the test $P_k \sqsupset P_q a$ on line 7 can require comparing up to m characters. Much faster procedures exist; by utilizing some cleverly computed information about the pattern P (see Exercise 32.4-8), we can improve the time required to compute δ from P to $O(m\,|\Sigma|)$. With this improved procedure for computing δ, we can find all occurrences of a length-m pattern in a length-n text over an alphabet Σ with $O(m\,|\Sigma|)$ preprocessing time and $\Theta(n)$ matching time.

Exercises

32.3-1
Construct the string-matching automaton for the pattern $P = \texttt{aabab}$ and illustrate its operation on the text string $T = \texttt{aaababaabaababaab}$.

32.3-2
Draw a state-transition diagram for a string-matching automaton for the pattern $\texttt{ababbabbababbababbabb}$ over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$.

32.3-3
We call a pattern P ***nonoverlappable*** if $P_k \sqsupset P_q$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

32.3-4 ★
Given two patterns P and P', describe how to construct a finite automaton that determines all occurrences of *either* pattern. Try to minimize the number of states in your automaton.

32.3-5
Given a pattern P containing gap characters (see Exercise 32.1-4), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ matching time, where $n = |T|$.

★ 32.4 The Knuth-Morris-Pratt algorithm

We now present a linear-time string-matching algorithm due to Knuth, Morris, and Pratt. This algorithm avoids computing the transition function δ altogether, and its matching time is $\Theta(n)$ using just an auxiliary function π, which we precompute from the pattern in time $\Theta(m)$ and store in an array $\pi[1\,..\,m]$. The array π allows us to compute the transition function δ efficiently (in an amortized sense) "on the fly" as needed. Loosely speaking, for any state $q = 0, 1, \ldots, m$ and any character

33 Computational Geometry

Computational geometry is the branch of computer science that studies algorithms for solving geometric problems. In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VLSI design, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics. The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order. The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

In this chapter, we look at a few computational-geometry algorithms in two dimensions, that is, in the plane. We represent each input object by a set of points $\{p_1, p_2, p_3, \ldots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$. For example, we represent an n-vertex polygon P by a sequence $\langle p_0, p_1, p_2, \ldots, p_{n-1} \rangle$ of its vertices in order of their appearance on the boundary of P. Computational geometry can also apply to three dimensions, and even higher-dimensional spaces, but such problems and their solutions can be very difficult to visualize. Even in two dimensions, however, we can see a good sample of computational-geometry techniques.

Section 33.1 shows how to answer basic questions about line segments efficiently and accurately: whether one segment is clockwise or counterclockwise from another that shares an endpoint, which way we turn when traversing two adjoining line segments, and whether two line segments intersect. Section 33.2 presents a technique called "sweeping" that we use to develop an $O(n \lg n)$-time algorithm for determining whether a set of n line segments contains any intersections. Section 33.3 gives two "rotational-sweep" algorithms that compute the convex hull (smallest enclosing convex polygon) of a set of n points: Graham's scan, which runs in time $O(n \lg n)$, and Jarvis's march, which takes $O(nh)$ time, where h is the number of vertices of the convex hull. Finally, Section 33.4 gives an $O(n \lg n)$-time divide-and-conquer algorithm for finding the closest pair of points in a set of n points in the plane.

33.1 Line-segment properties

Several of the computational-geometry algorithms in this chapter require answers to questions about the properties of line segments. A ***convex combination*** of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \le \alpha \le 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$. We also write that $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitively, p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line. Given two distinct points p_1 and p_2, the ***line segment*** $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2. We call p_1 and p_2 the ***endpoints*** of segment $\overline{p_1 p_2}$. Sometimes the ordering of p_1 and p_2 matters, and we speak of the ***directed segment*** $\overrightarrow{p_1 p_2}$. If p_1 is the ***origin*** $(0, 0)$, then we can treat the directed segment $\overrightarrow{p_1 p_2}$ as the ***vector*** p_2.

In this section, we shall explore the following questions:

1. Given two directed segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_0 p_2}$, is $\overrightarrow{p_0 p_1}$ clockwise from $\overrightarrow{p_0 p_2}$ with respect to their common endpoint p_0?
2. Given two line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$, if we traverse $\overline{p_0 p_1}$ and then $\overline{p_1 p_2}$, do we make a left turn at point p_1?
3. Do line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect?

There are no restrictions on the given points.

We can answer each question in $O(1)$ time, which should come as no surprise since the input size of each question is $O(1)$. Moreover, our methods use only additions, subtractions, multiplications, and comparisons. We need neither division nor trigonometric functions, both of which can be computationally expensive and prone to problems with round-off error. For example, the "straightforward" method of determining whether two segments intersect—compute the line equation of the form $y = mx + b$ for each segment (m is the slope and b is the y-intercept), find the point of intersection of the lines, and check whether this point is on both segments—uses division to find the point of intersection. When the segments are nearly parallel, this method is very sensitive to the precision of the division operation on real computers. The method in this section, which avoids division, is much more accurate.

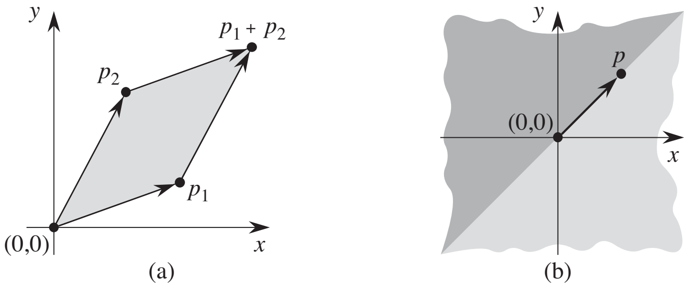


Figure 33.1 **(a)** The cross product of vectors p_1 and p_2 is the signed area of the parallelogram. **(b)** The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

Cross products

Computing cross products lies at the heart of our line-segment methods. Consider vectors p_1 and p_2, shown in Figure 33.1(a). We can interpret the ***cross product*** $p_1 \times p_2$ as the signed area of the parallelogram formed by the points $(0,0)$, p_1, p_2, and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. An equivalent, but more useful, definition gives the cross product as the determinant of a matrix:[1]

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \; . \end{aligned}$$

If $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2 with respect to the origin $(0,0)$; if this cross product is negative, then p_1 is counterclockwise from p_2. (See Exercise 33.1-1.) Figure 33.1(b) shows the clockwise and counterclockwise regions relative to a vector p. A boundary condition arises if the cross product is 0; in this case, the vectors are ***colinear***, pointing in either the same or opposite directions.

To determine whether a directed segment $\overrightarrow{p_0 p_1}$ is closer to a directed segment $\overrightarrow{p_0 p_2}$ in a clockwise direction or in a counterclockwise direction with respect to their common endpoint p_0, we simply translate to use p_0 as the origin. That is, we let $p_1 - p_0$ denote the vector $p_1' = (x_1', y_1')$, where $x_1' = x_1 - x_0$ and $y_1' = y_1 - y_0$, and we define $p_2 - p_0$ similarly. We then compute the cross product

[1] Actually, the cross product is a three-dimensional concept. It is a vector that is perpendicular to both p_1 and p_2 according to the "right-hand rule" and whose magnitude is $|x_1 y_2 - x_2 y_1|$. In this chapter, however, we find it convenient to treat the cross product simply as the value $x_1 y_2 - x_2 y_1$.

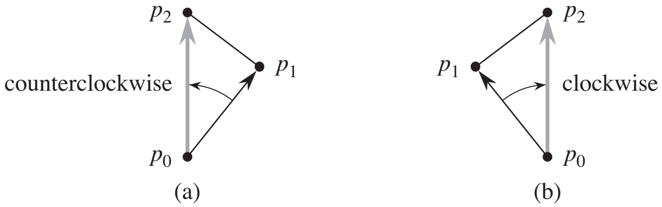


Figure 33.2 Using the cross product to determine how consecutive line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ turn at point p_1. We check whether the directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to the directed segment $\overrightarrow{p_0 p_1}$. **(a)** If counterclockwise, the points make a left turn. **(b)** If clockwise, they make a right turn.

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) .$$

If this cross product is positive, then $\overrightarrow{p_0 p_1}$ is clockwise from $\overrightarrow{p_0 p_2}$; if negative, it is counterclockwise.

Determining whether consecutive segments turn left or right

Our next question is whether two consecutive line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$ turn left or right at point p_1. Equivalently, we want a method to determine which way a given angle $\angle p_0 p_1 p_2$ turns. Cross products allow us to answer this question without computing the angle. As Figure 33.2 shows, we simply check whether directed segment $\overrightarrow{p_0 p_2}$ is clockwise or counterclockwise relative to directed segment $\overrightarrow{p_0 p_1}$. To do so, we compute the cross product $(p_2 - p_0) \times (p_1 - p_0)$. If the sign of this cross product is negative, then $\overrightarrow{p_0 p_2}$ is counterclockwise with respect to $\overrightarrow{p_0 p_1}$, and thus we make a left turn at p_1. A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 means that points p_0, p_1, and p_2 are colinear.

Determining whether two line segments intersect

To determine whether two line segments intersect, we check whether each segment straddles the line containing the other. A segment $\overline{p_1 p_2}$ ***straddles*** a line if point p_1 lies on one side of the line and point p_2 lies on the other side. A boundary case arises if p_1 or p_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions holds:

1. Each segment straddles the line containing the other.
2. An endpoint of one segment lies on the other segment. (This condition comes from the boundary case.)

The following procedures implement this idea. SEGMENTS-INTERSECT returns TRUE if segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect and FALSE if they do not. It calls the subroutines DIRECTION, which computes relative orientations using the cross-product method above, and ON-SEGMENT, which determines whether a point known to be colinear with a segment lies on that segment.

```
SEGMENTS-INTERSECT($p_1, p_2, p_3, p_4$)
 1  $d_1$ = DIRECTION($p_3, p_4, p_1$)
 2  $d_2$ = DIRECTION($p_3, p_4, p_2$)
 3  $d_3$ = DIRECTION($p_1, p_2, p_3$)
 4  $d_4$ = DIRECTION($p_1, p_2, p_4$)
 5  if (($d_1 > 0$ and $d_2 < 0$) or ($d_1 < 0$ and $d_2 > 0$)) and
        (($d_3 > 0$ and $d_4 < 0$) or ($d_3 < 0$ and $d_4 > 0$))
 6      return TRUE
 7  elseif $d_1$ == 0 and ON-SEGMENT($p_3, p_4, p_1$)
 8      return TRUE
 9  elseif $d_2$ == 0 and ON-SEGMENT($p_3, p_4, p_2$)
10      return TRUE
11  elseif $d_3$ == 0 and ON-SEGMENT($p_1, p_2, p_3$)
12      return TRUE
13  elseif $d_4$ == 0 and ON-SEGMENT($p_1, p_2, p_4$)
14      return TRUE
15  else return FALSE
```

```
DIRECTION($p_i, p_j, p_k$)
1  return $(p_k - p_i) \times (p_j - p_i)$
```

```
ON-SEGMENT($p_i, p_j, p_k$)
1  if $\min(x_i, x_j) \leq x_k \leq \max(x_i, x_j)$ and $\min(y_i, y_j) \leq y_k \leq \max(y_i, y_j)$
2      return TRUE
3  else return FALSE
```

SEGMENTS-INTERSECT works as follows. Lines 1–4 compute the relative orientation d_i of each endpoint p_i with respect to the other segment. If all the relative orientations are nonzero, then we can easily determine whether segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect, as follows. Segment $\overline{p_1 p_2}$ straddles the line containing segment $\overline{p_3 p_4}$ if directed segments $\overrightarrow{p_3 p_1}$ and $\overrightarrow{p_3 p_2}$ have opposite orientations relative to $\overrightarrow{p_3 p_4}$. In this case, the signs of d_1 and d_2 differ. Similarly, $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$ if the signs of d_3 and d_4 differ. If the test of line 5 is true, then the segments straddle each other, and SEGMENTS-INTERSECT returns TRUE. Figure 33.3(a) shows this case. Otherwise, the segments do not straddle

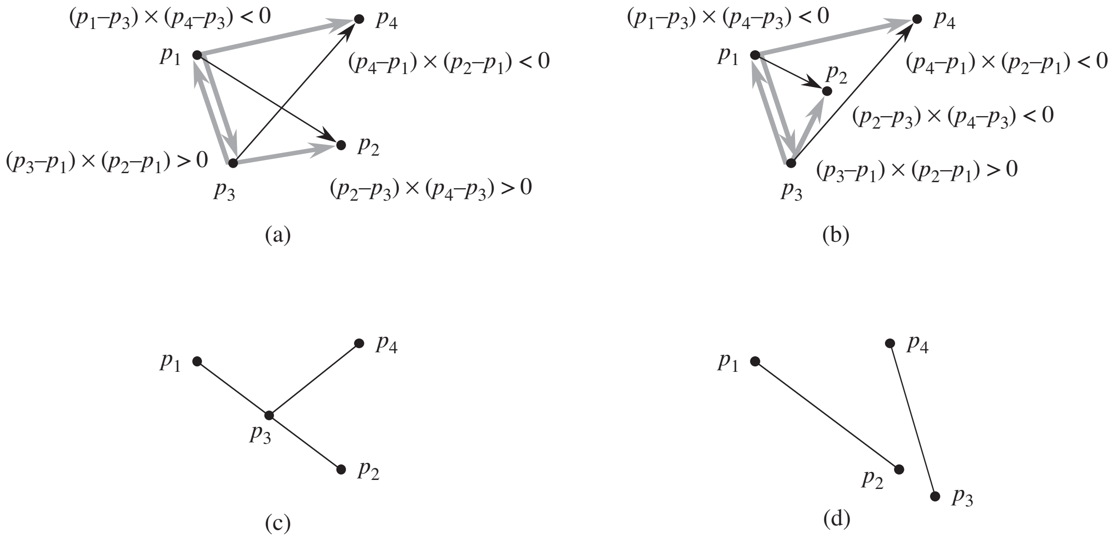

$(p_1 – p_3) \times (p_4 – p_3) < 0$
p_4
p_1
$(p_4 – p_1) \times (p_2 – p_1) < 0$
$(p_3 – p_1) \times (p_2 – p_1) > 0$
p_2
p_3
$(p_2 – p_3) \times (p_4 – p_3) > 0$
(a)
$(p_1 – p_3) \times (p_4 – p_3) < 0$
p_4
p_1
p_2
$(p_4 – p_1) \times (p_2 – p_1) < 0$
$(p_2 – p_3) \times (p_4 – p_3) < 0$
$(p_3 – p_1) \times (p_2 – p_1) > 0$
p_3
(b)
p_1
p_4
p_3
p_2
(c)
p_4
p_1
p_2
p_3
(d)

Figure 33.3 Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point p_3 is colinear with $\overline{p_1 p_2}$ and is between p_1 and p_2. **(d)** Point p_3 is colinear with $\overline{p_1 p_2}$, but it is not between p_1 and p_2. The segments do not intersect.

each other's lines, although a boundary case may apply. If all the relative orientations are nonzero, no boundary case applies. All the tests against 0 in lines 7–13 then fail, and SEGMENTS-INTERSECT returns FALSE in line 15. Figure 33.3(b) shows this case.

A boundary case occurs if any relative orientation d_k is 0. Here, we know that p_k is colinear with the other segment. It is directly on the other segment if and only if it is between the endpoints of the other segment. The procedure ON-SEGMENT returns whether p_k is between the endpoints of segment $\overline{p_i p_j}$, which will be the other segment when called in lines 7–13; the procedure assumes that p_k is colinear with segment $\overline{p_i p_j}$. Figures 33.3(c) and (d) show cases with colinear points. In Figure 33.3(c), p_3 is on $\overline{p_1 p_2}$, and so SEGMENTS-INTERSECT returns TRUE in line 12. No endpoints are on other segments in Figure 33.3(d), and so SEGMENTS-INTERSECT returns FALSE in line 15.

Other applications of cross products

Later sections of this chapter introduce additional uses for cross products. In Section 33.3, we shall need to sort a set of points according to their polar angles with respect to a given origin. As Exercise 33.1-3 asks you to show, we can use cross products to perform the comparisons in the sorting procedure. In Section 33.2, we shall use red-black trees to maintain the vertical ordering of a set of line segments. Rather than keeping explicit key values which we compare to each other in the red-black tree code, we shall compute a cross-product to determine which of two segments that intersect a given vertical line is above the other.

Exercises

33.1-1
Prove that if $p_1 \times p_2$ is positive, then vector p_1 is clockwise from vector p_2 with respect to the origin $(0, 0)$ and that if this cross product is negative, then p_1 is counterclockwise from p_2.

33.1-2
Professor van Pelt proposes that only the x-dimension needs to be tested in line 1 of ON-SEGMENT. Show why the professor is wrong.

33.1-3
The ***polar angle*** of a point p_1 with respect to an origin point p_0 is the angle of the vector $p_1 - p_0$ in the usual polar coordinate system. For example, the polar angle of $(3, 5)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 45 degrees or $\pi/4$ radians. The polar angle of $(3, 3)$ with respect to $(2, 4)$ is the angle of the vector $(1, -1)$, which is 315 degrees or $7\pi/4$ radians. Write pseudocode to sort a sequence $\langle p_1, p_2, \ldots, p_n \rangle$ of n points according to their polar angles with respect to a given origin point p_0. Your procedure should take $O(n \lg n)$ time and use cross products to compare angles.

33.1-4
Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are colinear.

33.1-5
A ***polygon*** is a piecewise-linear, closed curve in the plane. That is, it is a curve ending on itself that is formed by a sequence of straight-line segments, called the ***sides*** of the polygon. A point joining two consecutive sides is a ***vertex*** of the polygon. If the polygon is ***simple***, as we shall generally assume, it does not cross itself. The set of points in the plane enclosed by a simple polygon forms the ***interior*** of

the polygon, the set of points on the polygon itself forms its ***boundary***, and the set of points surrounding the polygon forms its ***exterior***. A simple polygon is ***convex*** if, given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior. A vertex of a convex polygon cannot be expressed as a convex combination of any two distinct points on the boundary or in the interior of the polygon.

Professor Amundsen proposes the following method to determine whether a sequence $\langle p_0, p_1, \ldots, p_{n-1} \rangle$ of n points forms the consecutive vertices of a convex polygon. Output "yes" if the set $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \ldots, n-1\}$, where subscript addition is performed modulo n, does not contain both left turns and right turns; otherwise, output "no." Show that although this method runs in linear time, it does not always produce the correct answer. Modify the professor's method so that it always produces the correct answer in linear time.

33.1-6

Given a point $p_0 = (x_0, y_0)$, the ***right horizontal ray*** from p_0 is the set of points $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ and } y_i = y_0\}$, that is, it is the set of points due right of p_0 along with p_0 itself. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{p_1 p_2}$ in $O(1)$ time by reducing the problem to that of determining whether two line segments intersect.

33.1-7

One way to determine whether a point p_0 is in the interior of a simple, but not necessarily convex, polygon P is to look at any ray from p_0 and check that the ray intersects the boundary of P an odd number of times but that p_0 itself is not on the boundary of P. Show how to compute in $\Theta(n)$ time whether a point p_0 is in the interior of an n-vertex polygon P. (*Hint:* Use Exercise 33.1-6. Make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.)

33.1-8

Show how to compute the area of an n-vertex simple, but not necessarily convex, polygon in $\Theta(n)$ time. (See Exercise 33.1-5 for definitions pertaining to polygons.)

33.2 Determining whether any pair of segments intersects

This section presents an algorithm for determining whether any two line segments in a set of segments intersect. The algorithm uses a technique known as "sweeping," which is common to many computational-geometry algorithms. Moreover, as the exercises at the end of this section show, this algorithm, or simple variations of it, can help solve other computational-geometry problems.

The algorithm runs in $O(n \lg n)$ time, where n is the number of segments we are given. It determines only whether or not any intersection exists; it does not print all the intersections. (By Exercise 33.2-1, it takes $\Omega(n^2)$ time in the worst case to find *all* the intersections in a set of n line segments.)

In ***sweeping***, an imaginary vertical ***sweep line*** passes through the given set of geometric objects, usually from left to right. We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time. Sweeping provides a method for ordering geometric objects, usually by placing them into a dynamic data structure, and for taking advantage of relationships among them. The line-segment-intersection algorithm in this section considers all the line-segment endpoints in left-to-right order and checks for an intersection each time it encounters an endpoint.

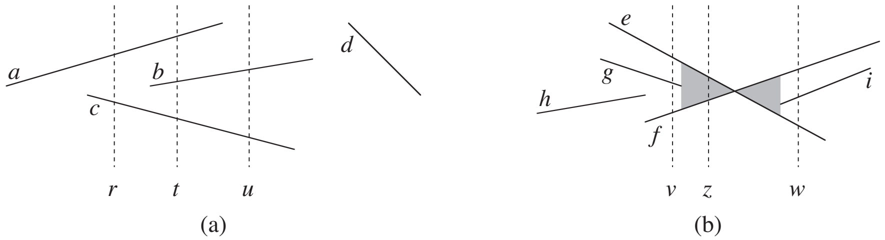
To describe and prove correct our algorithm for determining whether any two of n line segments intersect, we shall make two simplifying assumptions. First, we assume that no input segment is vertical. Second, we assume that no three input segments intersect at a single point. Exercises 33.2-8 and 33.2-9 ask you to show that the algorithm is robust enough that it needs only a slight modification to work even when these assumptions do not hold. Indeed, removing such simplifying assumptions and dealing with boundary conditions often present the most difficult challenges when programming computational-geometry algorithms and proving their correctness.

Ordering segments

Because we assume that there are no vertical segments, we know that any input segment intersecting a given vertical sweep line intersects it at a single point. Thus, we can order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection.

To be more precise, consider two segments s_1 and s_2. We say that these segments are ***comparable*** at x if the vertical sweep line with x-coordinate x intersects both of them. We say that s_1 is ***above*** s_2 at x, written $s_1 \succeq_x s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line, or if s_1 and s_2 intersect at the sweep line. In Figure 33.4(a), for example, we have the relationships $a \succeq_r c$, $a \succeq_t b$, $b \succeq_t c$, $a \succeq_t c$, and $b \succeq_u c$. Segment d is not comparable with any other segment.

For any given x, the relation "$\succeq_x$" is a total preorder (see Section B.2) for all segments that intersect the sweep line at x. That is, the relation is transitive, and if segments s_1 and s_2 each intersect the sweep line at x, then either $s_1 \succeq_x s_2$ or $s_2 \succeq_x s_1$, or both (if s_1 and s_2 intersect at the sweep line). (The relation $\succeq_x$ is



(a) (b)

Figure 33.4 The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$, and $b \succcurlyeq_u c$. Segment d is comparable with no other segment shown. **(b)** When segments e and f intersect, they reverse their orders: we have $e \succcurlyeq_v f$ but $f \succcurlyeq_w e$. Any sweep line (such as z) that passes through the shaded region has e and f consecutive in the ordering given by the relation $\succcurlyeq_z$.

also reflexive, but neither symmetric nor antisymmetric.) The total preorder may differ for differing values of x, however, as segments enter and leave the ordering. A segment enters the ordering when its left endpoint is encountered by the sweep, and it leaves the ordering when its right endpoint is encountered.

What happens when the sweep line passes through the intersection of two segments? As Figure 33.4(b) shows, the segments reverse their positions in the total preorder. Sweep lines v and w are to the left and right, respectively, of the point of intersection of segments e and f, and we have $e \succcurlyeq_v f$ and $f \succcurlyeq_w e$. Note that because we assume that no three segments intersect at the same point, there must be some vertical sweep line x for which intersecting segments e and f are *consecutive* in the total preorder $\succcurlyeq_x$. Any sweep line that passes through the shaded region of Figure 33.4(b), such as z, has e and f consecutive in its total preorder.

Moving the sweep line

Sweeping algorithms typically manage two sets of data:

1. The ***sweep-line status*** gives the relationships among the objects that the sweep line intersects.
2. The ***event-point schedule*** is a sequence of points, called ***event points***, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point, and then resumes. Changes to the sweep-line status occur only at event points.

For some algorithms (the algorithm asked for in Exercise 33.2-7, for example), the event-point schedule develops dynamically as the algorithm progresses. The algorithm at hand, however, determines all the event points before the sweep, based solely on simple properties of the input data. In particular, each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. (If two or more endpoints are ***covertical***, i.e., they have the same x-coordinate, we break the tie by putting all the covertical left endpoints before the covertical right endpoints. Within a set of covertical left endpoints, we put those with lower y-coordinates first, and we do the same within a set of covertical right endpoints.) When we encounter a segment's left endpoint, we insert the segment into the sweep-line status, and we delete the segment from the sweep-line status upon encountering its right endpoint. Whenever two segments first become consecutive in the total preorder, we check whether they intersect.

The sweep-line status is a total preorder T, for which we require the following operations:

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

It is possible for segments s_1 and s_2 to be mutually above each other in the total preorder T; this situation can occur if s_1 and s_2 intersect at the sweep line whose total preorder is given by T. In this case, the two segments may appear in either order in T.

If the input contains n segments, we can perform each of the operations INSERT, DELETE, ABOVE, and BELOW in $O(\lg n)$ time using red-black trees. Recall that the red-black-tree operations in Chapter 13 involve comparing keys. We can replace the key comparisons by comparisons that use cross products to determine the relative ordering of two segments (see Exercise 33.2-2).

Segment-intersection pseudocode

The following algorithm takes as input a set S of n line segments, returning the boolean value TRUE if any pair of segments in S intersects, and FALSE otherwise. A red-black tree maintains the total preorder T.

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  sort the endpoints of the segments in S from left to right,
          breaking ties by putting left endpoints before right endpoints
          and breaking further ties by putting points with lower
          y-coordinates first
 3  for each point p in the sorted list of endpoints
 4      if p is the left endpoint of a segment s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
 7              return TRUE
 8      if p is the right endpoint of a segment s
 9          if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

Figure 33.5 illustrates how the algorithm works. Line 1 initializes the total preorder to be empty. Line 2 determines the event-point schedule by sorting the $2n$ segment endpoints from left to right, breaking ties as described above. One way to perform line 2 is by lexicographically sorting the endpoints on (x, e, y), where x and y are the usual coordinates, $e = 0$ for a left endpoint, and $e = 1$ for a right endpoint.

Each iteration of the **for** loop of lines 3–11 processes one event point p. If p is the left endpoint of a segment s, line 5 adds s to the total preorder, and lines 6–7 return TRUE if s intersects either of the segments it is consecutive with in the total preorder defined by the sweep line passing through p. (A boundary condition occurs if p lies on another segment s'. In this case, we require only that s and s' be placed consecutively into T.) If p is the right endpoint of a segment s, then we need to delete s from the total preorder. But first, lines 9–10 return TRUE if there is an intersection between the segments surrounding s in the total preorder defined by the sweep line passing through p. If these segments do not intersect, line 11 deletes segment s from the total preorder. If the segments surrounding segment s intersect, they would have become consecutive after deleting s had the **return** statement in line 10 not prevented line 11 from executing. The correctness argument, which follows, will make it clear why it suffices to check the segments surrounding s. Finally, if we never find any intersections after having processed all $2n$ event points, line 12 returns FALSE.

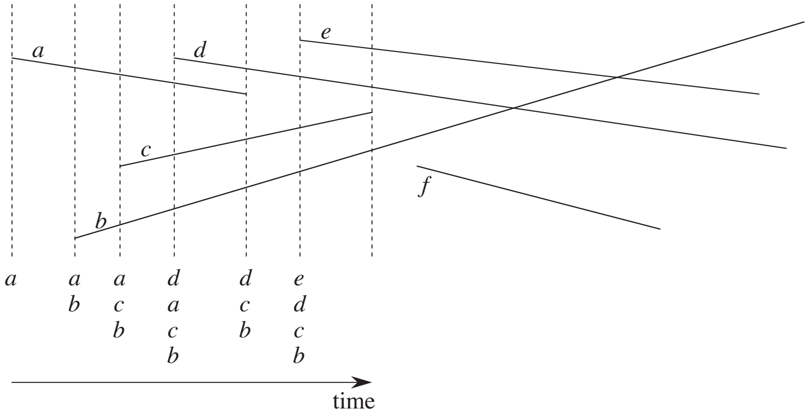


Figure 33.5 The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep line corresponds to the total preorder T at the end of the **for** loop processing the corresponding event point. The rightmost sweep line occurs when processing the right endpoint of segment c; because segments d and b surround c and intersect each other, the procedure returns TRUE.

Correctness

To show that ANY-SEGMENTS-INTERSECT is correct, we will prove that the call ANY-SEGMENTS-INTERSECT(S) returns TRUE if and only if there is an intersection among the segments in S.

It is easy to see that ANY-SEGMENTS-INTERSECT returns TRUE (on lines 7 and 10) only if it finds an intersection between two of the input segments. Hence, if it returns TRUE, there is an intersection.

We also need to show the converse: that if there is an intersection, then ANY-SEGMENTS-INTERSECT returns TRUE. Let us suppose that there is at least one intersection. Let p be the leftmost intersection point, breaking ties by choosing the point with the lowest y-coordinate, and let a and b be the segments that intersect at p. Since no intersections occur to the left of p, the order given by T is correct at all points to the left of p. Because no three segments intersect at the same point, a and b become consecutive in the total preorder at some sweep line z.[2] Moreover, z is to the left of p or goes through p. Some segment endpoint q on sweep line z

[2] If we allow three segments to intersect at the same point, there may be an intervening segment c that intersects both a and b at point p. That is, we may have $a \succeq_w c$ and $c \succeq_w b$ for all sweep lines w to the left of p for which $a \succeq_w b$. Exercise 33.2-8 asks you to show that ANY-SEGMENTS-INTERSECT is correct even if three segments do intersect at the same point.

is the event point at which a and b become consecutive in the total preorder. If p is on sweep line z, then $q = p$. If p is not on sweep line z, then q is to the left of p. In either case, the order given by T is correct just before encountering q. (Here is where we use the lexicographic order in which the algorithm processes event points. Because p is the lowest of the leftmost intersection points, even if p is on sweep line z and some other intersection point p' is on z, event point $q = p$ is processed before the other intersection p' can interfere with the total preorder T. Moreover, even if p is the left endpoint of one segment, say a, and the right endpoint of the other segment, say b, because left endpoint events occur before right endpoint events, segment b is in T upon first encountering segment a.) Either event point q is processed by ANY-SEGMENTS-INTERSECT or it is not processed.

If q is processed by ANY-SEGMENTS-INTERSECT, only two possible actions may occur:

1. Either a or b is inserted into T, and the other segment is above or below it in the total preorder. Lines 4–7 detect this case.
2. Segments a and b are already in T, and a segment between them in the total preorder is deleted, making a and b become consecutive. Lines 8–11 detect this case.

In either case, we find the intersection p and ANY-SEGMENTS-INTERSECT returns TRUE.

If event point q is not processed by ANY-SEGMENTS-INTERSECT, the procedure must have returned before processing all event points. This situation could have occurred only if ANY-SEGMENTS-INTERSECT had already found an intersection and returned TRUE.

Thus, if there is an intersection, ANY-SEGMENTS-INTERSECT returns TRUE. As we have already seen, if ANY-SEGMENTS-INTERSECT returns TRUE, there is an intersection. Therefore, ANY-SEGMENTS-INTERSECT always returns a correct answer.

Running time

If set S contains n segments, then ANY-SEGMENTS-INTERSECT runs in time $O(n \lg n)$. Line 1 takes $O(1)$ time. Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort. The **for** loop of lines 3–11 iterates at most once per event point, and so with $2n$ event points, the loop iterates at most $2n$ times. Each iteration takes $O(\lg n)$ time, since each red-black-tree operation takes $O(\lg n)$ time and, using the method of Section 33.1, each intersection test takes $O(1)$ time. The total time is thus $O(n \lg n)$.

Exercises

33.2-1

Show that a set of n line segments may contain $\Theta(n^2)$ intersections.

33.2-2

Given two segments a and b that are comparable at x, show how to determine in $O(1)$ time which of $a \succcurlyeq_x b$ or $b \succcurlyeq_x a$ holds. Assume that neither segment is vertical. (*Hint:* If a and b do not intersect, you can just use cross products. If a and b intersect—which you can of course determine using only cross products—you can still use only addition, subtraction, and multiplication, avoiding division. Of course, in the application of the $\succcurlyeq_x$ relation used here, if a and b intersect, we can just stop and declare that we have found an intersection.)

33.2-3

Professor Mason suggests that we modify ANY-SEGMENTS-INTERSECT so that instead of returning upon finding an intersection, it prints the segments that intersect and continues on to the next iteration of the **for** loop. The professor calls the resulting procedure PRINT-INTERSECTING-SEGMENTS and claims that it prints all intersections, from left to right, as they occur in the set of line segments. Professor Dixon disagrees, claiming that Professor Mason's idea is incorrect. Which professor is right? Will PRINT-INTERSECTING-SEGMENTS always find the leftmost intersection first? Will it always find all the intersections?

33.2-4

Give an $O(n \lg n)$-time algorithm to determine whether an n-vertex polygon is simple.

33.2-5

Give an $O(n \lg n)$-time algorithm to determine whether two simple polygons with a total of n vertices intersect.

33.2-6

A ***disk*** consists of a circle plus its interior and is represented by its center point and radius. Two disks intersect if they have any point in common. Give an $O(n \lg n)$-time algorithm to determine whether any two disks in a set of n intersect.

33.2-7

Given a set of n line segments containing a total of k intersections, show how to output all k intersections in $O((n + k) \lg n)$ time.

33.2-8

Argue that ANY-SEGMENTS-INTERSECT works correctly even if three or more segments intersect at the same point.

33.2-9

Show that ANY-SEGMENTS-INTERSECT works correctly in the presence of vertical segments if we treat the bottom endpoint of a vertical segment as if it were a left endpoint and the top endpoint as if it were a right endpoint. How does your answer to Exercise 33.2-2 change if we allow vertical segments?

33.3 Finding the convex hull

The ***convex hull*** of a set Q of points, denoted by $\mathrm{CH}(Q)$, is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. (See Exercise 33.1-5 for a precise definition of a convex polygon.) We implicitly assume that all points in the set Q are unique and that Q contains at least three points which are not colinear. Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails. Figure 33.6 shows a set of points and its convex hull.

In this section, we shall present two algorithms that compute the convex hull of a set of n points. Both algorithms output the vertices of the convex hull in counterclockwise order. The first, known as Graham's scan, runs in $O(n \lg n)$ time. The second, called Jarvis's march, runs in $O(nh)$ time, where h is the number of vertices of the convex hull. As Figure 33.6 illustrates, every vertex of $\mathrm{CH}(Q)$ is a

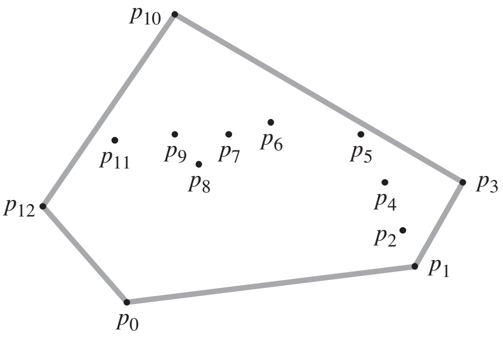


Figure 33.6 A set of points $Q = \{p_0, p_1, \ldots, p_{12}\}$ with its convex hull $\mathrm{CH}(Q)$ in gray.

point in Q. Both algorithms exploit this property, deciding which vertices in Q to keep as vertices of the convex hull and which vertices in Q to reject.

We can compute convex hulls in $O(n \lg n)$ time by any one of several methods. Both Graham's scan and Jarvis's march use a technique called "rotational sweep," processing vertices in the order of the polar angles they form with a reference vertex. Other methods include the following:

- In the ***incremental method***, we first sort the points from left to right, yielding a sequence $\langle p_1, p_2, \ldots, p_n \rangle$. At the ith stage, we update the convex hull of the $i - 1$ leftmost points, $\text{CH}(\{p_1, p_2, \ldots, p_{i-1}\})$, according to the ith point from the left, thus forming $\text{CH}(\{p_1, p_2, \ldots, p_i\})$. Exercise 33.3-6 asks you how to implement this method to take a total of $O(n \lg n)$ time.
- In the ***divide-and-conquer method***, we divide the set of n points in $\Theta(n)$ time into two subsets, one containing the leftmost $\lceil n/2 \rceil$ points and one containing the rightmost $\lfloor n/2 \rfloor$ points, recursively compute the convex hulls of the subsets, and then, by means of a clever method, combine the hulls in $O(n)$ time. The running time is described by the familiar recurrence $T(n) = 2T(n/2) + O(n)$, and so the divide-and-conquer method runs in $O(n \lg n)$ time.
- The ***prune-and-search method*** is similar to the worst-case linear-time median algorithm of Section 9.3. With this method, we find the upper portion (or "upper chain") of the convex hull by repeatedly throwing out a constant fraction of the remaining points until only the upper chain of the convex hull remains. We then do the same for the lower chain. This method is asymptotically the fastest: if the convex hull contains h vertices, it runs in only $O(n \lg h)$ time.

Computing the convex hull of a set of points is an interesting problem in its own right. Moreover, algorithms for some other computational-geometry problems start by computing a convex hull. Consider, for example, the two-dimensional ***farthest-pair problem***: we are given a set of n points in the plane and wish to find the two points whose distance from each other is maximum. As Exercise 33.3-3 asks you to prove, these two points must be vertices of the convex hull. Although we won't prove it here, we can find the farthest pair of vertices of an n-vertex convex polygon in $O(n)$ time. Thus, by computing the convex hull of the n input points in $O(n \lg n)$ time and then finding the farthest pair of the resulting convex-polygon vertices, we can find the farthest pair of points in any set of n points in $O(n \lg n)$ time.

Graham's scan

Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. It pushes each point of the input set Q onto the stack one time, and it eventually pops from the stack each point that is not a vertex of CH(Q). When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.

The procedure Graham-Scan takes as input a set Q of points, where $|Q| \geq 3$. It calls the functions Top(S), which returns the point on top of stack S without changing S, and Next-To-Top(S), which returns the point one entry below the top of stack S without changing S. As we shall prove in a moment, the stack S returned by Graham-Scan contains, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

```
Graham-Scan($Q$)
 1  let $p_0$ be the point in $Q$ with the minimum $y$-coordinate,
        or the leftmost such point in case of a tie
 2  let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in $Q$,
        sorted by polar angle in counterclockwise order around $p_0$
        (if more than one point has the same angle, remove all but
        the one that is farthest from $p_0$)
 3  let $S$ be an empty stack
 4  Push($p_0, S$)
 5  Push($p_1, S$)
 6  Push($p_2, S$)
 7  for $i = 3$ to $m$
 8      while the angle formed by points Next-To-Top($S$), Top($S$),
                and $p_i$ makes a nonleft turn
 9          Pop($S$)
10      Push($p_i, S$)
11  return $S$
```

Figure 33.7 illustrates the progress of Graham-Scan. Line 1 chooses point p_0 as the point with the lowest y-coordinate, picking the leftmost such point in case of a tie. Since there is no point in Q that is below p_0 and any other points with the same y-coordinate are to its right, p_0 must be a vertex of CH(Q). Line 2 sorts the remaining points of Q by polar angle relative to p_0, using the same method—comparing cross products—as in Exercise 33.1-3. If two or more points have the same polar angle relative to p_0, all but the farthest such point are convex combinations of p_0 and the farthest point, and so we remove them entirely from consideration. We let m denote the number of points other than p_0 that remain. The polar angle, measured in radians, of each point in Q relative to p_0 is in the half-open interval $[0, \pi)$. Since the points are sorted according to polar angles, they are sorted in counterclockwise order relative to p_0. We designate this sorted sequence of points by $\langle p_1, p_2, \ldots, p_m \rangle$. Note that points p_1 and p_m are vertices

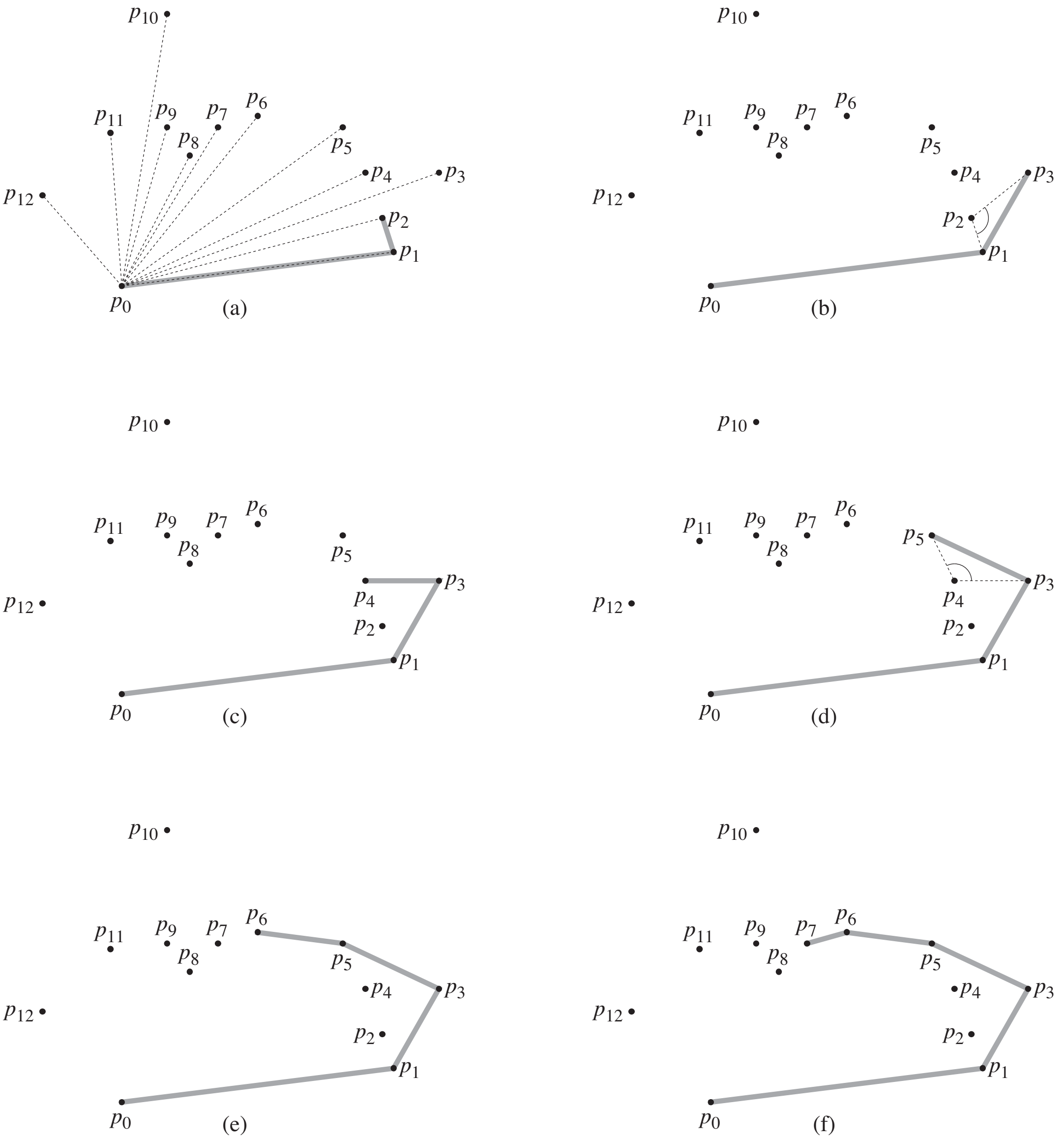


Figure 33.7 The execution of GRAHAM-SCAN on the set Q of Figure 33.6. The current convex hull contained in stack S is shown in gray at each step. **(a)** The sequence $\langle p_1, p_2, \ldots, p_{12} \rangle$ of points numbered in order of increasing polar angle relative to p_0, and the initial stack S containing p_0, p_1, and p_2. **(b)–(k)** Stack S after each iteration of the **for** loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack. In part (h), for example, the right turn at angle $\angle p_7 p_8 p_9$ causes p_8 to be popped, and then the right turn at angle $\angle p_6 p_7 p_9$ causes p_7 to be popped.

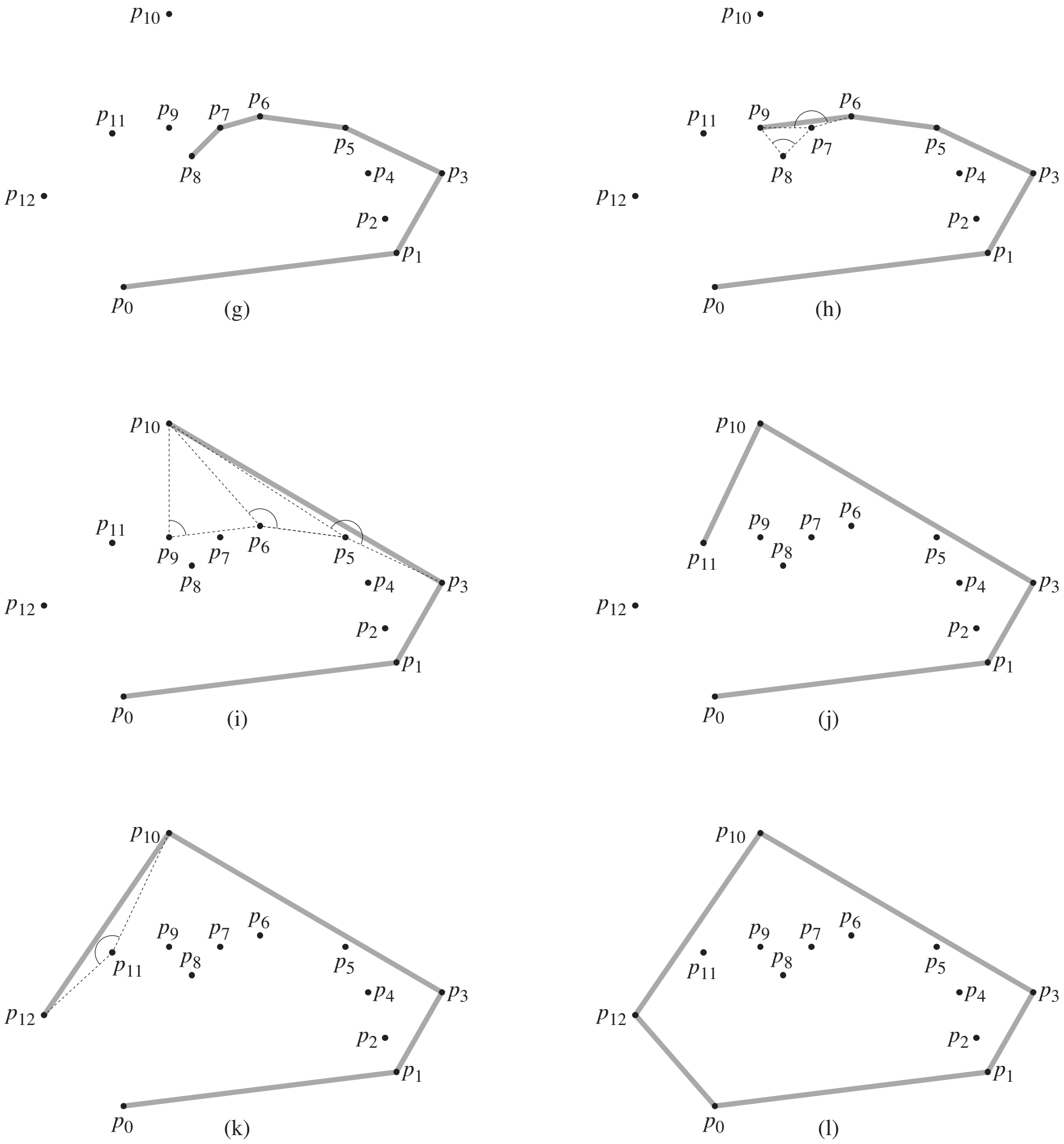


Figure 33.7, continued **(l)** The convex hull returned by the procedure, which matches that of Figure 33.6.

of CH(Q) (see Exercise 33.3-1). Figure 33.7(a) shows the points of Figure 33.6 sequentially numbered in order of increasing polar angle relative to p_0.

The remainder of the procedure uses the stack S. Lines 3–6 initialize the stack to contain, from bottom to top, the first three points p_0, p_1, and p_2. Figure 33.7(a) shows the initial stack S. The **for** loop of lines 7–10 iterates once for each point in the subsequence $\langle p_3, p_4, \ldots, p_m \rangle$. We shall see that after processing point p_i, stack S contains, from bottom to top, the vertices of CH($\{p_0, p_1, \ldots, p_i\}$) in counterclockwise order. The **while** loop of lines 8–9 removes points from the stack if we find them not to be vertices of the convex hull. When we traverse the convex hull counterclockwise, we should make a left turn at each vertex. Thus, each time the **while** loop finds a vertex at which we make a nonleft turn, we pop the vertex from the stack. (By checking for a nonleft turn, rather than just a right turn, this test precludes the possibility of a straight angle at a vertex of the resulting convex hull. We want no straight angles, since no vertex of a convex polygon may be a convex combination of other vertices of the polygon.) After we pop all vertices that have nonleft turns when heading toward point p_i, we push p_i onto the stack. Figures 33.7(b)–(k) show the state of the stack S after each iteration of the **for** loop. Finally, GRAHAM-SCAN returns the stack S in line 11. Figure 33.7(l) shows the corresponding convex hull.

The following theorem formally proves the correctness of GRAHAM-SCAN.

Theorem 33.1 (Correctness of Graham's scan)
If GRAHAM-SCAN executes on a set Q of points, where $|Q| \geq 3$, then at termination, the stack S consists of, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

Proof After line 2, we have the sequence of points $\langle p_1, p_2, \ldots, p_m \rangle$. Let us define, for $i = 2, 3, \ldots, m$, the subset of points $Q_i = \{p_0, p_1, \ldots, p_i\}$. The points in $Q - Q_m$ are those that were removed because they had the same polar angle relative to p_0 as some point in Q_m; these points are not in CH(Q), and so CH(Q_m) $=$ CH(Q). Thus, it suffices to show that when GRAHAM-SCAN terminates, the stack S consists of the vertices of CH(Q_m) in counterclockwise order, when listed from bottom to top. Note that just as p_0, p_1, and p_m are vertices of CH(Q), the points p_0, p_1, and p_i are all vertices of CH(Q_i).

The proof uses the following loop invariant:

> At the start of each iteration of the **for** loop of lines 7–10, stack S consists of, from bottom to top, exactly the vertices of CH(Q_{i-1}) in counterclockwise order.

Initialization: The invariant holds the first time we execute line 7, since at that time, stack S consists of exactly the vertices of $Q_2 = Q_{i-1}$, and this set of three

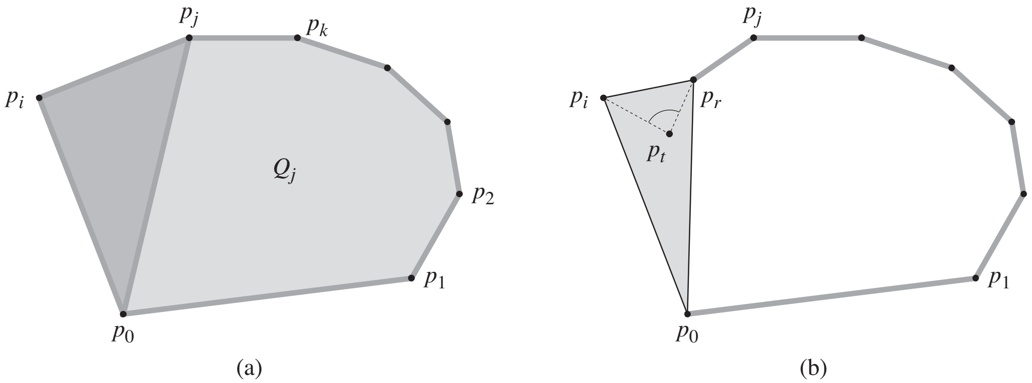


Figure 33.8 The proof of correctness of GRAHAM-SCAN. **(a)** Because p_i's polar angle relative to p_0 is greater than p_j's polar angle, and because the angle $\angle p_k p_j p_i$ makes a left turn, adding p_i to $\text{CH}(Q_j)$ gives exactly the vertices of $\text{CH}(Q_j \cup \{p_i\})$. **(b)** If the angle $\angle p_r p_t p_i$ makes a nonleft turn, then p_t is either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of the triangle, which means that it cannot be a vertex of $\text{CH}(Q_i)$.

vertices forms its own convex hull. Moreover, they appear in counterclockwise order from bottom to top.

Maintenance: Entering an iteration of the **for** loop, the top point on stack S is p_{i-1}, which was pushed at the end of the previous iteration (or before the first iteration, when $i = 3$). Let p_j be the top point on S after executing the while loop of lines 8–9 but before line 10 pushes p_i, and let p_k be the point just below p_j on S. At the moment that p_j is the top point on S and we have not yet pushed p_i, stack S contains exactly the same points it contained after iteration j of the **for** loop. By the loop invariant, therefore, S contains exactly the vertices of $\text{CH}(Q_j)$ at that moment, and they appear in counterclockwise order from bottom to top.

Let us continue to focus on this moment just before pushing p_i. We know that p_i's polar angle relative to p_0 is greater than p_j's polar angle and that the angle $\angle p_k p_j p_i$ makes a left turn (otherwise we would have popped p_j). Therefore, because S contains exactly the vertices of $\text{CH}(Q_j)$, we see from Figure 33.8(a) that once we push p_i, stack S will contain exactly the vertices of $\text{CH}(Q_j \cup \{p_i\})$, still in counterclockwise order from bottom to top.

We now show that $\text{CH}(Q_j \cup \{p_i\})$ is the same set of points as $\text{CH}(Q_i)$. Consider any point p_t that was popped during iteration i of the **for** loop, and let p_r be the point just below p_t on stack S at the time p_t was popped (p_r might be p_j). The angle $\angle p_r p_t p_i$ makes a nonleft turn, and the polar angle of p_t relative to p_0 is greater than the polar angle of p_r. As Figure 33.8(b) shows, p_t must be either in the interior of the triangle formed by p_0, p_r, and p_i or on a side of this triangle (but it is not a vertex of the triangle). Clearly, since p_t is within a triangle formed by three other points of Q_i, it cannot be a vertex of $\text{CH}(Q_i)$. Since p_t is not a vertex of $\text{CH}(Q_i)$, we have that

$$\text{CH}(Q_i - \{p_t\}) = \text{CH}(Q_i) . \tag{33.1}$$

Let P_i be the set of points that were popped during iteration i of the **for** loop. Since the equality (33.1) applies for all points in P_i, we can apply it repeatedly to show that $\text{CH}(Q_i - P_i) = \text{CH}(Q_i)$. But $Q_i - P_i = Q_j \cup \{p_i\}$, and so we conclude that $\text{CH}(Q_j \cup \{p_i\}) = \text{CH}(Q_i - P_i) = \text{CH}(Q_i)$.

We have shown that once we push p_i, stack S contains exactly the vertices of $\text{CH}(Q_i)$ in counterclockwise order from bottom to top. Incrementing i will then cause the loop invariant to hold for the next iteration.

Termination: When the loop terminates, we have $i = m + 1$, and so the loop invariant implies that stack S consists of exactly the vertices of $\text{CH}(Q_m)$, which is $\text{CH}(Q)$, in counterclockwise order from bottom to top. This completes the proof. ■

We now show that the running time of GRAHAM-SCAN is $O(n \lg n)$, where $n = |Q|$. Line 1 takes $\Theta(n)$ time. Line 2 takes $O(n \lg n)$ time, using merge sort or heapsort to sort the polar angles and the cross-product method of Section 33.1 to compare angles. (We can remove all but the farthest point with the same polar angle in total of $O(n)$ time over all n points.) Lines 3–6 take $O(1)$ time. Because $m \le n - 1$, the **for** loop of lines 7–10 executes at most $n - 3$ times. Since PUSH takes $O(1)$ time, each iteration takes $O(1)$ time exclusive of the time spent in the **while** loop of lines 8–9, and thus overall the **for** loop takes $O(n)$ time exclusive of the nested **while** loop.

We use aggregate analysis to show that the **while** loop takes $O(n)$ time overall. For $i = 0, 1, \ldots, m$, we push each point p_i onto stack S exactly once. As in the analysis of the MULTIPOP procedure of Section 17.1, we observe that we can pop at most the number of items that we push. At least three points—p_0, p_1, and p_m—are never popped from the stack, so that in fact at most $m - 2$ POP operations are performed in total. Each iteration of the **while** loop performs one POP, and so there are at most $m - 2$ iterations of the **while** loop altogether. Since the test in line 8 takes $O(1)$ time, each call of POP takes $O(1)$ time, and $m \le n - 1$, the total time taken by the **while** loop is $O(n)$. Thus, the running time of GRAHAM-SCAN is $O(n \lg n)$.

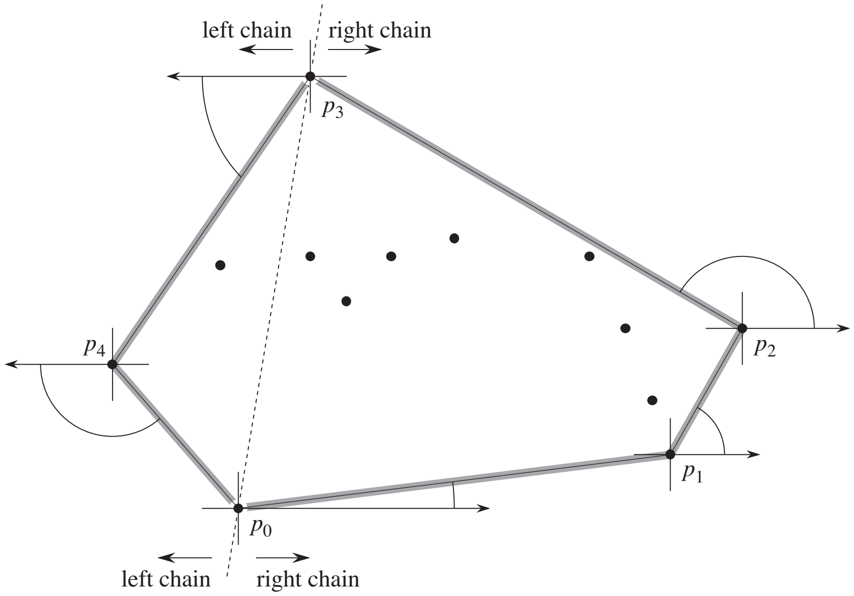


Figure 33.9 The operation of Jarvis's march. We choose the first vertex as the lowest point p_0. The next vertex, p_1, has the smallest polar angle of any point with respect to p_0. Then, p_2 has the smallest polar angle with respect to p_1. The right chain goes as high as the highest point p_3. Then, we construct the left chain by finding smallest polar angles with respect to the negative x-axis.

Jarvis's march

Jarvis's march computes the convex hull of a set Q of points by a technique known as ***package wrapping*** (or ***gift wrapping***). The algorithm runs in time $O(nh)$, where h is the number of vertices of $\text{CH}(Q)$. When h is $o(\lg n)$, Jarvis's march is asymptotically faster than Graham's scan.

Intuitively, Jarvis's march simulates wrapping a taut piece of paper around the set Q. We start by taping the end of the paper to the lowest point in the set, that is, to the same point p_0 with which we start Graham's scan. We know that this point must be a vertex of the convex hull. We pull the paper to the right to make it taut, and then we pull it higher until it touches a point. This point must also be a vertex of the convex hull. Keeping the paper taut, we continue in this way around the set of vertices until we come back to our original point p_0.

More formally, Jarvis's march builds a sequence $H = \langle p_0, p_1, \ldots, p_{h-1} \rangle$ of the vertices of $\text{CH}(Q)$. We start with p_0. As Figure 33.9 shows, the next vertex p_1 in the convex hull has the smallest polar angle with respect to p_0. (In case of ties, we choose the point farthest from p_0.) Similarly, p_2 has the smallest polar angle with respect to p_1, and so on. When we reach the highest vertex, say p_k (breaking ties by choosing the farthest such vertex), we have constructed, as Figure 33.9 shows, the ***right chain*** of CH(Q). To construct the ***left chain***, we start at p_k and choose p_{k+1} as the point with the smallest polar angle with respect to p_k, but *from the negative x-axis*. We continue on, forming the left chain by taking polar angles from the negative x-axis, until we come back to our original vertex p_0.

We could implement Jarvis's march in one conceptual sweep around the convex hull, that is, without separately constructing the right and left chains. Such implementations typically keep track of the angle of the last convex-hull side chosen and require the sequence of angles of hull sides to be strictly increasing (in the range of 0 to 2π radians). The advantage of constructing separate chains is that we need not explicitly compute angles; the techniques of Section 33.1 suffice to compare angles.

If implemented properly, Jarvis's march has a running time of $O(nh)$. For each of the h vertices of CH(Q), we find the vertex with the minimum polar angle. Each comparison between polar angles takes $O(1)$ time, using the techniques of Section 33.1. As Section 9.1 shows, we can compute the minimum of n values in $O(n)$ time if each comparison takes $O(1)$ time. Thus, Jarvis's march takes $O(nh)$ time.

Exercises

33.3-1

Prove that in the procedure GRAHAM-SCAN, points p_1 and p_m must be vertices of CH(Q).

33.3-2

Consider a model of computation that supports addition, comparison, and multiplication and for which there is a lower bound of $\Omega(n \lg n)$ to sort n numbers. Prove that $\Omega(n \lg n)$ is a lower bound for computing, in order, the vertices of the convex hull of a set of n points in such a model.

33.3-3

Given a set of points Q, prove that the pair of points farthest from each other must be vertices of CH(Q).

33.3-4

For a given polygon P and a point q on its boundary, the ***shadow*** of q is the set of points r such that the segment $\overline{qr}$ is entirely on the boundary or in the interior of P. As Figure 33.10 illustrates, a polygon P is ***star-shaped*** if there exists a point p in the interior of P that is in the shadow of every point on the boundary of P. The set of all such points p is called the ***kernel*** of P. Given an n-vertex,

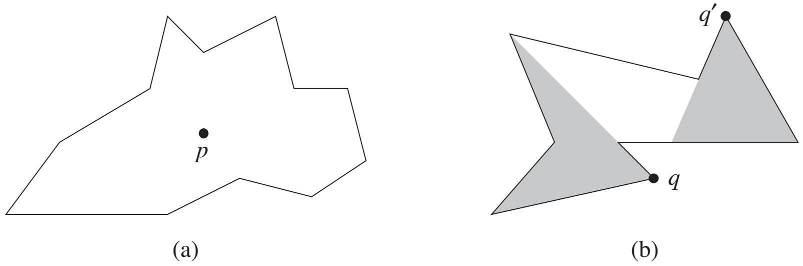


Figure 33.10 The definition of a star-shaped polygon, for use in Exercise 33.3-4. **(a)** A star-shaped polygon. The segment from point p to any point q on the boundary intersects the boundary only at q. **(b)** A non-star-shaped polygon. The shaded region on the left is the shadow of q, and the shaded region on the right is the shadow of q'. Since these regions are disjoint, the kernel is empty.

star-shaped polygon P specified by its vertices in counterclockwise order, show how to compute CH(P) in $O(n)$ time.

33.3-5
In the ***on-line convex-hull problem***, we are given the set Q of n points one point at a time. After receiving each point, we compute the convex hull of the points seen so far. Obviously, we could run Graham's scan once for each point, with a total running time of $O(n^2 \lg n)$. Show how to solve the on-line convex-hull problem in a total of $O(n^2)$ time.

33.3-6 ★
Show how to implement the incremental method for computing the convex hull of n points so that it runs in $O(n \lg n)$ time.

33.4 Finding the closest pair of points

We now consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points. "Closest" refers to the usual euclidean distance: the distance between points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Two points in set Q may be coincident, in which case the distance between them is zero. This problem has applications in, for example, traffic-control systems. A system for controlling air or sea traffic might need to identify the two closest vehicles in order to detect potential collisions.

A brute-force closest-pair algorithm simply looks at all the $\binom{n}{2} = \Theta(n^2)$ pairs of points. In this section, we shall describe a divide-and-conquer algorithm for this problem, whose running time is described by the familiar recurrence $T(n) = 2T(n/2) + O(n)$. Thus, this algorithm uses only $O(n \lg n)$ time.

The divide-and-conquer algorithm

Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrays X and Y, each of which contains all the points of the input subset P. The points in array X are sorted so that their x-coordinates are monotonically increasing. Similarly, array Y is sorted by monotonically increasing y-coordinate. Note that in order to attain the $O(n \lg n)$ time bound, we cannot afford to sort in each recursive call; if we did, the recurrence for the running time would be $T(n) = 2T(n/2) + O(n \lg n)$, whose solution is $T(n) = O(n \lg^2 n)$. (Use the version of the master method given in Exercise 4.6-2.) We shall see a little later how to use "presorting" to maintain this sorted property without actually sorting in each recursive call.

A given recursive invocation with inputs P, X, and Y first checks whether $|P| \le 3$. If so, the invocation simply performs the brute-force method described above: try all $\binom{|P|}{2}$ pairs of points and return the closest pair. If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows.

Divide: Find a vertical line l that bisects the point set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $|P_R| = \lfloor |P|/2 \rfloor$, all points in P_L are on or to the left of line l, and all points in P_R are on or to the right of l. Divide the array X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide the array Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.

Conquer: Having divided P into P_L and P_R, make two recursive calls, one to find the closest pair of points in P_L and the other to find the closest pair of points in P_R. The inputs to the first call are the subset P_L and arrays X_L and Y_L; the second call receives the inputs P_R, X_R, and Y_R. Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.

Combine: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R. The algorithm determines whether there is a pair with one point in P_L and the other point in P_R and whose distance is less than δ. Observe that if a pair of points has distance less than δ, both points of the pair must be within δ units of line l. Thus, as Figure 33.11(a) shows, they both must reside in the 2δ-wide vertical strip centered at line l. To find such a pair, if one exists, we do the following:

1. Create an array Y', which is the array Y with all points not in the 2δ-wide vertical strip removed. The array Y' is sorted by y-coordinate, just as Y is.
2. For each point p in the array Y', try to find points in Y' that are within δ units of p. As we shall see shortly, only the 7 points in Y' that follow p need be considered. Compute the distance from p to each of these 7 points, and keep track of the closest-pair distance δ' found over all pairs of points in Y'.
3. If $\delta' < \delta$, then the vertical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ found by the recursive calls.

The above description omits some implementation details that are necessary to achieve the $O(n \lg n)$ running time. After proving the correctness of the algorithm, we shall show how to implement the algorithm to achieve the desired time bound.

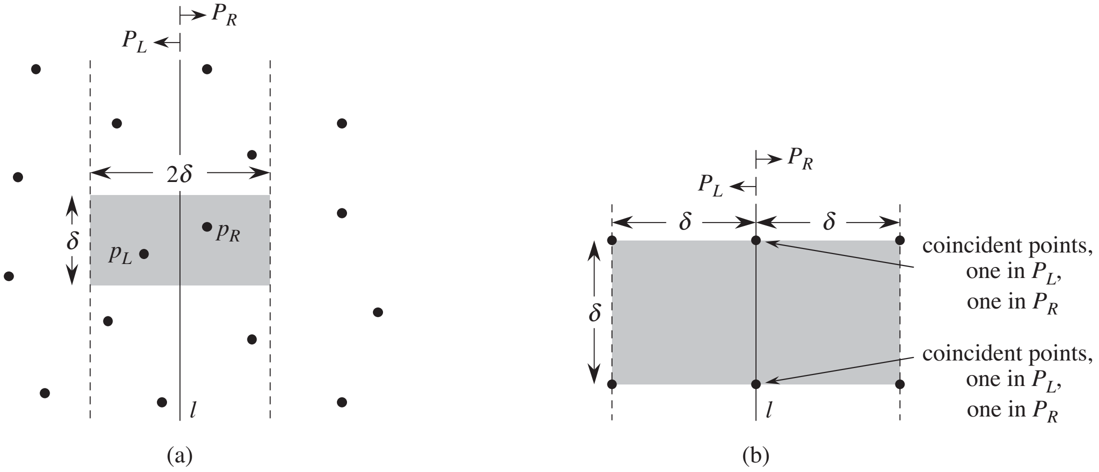
Correctness

The correctness of this closest-pair algorithm is obvious, except for two aspects. First, by bottoming out the recursion when $|P| \leq 3$, we ensure that we never try to solve a subproblem consisting of only one point. The second aspect is that we need only check the 7 points following each point p in array Y'; we shall now prove this property.

Suppose that at some level of the recursion, the closest pair of points is $p_L \in P_L$ and $p_R \in P_R$. Thus, the distance δ' between p_L and p_R is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away. Similarly, p_R is on or to the right of l and less than δ units away. Moreover, p_L and p_R are within δ units of each other vertically. Thus, as Figure 33.11(a) shows, p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at line l. (There may be other points within this rectangle as well.)

We next show that at most 8 points of P can reside within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square forming the left half of this rectangle. Since all points within P_L are at least δ units apart, at most 4 points can reside within this square; Figure 33.11(b) shows how. Similarly, at most 4 points in P_R can reside within the $\delta \times \delta$ square forming the right half of the rectangle. Thus, at most 8 points of P can reside within the $\delta \times 2\delta$ rectangle. (Note that since points on line l may be in either P_L or P_R, there may be up to 4 points on l. This limit is achieved if there are two pairs of coincident points such that each pair consists of one point from P_L and one point from P_R, one pair is at the intersection of l and the top of the rectangle, and the other pair is where l intersects the bottom of the rectangle.)

Having shown that at most 8 points of P can reside within the rectangle, we can easily see why we need to check only the 7 points following each point in the array Y'. Still assuming that the closest pair is p_L and p_R, let us assume without



P_R
P_L
2δ
δ
p_L
p_R
l
(a)
P_R
P_L
δ
δ
δ
coincident points, one in P_L, one in P_R
coincident points, one in P_L, one in P_R
l
(b)

Figure 33.11 Key concepts in the proof that the closest-pair algorithm needs to check only 7 points following each point in the array Y'. **(a)** If $p_L \in P_L$ and $p_R \in P_R$ are less than δ units apart, they must reside within a $\delta \times 2\delta$ rectangle centered at line l. **(b)** How 4 points that are pairwise at least δ units apart can all reside within a $\delta \times \delta$ square. On the left are 4 points in P_L, and on the right are 4 points in P_R. The $\delta \times 2\delta$ rectangle can contain 8 points if the points shown on line l are actually pairs of coincident points with one point in P_L and one in P_R.

loss of generality that p_L precedes p_R in array Y'. Then, even if p_L occurs as early as possible in Y' and p_R occurs as late as possible, p_R is in one of the 7 positions following p_L. Thus, we have shown the correctness of the closest-pair algorithm.

Implementation and running time

As we have noted, our goal is to have the recurrence for the running time be $T(n) = 2T(n/2) + O(n)$, where $T(n)$ is the running time for a set of n points. The main difficulty comes from ensuring that the arrays X_L, X_R, Y_L, and Y_R, which are passed to recursive calls, are sorted by the proper coordinate and also that the array Y' is sorted by y-coordinate. (Note that if the array X that is received by a recursive call is already sorted, then we can easily divide set P into P_L and P_R in linear time.)

The key observation is that in each call, we wish to form a sorted subset of a sorted array. For example, a particular invocation receives the subset P and the array Y, sorted by y-coordinate. Having partitioned P into P_L and P_R, it needs to form the arrays Y_L and Y_R, which are sorted by y-coordinate, in linear time. We can view the method as the opposite of the MERGE procedure from merge sort in

Section 2.3.1: we are splitting a sorted array into two sorted arrays. The following pseudocode gives the idea.

```
let $Y_L[1\,..\,Y.length]$ and $Y_R[1\,..\,Y.length]$ be new arrays
$Y_L.length = Y_R.length = 0$
for $i = 1$ to $Y.length$
    if $Y[i] \in P_L$
        $Y_L.length = Y_L.length + 1$
        $Y_L[Y_L.length] = Y[i]$
    else $Y_R.length = Y_R.length + 1$
        $Y_R[Y_R.length] = Y[i]$
```

We simply examine the points in array Y in order. If a point $Y[i]$ is in P_L, we append it to the end of array Y_L; otherwise, we append it to the end of array Y_R. Similar pseudocode works for forming arrays X_L, X_R, and Y'.

The only remaining question is how to get the points sorted in the first place. We ***presort*** them; that is, we sort them once and for all *before* the first recursive call. We pass these sorted arrays into the first recursive call, and from there we whittle them down through the recursive calls as necessary. Presorting adds an additional $O(n \lg n)$ term to the running time, but now each step of the recursion takes linear time exclusive of the recursive calls. Thus, if we let $T(n)$ be the running time of each recursive step and $T'(n)$ be the running time of the entire algorithm, we get $T'(n) = T(n) + O(n \lg n)$ and

$$T(n) = \begin{cases} 2T(n/2) + O(n) & \text{if } n > 3\,, \\ O(1) & \text{if } n \le 3\,. \end{cases}$$

Thus, $T(n) = O(n \lg n)$ and $T'(n) = O(n \lg n)$.

Exercises

33.4-1

Professor Williams comes up with a scheme that allows the closest-pair algorithm to check only 5 points following each point in array Y'. The idea is always to place points on line l into set P_L. Then, there cannot be pairs of coincident points on line l with one point in P_L and one in P_R. Thus, at most 6 points can reside in the $\delta \times 2\delta$ rectangle. What is the flaw in the professor's scheme?

33.4-2

Show that it actually suffices to check only the points in the 5 array positions following each point in the array Y'.

33.4-3

We can define the distance between two points in ways other than euclidean. In the plane, the ***L_m-distance*** between points p_1 and p_2 is given by the expression $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Euclidean distance, therefore, is L_2-distance. Modify the closest-pair algorithm to use the L_1-distance, which is also known as the ***Manhattan distance***.

33.4-4

Given two points p_1 and p_2 in the plane, the L_∞-distance between them is given by $\max(|x_1 - x_2|, |y_1 - y_2|)$. Modify the closest-pair algorithm to use the L_∞-distance.

33.4-5

Suppose that $\Omega(n)$ of the points given to the closest-pair algorithm are covertical. Show how to determine the sets P_L and P_R and how to determine whether each point of Y is in P_L or P_R so that the running time for the closest-pair algorithm remains $O(n \lg n)$.

33.4-6

Suggest a change to the closest-pair algorithm that avoids presorting the Y array but leaves the running time as $O(n \lg n)$. (*Hint:* Merge sorted arrays Y_L and Y_R to form the sorted array Y.)

Problems

33-1 Convex layers

Given a set Q of points in the plane, we define the ***convex layers*** of Q inductively. The first convex layer of Q consists of those points in Q that are vertices of $\text{CH}(Q)$. For $i > 1$, define Q_i to consist of the points of Q with all points in convex layers $1, 2, \ldots, i-1$ removed. Then, the ith convex layer of Q is $\text{CH}(Q_i)$ if $Q_i \neq \emptyset$ and is undefined otherwise.

a. Give an $O(n^2)$-time algorithm to find the convex layers of a set of n points.

b. Prove that $\Omega(n \lg n)$ time is required to compute the convex layers of a set of n points with any model of computation that requires $\Omega(n \lg n)$ time to sort n real numbers.

33-2 Maximal layers

Let Q be a set of n points in the plane. We say that point (x, y) ***dominates*** point (x', y') if $x \geq x'$ and $y \geq y'$. A point in Q that is dominated by no other points in Q is said to be ***maximal***. Note that Q may contain many maximal points, which can be organized into ***maximal layers*** as follows. The first maximal layer L_1 is the set of maximal points of Q. For $i > 1$, the ith maximal layer L_i is the set of maximal points in $Q - \bigcup_{j=1}^{i-1} L_j$.

Suppose that Q has k nonempty maximal layers, and let y_i be the y-coordinate of the leftmost point in L_i for $i = 1, 2, \ldots, k$. For now, assume that no two points in Q have the same x- or y-coordinate.

a. Show that $y_1 > y_2 > \cdots > y_k$.

Consider a point (x, y) that is to the left of any point in Q and for which y is distinct from the y-coordinate of any point in Q. Let $Q' = Q \cup \{(x, y)\}$.

b. Let j be the minimum index such that $y_j < y$, unless $y < y_k$, in which case we let $j = k + 1$. Show that the maximal layers of Q' are as follows:

- If $j \leq k$, then the maximal layers of Q' are the same as the maximal layers of Q, except that L_j also includes (x, y) as its new leftmost point.
- If $j = k+1$, then the first k maximal layers of Q' are the same as for Q, but in addition, Q' has a nonempty $(k + 1)$st maximal layer: $L_{k+1} = \{(x, y)\}$.

c. Describe an $O(n \lg n)$-time algorithm to compute the maximal layers of a set Q of n points. (*Hint:* Move a sweep line from right to left.)

d. Do any difficulties arise if we now allow input points to have the same x- or y-coordinate? Suggest a way to resolve such problems.

33-3 Ghostbusters and ghosts

A group of n Ghostbusters is battling n ghosts. Each Ghostbuster carries a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The Ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n Ghostbuster-ghost pairs, and then simultaneously each Ghostbuster will shoot a stream at his chosen ghost. As we all know, it is *very* dangerous to let streams cross, and so the Ghostbusters must choose pairings for which no streams will cross.

Assume that the position of each Ghostbuster and each ghost is a fixed point in the plane and that no three positions are colinear.

a. Argue that there exists a line passing through one Ghostbuster and one ghost such that the number of Ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \lg n)$ time.

b. Give an $O(n^2 \lg n)$-time algorithm to pair Ghostbusters with ghosts in such a way that no streams cross.

33-4 Picking up sticks

Professor Charon has a set of n sticks, which are piled up in some configuration. Each stick is specified by its endpoints, and each endpoint is an ordered triple giving its (x, y, z) coordinates. No stick is vertical. He wishes to pick up all the sticks, one at a time, subject to the condition that he may pick up a stick only if there is no other stick on top of it.

a. Give a procedure that takes two sticks a and b and reports whether a is above, below, or unrelated to b.

b. Describe an efficient algorithm that determines whether it is possible to pick up all the sticks, and if so, provides a legal order in which to pick them up.

33-5 Sparse-hulled distributions

Consider the problem of computing the convex hull of a set of points in the plane that have been drawn according to some known random distribution. Sometimes, the number of points, or size, of the convex hull of n points drawn from such a distribution has expectation $O(n^{1-\epsilon})$ for some constant $\epsilon > 0$. We call such a distribution ***sparse-hulled***. Sparse-hulled distributions include the following:

- Points drawn uniformly from a unit-radius disk. The convex hull has expected size $\Theta(n^{1/3})$.
- Points drawn uniformly from the interior of a convex polygon with k sides, for any constant k. The convex hull has expected size $\Theta(\lg n)$.
- Points drawn according to a two-dimensional normal distribution. The convex hull has expected size $\Theta(\sqrt{\lg n})$.

a. Given two convex polygons with n_1 and n_2 vertices respectively, show how to compute the convex hull of all $n_1 + n_2$ points in $O(n_1 + n_2)$ time. (The polygons may overlap.)

b. Show how to compute the convex hull of a set of n points drawn independently according to a sparse-hulled distribution in $O(n)$ average-case time. (*Hint:* Recursively find the convex hulls of the first $n/2$ points and the second $n/2$ points, and then combine the results.)

Chapter notes

This chapter barely scratches the surface of computational-geometry algorithms and techniques. Books on computational geometry include those by Preparata and Shamos [282], Edelsbrunner [99], and O'Rourke [269].

Although geometry has been studied since antiquity, the development of algorithms for geometric problems is relatively new. Preparata and Shamos note that the earliest notion of the complexity of a problem was given by E. Lemoine in 1902. He was studying euclidean constructions—those using a compass and a ruler—and devised a set of five primitives: placing one leg of the compass on a given point, placing one leg of the compass on a given line, drawing a circle, passing the ruler's edge through a given point, and drawing a line. Lemoine was interested in the number of primitives needed to effect a given construction; he called this amount the "simplicity" of the construction.

The algorithm of Section 33.2, which determines whether any segments intersect, is due to Shamos and Hoey [313].

The original version of Graham's scan is given by Graham [150]. The package-wrapping algorithm is due to Jarvis [189]. Using a decision-tree model of computation, Yao [359] proved a worst-case lower bound of $\Omega(n \lg n)$ for the running time of any convex-hull algorithm. When the number of vertices h of the convex hull is taken into account, the prune-and-search algorithm of Kirkpatrick and Seidel [206], which takes $O(n \lg h)$ time, is asymptotically optimal.

The $O(n \lg n)$-time divide-and-conquer algorithm for finding the closest pair of points is by Shamos and appears in Preparata and Shamos [282]. Preparata and Shamos also show that the algorithm is asymptotically optimal in a decision-tree model.