Part IV Advanced Design and Analysis Techniques

Introduction

This part covers three important techniques used in designing and analyzing efficient algorithms: dynamic programming (Chapter 14), greedy algorithms (Chapter 15), and amortized analysis (Chapter 16). Earlier parts have presented other widely applicable techniques, such as divide-and-conquer, randomization, and how to solve recurrences. The techniques in this part are somewhat more sophisticated, but you will be able to use them solve many computational problems. The themes introduced in this part will recur later in this book.

Dynamic programming typically applies to optimization problems in which you make a set of choices in order to arrive at an optimal solution, each choice generates subproblems of the same form as the original problem, and the same subproblems arise repeatedly. The key strategy is to store the solution to each such subproblem rather than recompute it. Chapter 14 shows how this simple idea can sometimes transform exponential-time algorithms into polynomial-time algorithms.

Like dynamic-programming algorithms, greedy algorithms typically apply to optimization problems in which you make a set of choices in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner, resulting in a faster algorithm than you get with dynamic programming. Chapter 15 will help you determine when the greedy approach works.

The technique of amortized analysis applies to certain algorithms that perform a sequence of similar operations. Instead of bounding the cost of the sequence of operations by bounding the actual cost of each operation separately, an amortized analysis provides a worst-case bound on the actual cost of the entire sequence. One advantage of this approach is that although some operations might be expensive, many others might be cheap. You can use amortized analysis when designing algorithms, since the design of an algorithm and the analysis of its running time are often closely intertwined. Chapter 16 introduces three ways to perform an amortized analysis of an algorithm.

14 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.

Dynamic programming typically applies to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and you want 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.

To develop a dynamic-programming algorithm, 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 you need only the value of an optimal solution, and not the solution itself, then you can omit step 4. When you do perform step 4, it often pays to maintain additional information during step 3 so that you can easily construct an optimal solution.

The sections that follow use the dynamic-programming method to solve some optimization problems. Section 14.1 examines the problem of cutting a rod into

rods of smaller length in a way that maximizes their total value. Section 14.2 shows how to multiply a chain of matrices while performing the fewest total scalar multiplications. Given these examples of dynamic programming, Section 14.3 discusses two key characteristics that a problem must have for dynamic programming to be a viable solution technique. Section 14.4 then shows how to find the longest common subsequence of two sequences via dynamic programming. Finally, Section 14.5 uses dynamic programming to construct binary search trees that are optimal, given a known distribution of keys to be looked up.

14.1 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.

Serling Enterprises has a table giving, for $i = 1, 2, \ldots$, the price p_i in dollars that they charge for a rod of length i inches. The length of each rod in inches is always an integer. Figure 14.1 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. If the price p_n for a rod of length n is large enough, an optimal solution might require no cutting at all.

Consider the case when $n = 4$. Figure 14.2 shows all the ways to cut up a rod of 4 inches in length, including the way with no cuts at all. Cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$, which is optimal.

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

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

[1] If pieces are required to be cut in order of monotonically increasing size, there are fewer ways to consider. For $n = 4$, only 5 such ways are possible: parts (a), (b), (c), (e), and (h) in Figure 14.2. The number of ways is called the ***partition function***, which 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 won't pursue this line of inquiry further, however.

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

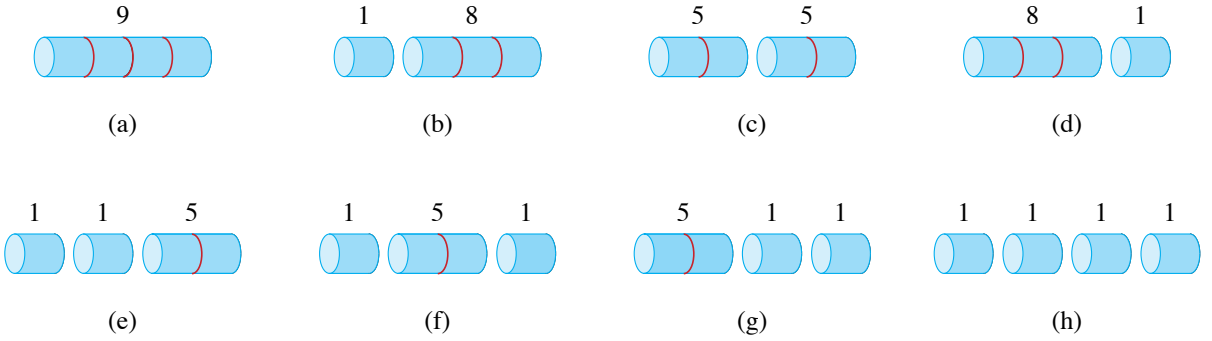


Figure 14.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 14.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue

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

For the sample problem in Figure 14.1, you can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

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

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

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\} \ . \tag{14.1}$$

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 you don't know ahead of time which value of i optimizes revenue, you have to consider all possible values for i and pick the one that maximizes revenue. You also have the option of picking no i at all if the greatest revenue comes from selling the rod uncut.

To solve the original problem of size n, you solve smaller problems of the same type. Once you make the first cut, the two resulting pieces form independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two resulting 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 you may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, let's 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. Think of every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. Then we can express the solution with no cuts at all by 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 (14.1):

$$r_n = \max\{p_i + r_{n-i} : 1 \leq i \leq n\} \ . \tag{14.2}$$

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

Recursive top-down implementation

The CUT-ROD procedure on the following page implements the computation implicit in equation (14.2) in a straightforward, top-down, recursive manner. It 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. For length $n = 0$, 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

$q = \max\{p_i + \text{CUT-ROD}(p, n-i) : 1 \le i \le n\}$. 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 (14.2).

```
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
```

If you code up CUT-ROD in your favorite programming language and run it on your computer, you'll find that once the input size becomes moderately large, your program takes a long time to run. For $n = 40$, your program may take several minutes and possibly more than an hour. For large values of n, you'll also discover that each time you increase n by 1, your program's running time approximately doubles.

Why is CUT-ROD so inefficient? The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values, which means that it solves the same subproblems repeatedly. Figure 14.3 shows a recursion tree demonstrating 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(p, n) for a particular value of 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

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \,. \tag{14.3}$$

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 14.1-1 asks you to show,

$$T(n) = 2^n \,, \tag{14.4}$$

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 possible ways of cutting up a rod of length n. How many ways

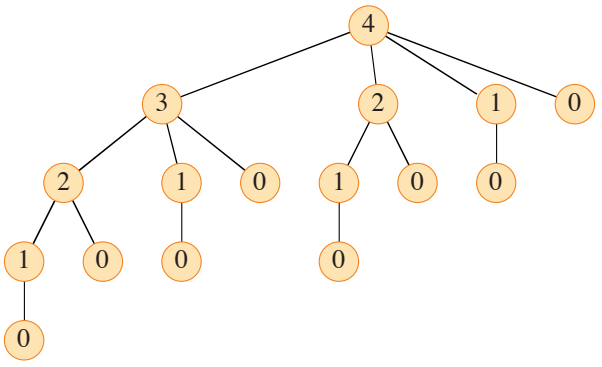


Figure 14.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and 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.

are there? A rod of length n has $n - 1$ potential locations to cut. Each possible way to cut up the rod makes a cut at some subset of these $n - 1$ locations, including the empty set, which makes for no cuts. Viewing each cut location as a distinct member of a set of $n - 1$ elements, you can see that there are 2^{n-1} subsets. Each leaf in the recursion tree of Figure 14.3 corresponds to one possible way to cut up the rod. Hence, the recursion tree has 2^{n-1} leaves. 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

Now, let's see how to use dynamic programming to convert CUT-ROD into an efficient algorithm.

The dynamic-programming method works as follows. Instead of solving the same subproblems repeatedly, as in the naive recursion solution, arrange for each subproblem to be solved *only once*. There's actually an obvious way to do so: the first time you solve a subproblem, *save its solution*. If you need to refer to this subproblem's solution again later, just look it up, rather than recomputing it.

Saving subproblem solutions comes with a cost: the additional memory needed to store solutions. Dynamic programming thus serves as an example of a ***time-memory trade-off***. The savings may be dramatic. For example, we're about to use dynamic programming to go from the exponential-time algorithm for rod cutting

down to a $\Theta(n^2)$-time algorithm. A dynamic-programming approach runs in polynomial time when the number of *distinct* subproblems involved is polynomial in the input size and you can solve each such subproblem in polynomial time.

There are usually two equivalent ways to implement a dynamic-programming approach. Solutions to the rod-cutting problem illustrate both of them.

The first approach is ***top-down*** with ***memoization***.[2] In this approach, you 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 but also saves it. 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. Solve the subproblems in size order, smallest first, storing the solution to each subproblem when it is first solved. In this way, when solving a particular subproblem, there are already saved solutions for all of the smaller subproblems its solution depends upon. You need to solve each subproblem only once, and when you first see it, you 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 lower overhead for procedure calls.

The procedures MEMOIZED-CUT-ROD and MEMOIZED-CUT-ROD-AUX on the facing page demonstrate how to memoize the top-down CUT-ROD procedure. The main procedure MEMOIZED-CUT-ROD initializes a new auxiliary array $r[0:n]$ with the value $-\infty$ which, since known revenue values are always nonnegative, is a convenient choice for denoting "unknown." MEMOIZED-CUT-ROD then calls its helper procedure, MEMOIZED-CUT-ROD-AUX, which is just the memoized version of the exponential-time 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, BOTTOM-UP-CUT-ROD on the next page, is even simpler. Using the bottom-up dynamic-programming approach, BOTTOM-UP-CUT-ROD takes advantage of the natural ordering of the subproblems: a subproblem of

---

[2] The technical term "memoization" is not a misspelling of "memorization." The word "memoization" comes from "memo," since the technique consists of recording a value to be looked up later.

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0                        // already have a solution for length n?
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n                 // i is the position of the first cut
7          q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r)}
8  r[n] = q                           // remember the solution value for length n
9  return q

BOTTOM-UP-CUT-ROD(p, n)
1  let r[0:n] be a new array          // will remember solution values in r
2  r[0] = 0
3  for j = 1 to n                     // for increasing rod length j
4      q = -∞
5      for i = 1 to j                 // i is the position of the first cut
6          q = max{q, p[i] + r[j - i]}
7      r[j] = q                       // remember the solution value for length j
8  return r[n]
```

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 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 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 BOTTOM-UP-CUT-ROD is $\Theta(n^2)$, due to its doubly nested

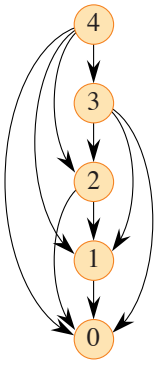


Figure 14.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y. This graph is a reduced version of the recursion tree of Figure 14.3, in which all nodes with the same label are collapsed into a single vertex and all edges go from parent to child.

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'll see aggregate analysis in detail in Section 16.1.)

Subproblem graphs

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

The ***subproblem graph*** for the problem embodies exactly this information. Figure 14.4 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. You can think of the subproblem graph as

a "reduced" or "collapsed" version of the recursion tree for the top-down recursive method, with all nodes for the same subproblem coalesced into a single vertex and all edges directed from parent to child.

The bottom-up method for dynamic programming considers the vertices of the subproblem graph in such an order that you solve the subproblems y adjacent to a given subproblem x before you solve subproblem x. (As Section B.4 notes, the adjacency relation in a directed graph is not necessarily symmetric.) Using terminology that we'll see in Section 20.4, in a bottom-up dynamic-programming algorithm, you consider the vertices of the subproblem graph in an order that is a "reverse topological sort," or a "topological sort of the transpose" 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 that we'll visit in Section 20.3, you can view the top-down method (with memoization) for dynamic programming as a "depth-first search" of the subproblem graph.

The size of the subproblem graph $G = (V, E)$ can help you determine the running time of the dynamic-programming algorithm. Since you 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

The procedures MEMOIZED-CUT-ROD and BOTTOM-UP-CUT-ROD return the *value* of an optimal solution to the rod-cutting problem, but they do not return the solution *itself*: a list of piece sizes.

Let's see how to 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, you can readily print an optimal solution. The procedure EXTENDED-BOTTOM-UP-CUT-ROD on the next page 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. It's 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 procedure PRINT-CUT-ROD-SOLUTION on the following page takes as input an array $p[1:n]$ of prices and a rod size n. It calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the array $s[1:n]$ of optimal first-piece sizes. Then it prints out the complete list of piece sizes in an optimal decomposition of a

rod of length n. For the sample price chart appearing in Figure 14.1, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ returns 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
$s[i]$		1	2	3	2	2	6	1	2	3	10

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

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0:n] and s[1:n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n                  // for increasing rod length j
 4      q = -∞
 5      for i = 1 to j              // i is the position of the first cut
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i            // best cut location so far for length j
 9      r[j] = q                    // remember the solution value for length j
10  return r and s

PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]          // cut location for length n
4      n = n - s[n]        // length of the remainder of the rod
```

Exercises

14.1-1

Show that equation (14.4) follows from equation (14.3) and the initial condition $T(0) = 1$.

14.1-2

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 $1 \le i \le n$, having maximum

density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

14.1-3
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.

14.1-4
Modify CUT-ROD and MEMOIZED-CUT-ROD-AUX so that their **for** loops go up to only $\lfloor n/2 \rfloor$, rather than up to n. What other changes to the procedures do you need to make? How are their running times affected?

14.1-5
Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution.

14.1-6
The Fibonacci numbers are defined by recurrence (3.31) on page 69. Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges does the graph contain?

14.2 Matrix-chain multiplication

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, where the matrices aren't necessarily square, the goal is to compute the product

$$A_1 A_2 \cdots A_n \; . \tag{14.5}$$

using the standard algorithm[3] for multiplying rectangular matrices, which we'll see in a moment, while minimizing the number of scalar multiplications.

You can evaluate the expression (14.5) using the algorithm for multiplying pairs of rectangular matrices as a subroutine once you 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

[3] None of the three methods from Sections 4.1 and Section 4.2 can be used directly, because they apply only to square matrices.

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 you can fully parenthesize the product $A_1 A_2 A_3 A_4$ in five distinct ways:

$(A_1(A_2(A_3 A_4)))$,
$(A_1((A_2 A_3)A_4))$,
$((A_1 A_2)(A_3 A_4))$,
$((A_1(A_2 A_3))A_4)$,
$(((A_1 A_2)A_3)A_4)$.

How you parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two rectangular matrices. The standard algorithm is given by the procedure RECTANGULAR-MATRIX-MULTIPLY, which generalizes the square-matrix multiplication procedure MATRIX-MULTIPLY on page 81. The RECTANGULAR-MATRIX-MULTIPLY procedure computes $C = C + A \cdot B$ for three matrices $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$, where A is $p \times q$, B is $q \times r$, and C is $p \times r$.

```
RECTANGULAR-MATRIX-MULTIPLY(A, B, C, p, q, r)
1  for i = 1 to p
2      for j = 1 to r
3          for k = 1 to q
4              c_ij = c_ij + a_ik · b_kj
```

The running time of RECTANGULAR-MATRIX-MULTIPLY is dominated by the number of scalar multiplications in line 4, which is pqr. Therefore, we'll consider the cost of multiplying matrices to be the number of scalar multiplications. (The number of scalar multiplications dominates even if we consider initializing $C = 0$ to perform just $C = A \cdot B$.)

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. Multiplying according to the parenthesization $((A_1 A_2)A_3)$ performs $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. Multiplying according to the alternative parenthesization $(A_1(A_2 A_3))$ performs $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product $A_2 A_3$, 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. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

The matrix-chain multiplication problem does not entail actually multiplying matrices. The 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 is not an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, the sequence consists of just one matrix, and therefore there is 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

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \displaystyle\sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 . \end{cases} \tag{14.6}$$

Problem 12-4 on page 329 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 14.2-3) is to show that the solution to the recurrence (14.6) 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

Let's use the dynamic-programming method to determine how to optimally parenthesize a matrix chain, by following 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'll go through these steps in order, demonstrating how to apply each step to the problem.

Step 1: The structure of an optimal parenthesization

In the first step of the dynamic-programming method, you find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. To perform this step for the matrix-chain multiplication problem, it's convenient to first introduce some notation. Let $A_{i:j}$, where $i \le j$, denote the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$. If the problem is nontrivial, that is, $i < j$, then to parenthesize the product $A_i A_{i+1} \cdots A_j$, the product must split between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, 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 $A_i A_{i+1} \cdots A_j$, you split the product between A_k and A_{k+1}. Then the way you 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 you 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 is lower than the optimum: a contradiction. A similar observation holds for how to 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 let's use the optimal substructure to show how to construct an optimal solution to the problem from optimal solutions to subproblems. Any solution to a nontrivial instance of the matrix-chain multiplication problem requires splitting the product, and any optimal solution contains within it optimal solutions to subproblem instances. Thus, to build an optimal solution to an instance of the matrix-chain multiplication problem, split the problem into two subproblems (optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$), find optimal solutions to the two subproblem instances, and then combine these optimal subproblem solutions. To ensure that you've examined the optimal split, you must consider all possible splits.

Step 2: A recursive solution

The next step is to define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, a subproblem is to determine the minimum cost of parenthesizing $A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$. Given the input dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$, an index pair i, j specifies a subproblem. 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 $A_{1:n}$ is thus $m[1, n]$.

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. Suppose that an optimal parenthesization splits 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 $m[i, k]$ for computing the subproduct $A_{i:k}$, plus the minimum cost $m[k+1, j]$ for computing the subproduct, $A_{k+1:j}$, plus the cost of multiplying these two matrices together. Because each matrix A_i is $p_{i-1} \times p_i$, 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 you know the value of k. But you don't, at least not yet. You have to try all possible values of k. How many are there? Just $j - i$, namely $k = i, i + 1, \ldots, j - 1$. Since the optimal parenthesization must use one of these values for k, you need only check them all to find the best. Thus, the recursive definition for the minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j \ , \\ \min \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j : i \le k < j\} & \text{if } i < j \ . \end{cases} \tag{14.7}$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information you need to construct an optimal solution. To help you do so, let's define $s[i, j]$ to be a value of k at which you 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, you could write a recursive algorithm based on recurrence (14.7) to compute the minimum cost $m[1, n]$ for multiplying $A_1 A_2 \cdots A_n$. But as we saw

for the rod-cutting problem, and as we shall see in Section 14.3, this recursive algorithm takes exponential time. That's no better than the brute-force method of checking each way of parenthesizing the product.

Fortunately, there aren't all that many distinct subproblems: just 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.[4] 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 (14.7) recursively, let's compute the optimal cost by using a tabular, bottom-up approach, as in the procedure MATRIX-CHAIN-ORDER. (The corresponding top-down approach using memoization appears in Section 14.3.) The input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$ of matrix dimensions, along with n, so that for $i = 1, 2, \ldots, n$, matrix A_i has dimensions $p_{i-1} \times p_i$. The procedure uses an auxiliary table $m[1:n, 1:n]$ to store the $m[i, j]$ costs and another auxiliary table $s[1:n-1, 2:n]$ that records which index k achieved the optimal cost in computing $m[i, j]$. The table s will help in constructing an optimal solution.

```
MATRIX-CHAIN-ORDER(p, n)
 1  let m[1:n, 1:n] and s[1:n − 1, 2:n] be new tables
 2  for i = 1 to n                          // chain length 1
 3      m[i, i] = 0
 4  for l = 2 to n                          // l is the chain length
 5      for i = 1 to n − l + 1              // chain begins at A_i
 6          j = i + l − 1                   // chain ends at A_j
 7          m[i, j] = ∞
 8          for k = i to j − 1              // try A_{i:k} A_{k+1:j}
 9              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] = q             // remember this cost
12                  s[i, j] = k             // remember this index
13  return m and s
```

In what order should the algorithm fill in the table entries? To answer this question, let's see which entries of the table need to be accessed when computing the

[4] The $\binom{n}{2}$ term counts all pairs in which $i < j$. Because i and j may be equal, we need to add in the n term.

cost $m[i, j]$. Equation (14.7) tells us that to compute the cost of matrix product $A_{i:j}$, first the costs of the products $A_{i:k}$ and $A_{k+1:j}$ need to have been computed for all $k = i, i+1, \ldots, j-1$. The chain $A_i A_{i+1} \cdots A_j$ consists of $j - i + 1$ matrices, and the chains $A_i A_{i+1} \ldots A_k$ and $A_{k+1} A_{k+2} \ldots A_j$ consist of $k - i + 1$ and $j - k$ matrices, respectively. Since $k < j$, a chain of $k - i + 1$ matrices consists of fewer than $j - i + 1$ matrices. Likewise, since $k \geq i$, a chain of $j - k$ matrices consists of fewer than $j - i + 1$ matrices. Thus, the algorithm should fill in the table m from shorter matrix chains to longer matrix chains. That is, for the subproblem of optimally parenthesizing the chain $A_i A_{i+1} \cdots A_j$, it makes sense to consider the subproblem size as the length $j - i + 1$ of the chain.

Now, let's see how the MATRIX-CHAIN-ORDER procedure fills in the $m[i, j]$ entries in order of increasing chain length. Lines 2–3 initialize $m[i, i] = 0$ for $i = 1, 2, \ldots, n$, since any matrix chain with just one matrix requires no scalar multiplications. In the **for** loop of lines 4–12, the loop variable l denotes the length of matrix chains whose minimum costs are being computed. Each iteration of this loop uses recurrence (14.7) to compute $m[i, i+l-1]$ for $i = 1, 2, \ldots, n-l+1$. In the first iteration, $l = 2$, and so the loop computes $m[i, i+1]$ for $i = 1, 2, \ldots, n-1$: the minimum costs for chains of length $l = 2$. 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 on, ending with a single matrix chain of length $l = n$ and computing $m[1, n]$. When lines 7–12 compute an $m[i, j]$ cost, this cost depends only on table entries $m[i, k]$ and $m[k+1, j]$, which have already been computed.

Figure 14.5 illustrates the m and s tables, as filled in by the MATRIX-CHAIN-ORDER procedure on a chain of $n = 6$ matrices. Since $m[i, j]$ is defined only for $i \leq j$, only the portion of the table m on or 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, the minimum cost $m[i, j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices appears at the intersection of lines running northeast from A_i and northwest from A_j. Reading across, each diagonal 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 14.2-5 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.

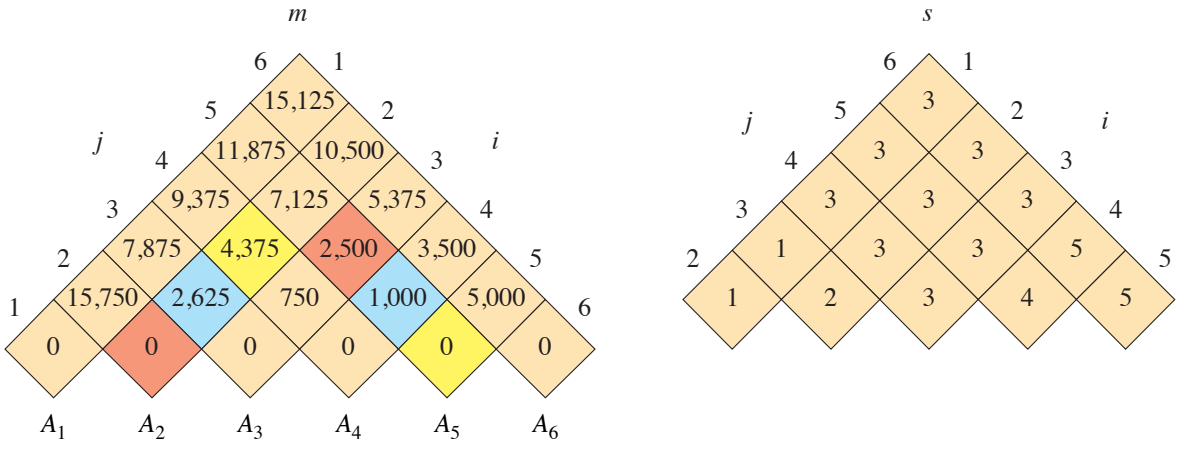


Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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 entries that are not tan, the pairs that have the same color are taken together in line 9 when computing

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

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 $s[1 : n-1, 2 : n]$ provides the information needed 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}. The final matrix multiplication in computing $A_{1:n}$ optimally is $A_{1:s[1,n]} A_{s[1,n]+1:n}$. The s table contains the information needed to determine the earlier matrix multiplications as well, using recursion: $s[1, s[1, n]]$ 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 recursive procedure PRINT-OPTIMAL-PARENS on the facing page prints an optimal parenthesization of the matrix chain product $A_i A_{i+1} \cdots A_j$, given the s table computed by MATRIX-CHAIN-ORDER and the in-

dices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of the full matrix chain product $A_1 A_2 \cdots A_n$. In the example of Figure 14.5, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the optimal parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

Exercises

14.2-1

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

14.2-2

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 is MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.) Assume that the call RECTANGULAR-MATRIX-MULTIPLY(A, B) returns the product of matrices A and B.

14.2-3

Use the substitution method to show that the solution to the recurrence (14.6) is $\Omega(2^n)$.

14.2-4

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?

14.2-5

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.4) on page 1141 useful.)

14.2-6
Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

14.3 Elements of dynamic programming

Although you have just seen two complete examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should you look for a dynamic-programming solution to a problem? In this section, we'll examine the two key ingredients that an optimization problem must have in order for dynamic programming to apply: optimal substructure and overlapping subproblems. We'll also revisit and discuss more fully how memoization might help you take advantage of the overlapping-subproblems property in a top-down recursive approach.

Optimal substructure

The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. Recall that a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. When a problem exhibits optimal substructure, that gives you a good clue that dynamic programming might apply. (As Chapter 15 discusses, it also might mean that a greedy strategy applies, however.) Dynamic programming builds an optimal solution to the problem from optimal solutions to subproblems. Consequently, you must take care to ensure that the range of subproblems you consider includes those used in an optimal solution.

Optimal substructure was key to solving both of the previous problems in this chapter. In Section 14.1, we observed that the optimal way of cutting up a rod of length n (if Serling Enterprises makes any cuts at all) involves optimally cutting up the two pieces resulting from the first cut. In Section 14.2, we noted that an optimal parenthesization of the matrix chain product $A_i A_{i+1} \cdots A_j$ that splits the product between A_k and A_{k+1} contains within it optimal solutions to the problems of parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of making a choice, such as choosing an initial cut in a rod or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by "cutting out" the nonoptimal solution to each subproblem and "pasting in" the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If an optimal solution gives rise to more than one subproblem, they are typically so similar that you can modify the cut-and-paste argument for one to apply to the others with little effort.

To characterize the space of subproblems, a good rule of thumb says to try to keep the space as simple as possible and then expand it as necessary. For example, the space of subproblems for the rod-cutting problem contained the problems of optimally cutting up a rod of length i for each size i. This subproblem space worked well, and it was not necessary to try a more general space of subproblems.

Conversely, suppose that you tried to constrain the subproblem space for matrix-chain multiplication to matrix products of the form $A_1 A_2 \cdots A_j$. As before, an optimal parenthesization must split this product between A_k and A_{k+1} for some $1 \le k < j$. Unless you can guarantee that k always equals $j-1$, you will find that you have subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. Moreover, the latter subproblem does not have the form $A_1 A_2 \cdots A_j$. To solve this problem by dynamic programming, you need to allow the subproblems to vary at "both ends." That is, both i and j need to vary in the subproblem of parenthesizing the product $A_i A_{i+1} \cdots A_j$.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems an optimal solution to the original problem uses, and
2. how many choices you have in determining which subproblem(s) to use in an optimal solution.

In the rod-cutting problem, an optimal solution for cutting up a rod of size n uses just one subproblem (of size $n-i$), but we have to consider n choices for i in order to determine which one yields an optimal solution. Matrix-chain multiplication for the subchain $A_i A_{i+1} \cdots A_j$ serves an example with two subproblems and $j-i$ choices. For a given matrix A_k where the product splits, two subproblems arise—parenthesizing $A_i A_{i+1} \cdots A_k$ and parenthesizing $A_{k+1} A_{k+2} \cdots A_j$—and we have to solve *both* of them optimally. Once we determine the optimal solutions to subproblems, we choose from among $j-i$ candidates for the index k.

Informally, the running time of a dynamic-programming algorithm depends on the product of two factors: the number of subproblems overall and how many choices you look at for each subproblem. In rod cutting, we had $\Theta(n)$ subproblems overall, and at most n choices to examine for each, yielding an $O(n^2)$ running time. Matrix-chain multiplication had $\Theta(n^2)$ subproblems overall, and each had at most $n-1$ choices, giving an $O(n^3)$ running time (actually, a $\Theta(n^3)$ running time, by Exercise 14.2-5).

Usually, the subproblem graph gives an alternative way to perform the same analysis. Each vertex corresponds to a subproblem, and the choices for a subproblem are the edges incident from that subproblem. Recall that in rod cutting, the subproblem graph has n vertices and at most n edges per vertex, yielding an $O(n^2)$ running time. For matrix-chain multiplication, if you were to draw the subproblem graph, it would have $\Theta(n^2)$ vertices and each vertex would have degree at most $n-1$, giving a total of $O(n^3)$ vertices and edges.

Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, you first find optimal solutions to subproblems and, having solved the subproblems, you find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among subproblems as to which you will use in solving the problem. The cost of the problem solution is usually the subproblem costs plus a cost that is directly attributable to the choice itself. In rod cutting, for example, first we solved the subproblems of determining optimal ways to cut up rods of length i for $i = 0, 1, \ldots, n-1$, and then we determined which of these subproblems yielded an optimal solution for a rod of length n, using equation (14.2). The cost attributable to the choice itself is the term p_i in equation (14.2). In matrix-chain multiplication, we determined optimal parenthesizations of subchains of $A_i A_{i+1} \cdots A_j$, and then we chose the matrix A_k at which to split the product. The cost attributable to the choice itself is the term $p_{i-1} p_k p_j$.

Chapter 15 explores "greedy algorithms," which have many similarities to dynamic programming. In particular, problems to which greedy algorithms apply have optimal substructure. One major difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to subproblems and then making an informed choice, greedy algorithms first make a "greedy" choice—the choice that looks best at the time—and then solve a resulting subprob-

lem, without bothering to solve all possible related smaller subproblems. Surprisingly, in some cases this strategy works!

Subtleties

You should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems whose input consists of a directed graph $G = (V, E)$ and vertices $u, v \in V$.

Unweighted shortest path:[5] Find a path from u to v consisting of the fewest edges. Such a path must be simple, since removing a cycle from a path produces a path with fewer edges.

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. (Without the requirement that the path must be simple, the problem is undefined, since repeatedly traversing a cycle creates paths with an arbitrarily large number of edges.)

The unweighted shortest-path problem exhibits optimal substructure. Here's how. Suppose that $u \neq v$, so that the problem is nontrivial. Then, any path p from u to v must contain an intermediate vertex, say w. (Note that w may be u or v.) Then, we can decompose the path $u \overset{p}{\rightsquigarrow} v$ into subpaths $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$. The number of edges in p equals the number of edges in p_1 plus the number of edges in p_2. We claim that if p is an optimal (i.e., shortest) path from u to v, then p_1 must be a shortest path from u to w. Why? As suggested earlier, use a "cut-and-paste" argument: if there were another path, say p_1', from u to w with fewer edges than p_1, then we could cut out p_1 and paste in p_1' to produce a path $u \overset{p_1'}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$ with fewer edges than p, thus contradicting p's optimality. Likewise, p_2 must be a shortest path from w to v. Thus, to find a shortest path from u to v, consider all intermediate vertices w, find a shortest path from u to w and a shortest path from w to v, and choose an intermediate vertex w that yields the overall shortest path. Section 23.2 uses a variant of this observation of optimal substructure to find a shortest path between every pair of vertices on a weighted, directed graph.

You might be tempted to assume that the problem of finding an unweighted longest simple path exhibits optimal substructure as well. After all, if we decompose a longest simple path $u \overset{p}{\rightsquigarrow} v$ into subpaths $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, then mustn't p_1 be a longest simple path from u to w, and mustn't p_2 be a longest simple path from w to v? The answer is no! Figure 14.6 supplies an example. Consider the

---

[5] We use the term "unweighted" to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 22 and 23. You can use the breadth-first search technique of Chapter 20 to solve the unweighted problem.

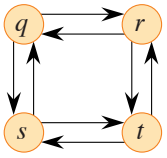


Figure 14.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

path $q \to r \to t$, which is a longest simple path from q to t. Is $q \to r$ a longest simple path from q to r? No, for the path $q \to s \to t \to r$ is a simple path that is longer. Is $r \to t$ a longest simple path from r to t? No again, for the path $r \to q \to s \to t$ is a simple path that is longer.

This example shows that for longest simple paths, not only does the problem lack optimal substructure, but you cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems. If you combine the longest simple paths $q \to s \to t \to r$ and $r \to q \to s \to t$, you get the path $q \to s \to t \to r \to q \to s \to t$, which is not simple. Indeed, the problem of finding an unweighted longest simple path does not appear to have any sort of optimal substructure. No efficient dynamic-programming algorithm for this problem has ever been found. In fact, this problem is NP-complete, which—as we shall see in Chapter 34—means that we are unlikely to find a way to solve it in polynomial time.

Why is the substructure of a longest simple path so different from that of a shortest path? Although a solution to a problem for both longest and shortest paths uses two subproblems, the subproblems in finding the longest simple path are not ***independent***, whereas for shortest paths they are. What do we mean by subproblems being independent? We mean that the solution to one subproblem does not affect the solution to another subproblem of the same problem. For the example of Figure 14.6, we have the problem of finding a longest simple path from q to t with two subproblems: finding longest simple paths from q to r and from r to t. For the first of these subproblems, we chose the path $q \to s \to t \to r$, which used the vertices s and t. These vertices cannot appear in a solution to the second subproblem, since the combination of the two solutions to subproblems yields a path that is not simple. If vertex t cannot be in the solution to the second problem, then there is no way to solve it, since t is required to be on the path that forms the solution, and it is not the vertex where the subproblem solutions are "spliced" together (that vertex being r). Because vertices s and t appear in one subproblem solution, they cannot appear in the other subproblem solution. One of them must be in the solution to the other subproblem, however, and an optimal solution requires both.

Thus, we say that these subproblems are not independent. Looked at another way, using resources in solving one subproblem (those resources being vertices) renders them unavailable for the other subproblem.

Why, then, are the subproblems independent for finding a shortest path? The answer is that by nature, the subproblems do not share resources. We claim that if a vertex w is on a shortest path p from u to v, then we can splice together *any* shortest path $u \overset{p_1}{\rightsquigarrow} w$ and *any* shortest path $w \overset{p_2}{\rightsquigarrow} v$ to produce a shortest path from u to v. We are assured that, other than w, no vertex can appear in both paths p_1 and p_2. Why? Suppose that some vertex $x \neq w$ appears in both p_1 and p_2, so that we can decompose p_1 as $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ and p_2 as $w \rightsquigarrow x \overset{p_{xv}}{\rightsquigarrow} v$. By the optimal substructure of this problem, path p has as many edges as p_1 and p_2 together. Let's say that p has e edges. Now let us construct a path $p' = u \overset{p_{ux}}{\rightsquigarrow} x \overset{p_{xv}}{\rightsquigarrow} v$ from u to v. Because we have excised the paths from x to w and from w to x, each of which contains at least one edge, path p' contains at most $e - 2$ edges, which contradicts the assumption that p is a shortest path. Thus, we are assured that the subproblems for the shortest-path problem are independent.

The two problems examined in Sections 14.1 and 14.2 have independent subproblems. In matrix-chain multiplication, the subproblems are multiplying subchains $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$. These subchains are disjoint, so that no matrix could possibly be included in both of them. In rod cutting, to determine the best way to cut up a rod of length n, we looked at the best ways of cutting up rods of length i for $i = 0, 1, \ldots, n-1$. Because an optimal solution to the length-n problem includes just one of these subproblem solutions (after cutting off the first piece), independence of subproblems is not an issue.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be "small" in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems***.[6] In contrast, a problem for which a divide-and-

[6] It may seem strange that dynamic programming relies on subproblems being both independent and overlapping. Although these requirements may sound contradictory, they describe two different notions, rather than two points on the same axis. Two subproblems of the same problem are independent if they do not share resources. Two subproblems are overlapping if they are really the same subproblem that occurs as a subproblem of different problems.

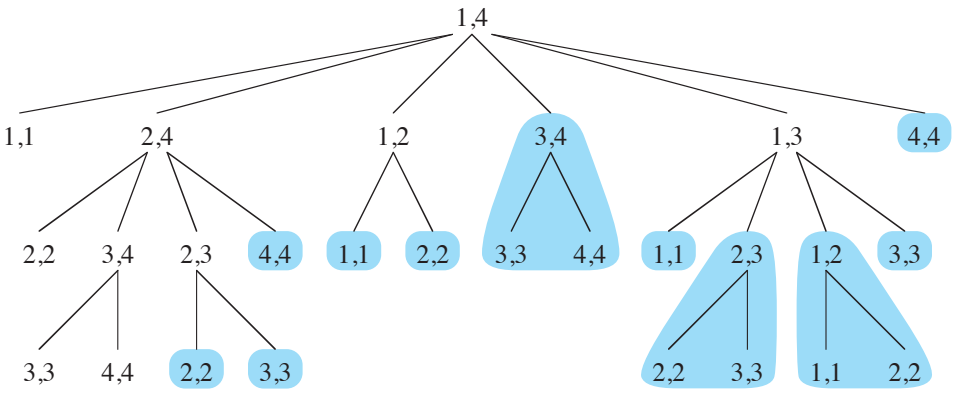


Figure 14.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a subtree shaded blue are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

In Section 14.1, we briefly examined how a recursive solution to rod cutting makes exponentially many calls to find solutions of smaller subproblems. The dynamic-programming solution reduces the running time from the exponential time of the recursive algorithm down to quadratic time.

To illustrate the overlapping-subproblems property in greater detail, let's revisit the matrix-chain multiplication problem. Referring back to Figure 14.5, observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, it references entry $m[3, 4]$ four times: during the computations of $m[2, 4]$, $m[1, 4]$, $m[3, 5]$, and $m[3, 6]$. If the algorithm were to recompute $m[3, 4]$ each time, rather than just looking it up, the running time would increase dramatically. To see how, consider the inefficient recursive procedure RECURSIVE-MATRIX-CHAIN on the facing page, which determines $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix-chain product $A_{i:j} = A_i A_{i+1} \cdots A_j$. The procedure is based directly on the recurrence (14.7). Figure 14.7 shows the recursion tree produced by the call RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node is labeled by the values of the parameters i and j. Observe that some pairs of values occur many times.

In fact, the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. To see why, let $T(n)$ denote the time taken by RECURSIVE-MATRIX-

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

CHAIN to compute an optimal parenthesization of a chain of n matrices. Because the execution of lines 1–2 and of lines 6–7 each take at least unit time, as does the multiplication in line 5, inspection of the procedure yields the recurrence

$$T(n) \geq \begin{cases} 1 & \text{if } n = 1 , \\ 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) & \text{if } n > 1 . \end{cases}$$

Noting that for $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n . \tag{14.8}$$

Let's prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we'll show that $T(n) \geq 2^{n-1}$ for all $n \geq 1$. For the base case $n = 1$, the summation is empty, and we get $T(1) \geq 1 = 2^0$. Inductively, for $n \geq 2$ we have

$$\begin{aligned} T(n) &\geq 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{j=0}^{n-2} 2^j + n && \text{(letting } j = i - 1) \\ &= 2(2^{n-1} - 1) + n && \text{(by equation (A.6) on page 1142)} \\ &= 2^n - 2 + n \\ &\geq 2^{n-1} , \end{aligned}$$

which completes the proof. Thus, the total amount of work performed by the call RECURSIVE-MATRIX-CHAIN$(p, 1, n)$ is at least exponential in n.

Compare this top-down, recursive algorithm (without memoization) with the bottom-up dynamic-programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property. Matrix-chain multiplication has only $\Theta(n^2)$ distinct subproblems, and the dynamic-programming algorithm solves each exactly once. The recursive algorithm, on the other hand, must solve each subproblem every time it reappears in the recursion tree. Whenever a recursion tree for the natural recursive solution to a problem contains the same subproblem repeatedly, and the total number of distinct subproblems is small, dynamic programming can improve efficiency, sometimes dramatically.

Reconstructing an optimal solution

As a practical matter, you'll often want to store in a separate table which choice you made in each subproblem so that you do not have to reconstruct this information from the table of costs.

For matrix-chain multiplication, the table $s[i, j]$ saves a significant amount of work when we need to reconstruct an optimal solution. Suppose that the MATRIX-CHAIN-ORDER procedure on page 378 did not maintain the $s[i, j]$ table, so that it filled in only the table $m[i, j]$ containing optimal subproblem costs. The procedure chooses from among $j - i$ possibilities when determining which subproblems to use in an optimal solution to parenthesizing $A_i A_{i+1} \cdots A_j$, and $j - i$ is not a constant. Therefore, it would take $\Theta(j - i) = \omega(1)$ time to reconstruct which subproblems it chose for a solution to a given problem. Because MATRIX-CHAIN-ORDER stores in $s[i, j]$ the index of the matrix at which it split the product $A_i A_{i+1} \cdots A_j$, the PRINT-OPTIMAL-PARENS procedure on page 381 can look up each choice in $O(1)$ time.

Memoization

As we saw for the rod-cutting problem, there is an alternative approach to dynamic programming that often offers the efficiency of the bottom-up dynamic-programming approach while maintaining a top-down strategy. The idea is to ***memoize*** the natural, but inefficient, recursive algorithm. As in the bottom-up approach, you maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table.

Each subsequent encounter of this subproblem simply looks up the value stored in the table and returns it.[7]

The procedure MEMOIZED-MATRIX-CHAIN is a memoized version of the procedure RECURSIVE-MATRIX-CHAIN on page 389. Note where it resembles the memoized top-down method on page 369 for the rod-cutting problem.

```
MEMOIZED-MATRIX-CHAIN(p, n)
1  let m[1:n, 1:n] be a new table
2  for i = 1 to n
3      for j = i to n
4          m[i, j] = ∞
5  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
                + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

The MEMOIZED-MATRIX-CHAIN procedure, like the bottom-up MATRIX-CHAIN-ORDER procedure on page 378, maintains a table $m[1:n, 1:n]$ of computed values of $m[i, j]$, the minimum number of scalar multiplications needed to compute the matrix $A_{i:j}$. Each table entry initially contains the value ∞ to indicate that the entry has yet to be filled in. Upon calling LOOKUP-CHAIN(m, p, i, j), if line 1 finds that $m[i, j] < \infty$, then the procedure simply returns the previously computed cost $m[i, j]$ in line 2. Otherwise, the cost is computed as in RECURSIVE-MATRIX-CHAIN, stored in $m[i, j]$, and returned. Thus, LOOKUP-CHAIN(m, p, i, j) always returns the value of $m[i, j]$, but it computes it only upon the first call of LOOKUP-CHAIN with these specific values of i and j.

[7] This approach presupposes that you know the set of all possible subproblem parameters and that you have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys.

Figure 14.7 illustrates how MEMOIZED-MATRIX-CHAIN saves time compared with RECURSIVE-MATRIX-CHAIN. Subtrees shaded blue represent values that are looked up rather than recomputed.

Like the bottom-up procedure MATRIX-CHAIN-ORDER, the memoized procedure MEMOIZED-MATRIX-CHAIN runs in $O(n^3)$ time. To begin with, line 4 of MEMOIZED-MATRIX-CHAIN executes $\Theta(n^2)$ times, which dominates the running time outside of the call to LOOKUP-CHAIN in line 5. We can categorize the calls of LOOKUP-CHAIN into two types:

1. calls in which $m[i, j] = \infty$, so that lines 3–9 execute, and
2. calls in which $m[i, j] < \infty$, so that LOOKUP-CHAIN simply returns in line 2.

There are $\Theta(n^2)$ calls of the first type, one per table entry. All calls of the second type are made as recursive calls by calls of the first type. Whenever a given call of LOOKUP-CHAIN makes recursive calls, it makes $O(n)$ of them. Therefore, there are $O(n^3)$ calls of the second type in all. Each call of the second type takes $O(1)$ time, and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls. The total time, therefore, is $O(n^3)$. Memoization thus turns an $\Omega(2^n)$-time algorithm into an $O(n^3)$-time algorithm.

We have seen how to solve the matrix-chain multiplication problem by either a top-down, memoized dynamic-programming algorithm or a bottom-up dynamic-programming algorithm in $O(n^3)$ time. Both the bottom-up and memoized methods take advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ distinct subproblems in total, and either of these methods computes the solution to each subproblem only once. Without memoization, the natural recursive algorithm runs in exponential time, since solved subproblems are repeatedly solved.

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table. Moreover, for some problems you can exploit the regular pattern of table accesses in the dynamic-programming algorithm to reduce time or space requirements even further. On the other hand, in certain situations, some of the subproblems in the subproblem space might not need to be solved at all. In that case, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercises

14.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesiz-

ing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

14.3-2
Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

14.3-3
Consider the antithetical variant of the matrix-chain multiplication problem where the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

14.3-4
As stated, in dynamic programming, you first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that she does not always need to solve all the subproblems in order to find an optimal solution. She suggests that she 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.

14.3-5
Suppose that the rod-cutting problem of Section 14.1 also had a limit l_i on the number of pieces of length i allowed to be produced, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 14.1 no longer holds.

14.4 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, cytosine, guanine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the 4-element set $\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. (See Section C.1 for the definition of a string.) For example, the DNA of one organism may be $S_1 = \texttt{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$, and the DNA of another organism may be $S_2 = \texttt{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$. One reason to com-

pare 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 14-5 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 0 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 sequences 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***, the input is two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and the goal is 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

You can solve the LCS problem with a brute-force approach: 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 you 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'll 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 14.1 (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 $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then 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 14.1 characterizes longest common subsequences says 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'll see in a moment.

Step 2: A recursive solution

Theorem 14.1 implies that you 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 $x_m = y_n$, you need to 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 you have to 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, one of the optimal subproblem solutions must appear within an LCS of X and Y.

The LCS problem has the overlapping-subproblems property. Here's how. To find an LCS of X and Y, you might 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, solving the LCS problem recursively involves establishing a recurrence for the value of an optimal solution. Let's 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

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \tag{14.9}$$

In this recursive formulation, a condition in the problem restricts which subproblems to consider. When $x_i = y_j$, you can and should consider the subproblem of finding an LCS of X_{i-1} and Y_{j-1}. Otherwise, you 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 didn't rule out any 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 14-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (14.9), you could 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 (computing $c[i, j]$ for $0 \le i \le m$ and $0 \le j \le n$), dynamic programming can compute the solutions bottom up.

The procedure LCS-LENGTH on the next page 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, along with their lengths. It stores the $c[i, j]$ values in a table $c[0:m, 0:n]$, 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 $b[1:m, 1:n]$ to help in constructing 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, where $c[m, n]$ contains the length of an LCS of X and Y. Figure 14.8 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.

```
LCS-LENGTH(X, Y, m, n)
 1  let b[1:m, 1:n] and c[0:m, 0:n] be new tables
 2  for i = 1 to m
 3      c[i, 0] = 0
 4  for j = 0 to n
 5      c[0, j] = 0
 6  for i = 1 to m              // compute table entries in row-major order
 7      for j = 1 to n
 8          if x_i == y_j
 9              c[i, j] = c[i − 1, j − 1] + 1
10              b[i, j] = "↖"
11          elseif c[i − 1, j] ≥ c[i, j − 1]
12              c[i, j] = c[i − 1, j]
13              b[i, j] = "↑"
14          else c[i, j] = c[i, j − 1]
15              b[i, j] = "←"
16  return c and b

PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return                  // the LCS has length 0
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i               // same as y_j
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Step 4: Constructing an LCS

With the b table returned by LCS-LENGTH, you can quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Begin at $b[m, n]$ and trace through the table by following the arrows. Each "↖" encountered in an entry $b[i, j]$ implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found. This method gives you the elements of this LCS in reverse order. The recursive procedure PRINT-LCS prints out an LCS of X and Y in the proper, forward order.

i		j	0	1	2	3	4	5	6
			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 14.8 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, as shown by the sequence shaded blue. Each "↖" on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

The initial call is PRINT-LCS(b, X, m, n). For the b table in Figure 14.8, this procedure prints $BCBA$. The procedure takes $O(m + n)$ time, 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, you 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 $c[i, j]$, you can determine in $O(1)$ time which of these three values was used to compute $c[i, j]$, without inspecting table b. Thus, you can reconstruct an LCS in $O(m+n)$ time using a procedure similar to PRINT-LCS. (Exercise 14.4-2 asks you to give the pseudocode.) Although this method saves $\Theta(mn)$ space, the auxiliary space requirement for computing

an LCS does not asymptotically decrease, since the c table takes $\Theta(mn)$ space anyway.

You 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 14.4-4 asks you to show, you can use only slightly more than the space for one row of c to compute the length of an LCS.) This improvement works if you need only the length of an LCS. If you need to reconstruct the elements of an LCS, the smaller table does not keep enough information to retrace the algorithm's steps in $O(m + n)$ time.

Exercises

14.4-1

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

14.4-2

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.

14.4-3

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

14.4-4

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.

14.4-5

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

★ *14.4-6*

Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (*Hint:* 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.)

14.5 Optimal binary search trees

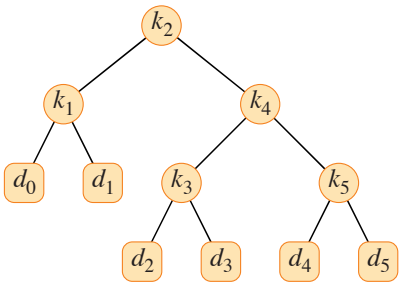
Suppose that you are designing a program to translate text from English to Latvian. For each occurrence of each English word in the text, you need to look up its Latvian equivalent. You can perform these lookup operations by building a binary search tree with n English words as keys and their Latvian equivalents as satellite data. Because you will search the tree for each individual word in the text, you want the total time spent searching to be as low as possible. You can 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* can end up appearing far from the root while a rarely used word such as *naumachia* 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 1 plus the depth of the node containing the key. You want words that occur frequently in the text to be placed nearer the root.[8] Moreover, some words in the text might have no Latvian translation,[9] and such words would not appear in the binary search tree at all. How can you organize a binary search tree so as to minimize the number of nodes visited in all searches, given that you know how often each word occurs?

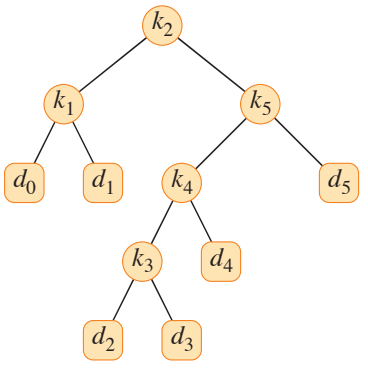
What you need is an ***optimal binary search tree***. Formally, given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys such that $k_1 < k_2 < \cdots < k_n$, build a binary search tree containing them. For each key k_i, you are given the probability p_i that any given search is for key k_i. Since some searches may be for values not in K, you also have $n + 1$ "dummy" keys $d_0, d_1, d_2, \ldots, d_n$ representing those values. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, you have the probability q_i that a search corresponds to d_i. Figure 14.9 shows two binary search trees for a set of $n = 5$ keys. Each key k_i is an internal node, and each dummy key d_i is a leaf. Since every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \,. \tag{14.10}$$

[8] If the subject of the text is ancient Rome, you might want *naumachia* to appear near the root.

[9] Yes, *naumachia* has a Latvian counterpart: *nomačija*.





node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

(a)

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

(b)

Figure 14.9 Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.

Knowing the probabilities of searches for each key and each dummy key allows us to determine the expected cost of a search in a given binary search tree T. Let us assume that the actual cost of a search equals the number of nodes examined, which is the depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is

$$\begin{aligned} \mathrm{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i \ , \end{aligned} \tag{14.11}$$

where depth_T denotes a node's depth in the tree T. The last equation follows from equation (14.10). Figure 14.9 shows how to calculate the expected search cost node by node.

For a given set of probabilities, your goal is to construct a binary search tree whose expected search cost is smallest. We call such a tree an ***optimal binary search tree***. Figure 14.9(a) shows one binary search tree, with expected cost 2.80, for the probabilities given in the figure caption. Part (b) of the figure displays an optimal binary search tree, with expected cost 2.75. This example demonstrates that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor does an optimal binary search tree always have the key with the greatest probability at the root. Here, key k_5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k_2. (The lowest expected cost of any binary search tree with k_5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. You can label the nodes of any n-node binary tree with the keys $k_1, k_2, \ldots, k_n$ to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4 on page 329, we saw that the number of binary trees with n nodes is $\Omega(4^n/n^{3/2})$. Thus you would need to examine an exponential number of binary search trees to perform an exhaustive search. We'll see how to solve this problem more efficiently with dynamic programming.

Step 1: The structure of an optimal binary search tree

To characterize the optimal substructure of optimal binary search trees, we start with an observation about subtrees. Consider any subtree of a binary search tree. It must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, a subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.

Now we can state the optimal substructure: if an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. The usual cut-and-paste argument applies. If there were a subtree T'' whose expected cost is lower than that of T', then cutting T' out of T and pasting in T'' would result in a binary search tree of lower expected cost than T, thus contradicting the optimality of T.

With the optimal substructure in hand, here is how to construct an optimal solution to the problem from optimal solutions to subproblems. Given keys $k_i, \ldots, k_j$, one of these keys, say k_r ($i \le r \le j$), is the root of an optimal subtree containing these keys. The left subtree of the root k_r contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$), and the right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$). As long as you examine all candidate roots k_r,

where $i \le r \le j$, and you determine all optimal binary search trees containing $k_i, \ldots, k_{r-1}$ and those containing $k_{r+1}, \ldots, k_j$, you are guaranteed to find an optimal binary search tree.

There is one technical detail worth understanding about "empty" subtrees. Suppose that in a subtree with keys $k_i, \ldots, k_j$, you select k_i as the root. By the above argument, k_i's left subtree contains the keys $k_i, \ldots, k_{i-1}$: no keys at all. Bear in mind, however, that subtrees also contain dummy keys. We adopt the convention that a subtree containing keys $k_i, \ldots, k_{i-1}$ has no actual keys but does contain the single dummy key d_{i-1}. Symmetrically, if you select k_j as the root, then k_j's right subtree contains the keys $k_{j+1}, \ldots, k_j$. This right subtree contains no actual keys, but it does contain the dummy key d_j.

Step 2: A recursive solution

To define the value of an optimal solution recursively, the subproblem domain is finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \ge 1$, $j \le n$, and $j \ge i - 1$. (When $j = i - 1$, there is just the dummy key d_{i-1}, but no actual keys.) Let $e[i, j]$ denote the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Your goal is to compute $e[1, n]$, the expected cost of searching an optimal binary search tree for all the actual and dummy keys.

The easy case occurs when $j = i - 1$. Then the subproblem consists of just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.

When $j \ge i$, you need to select a root k_r from among $k_i, \ldots, k_j$ and then make an optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree and an optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (14.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys $k_i, \ldots, k_j$, denote this sum of probabilities as

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \; . \tag{14.12}$$

Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) \; .$$

Noting that

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j) \; ,$$

we rewrite $e[i, j]$ as

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \,. \tag{14.13}$$

The recursive equation (14.13) assumes that you know which node k_r to use as the root. Of course, you choose the root that gives the lowest expected search cost, giving the final recursive formulation:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \,, \\ \min\{e[i, r-1] + e[r+1, j] + w(i, j) : i \le r \le j\} & \text{if } i \le j \,. \end{cases} \tag{14.14}$$

The $e[i, j]$ values give the expected search costs in optimal binary search trees. To help keep track of the structure of optimal binary search trees, define $root[i, j]$, for $1 \le i \le j \le n$, to be the index r for which k_r is the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. Although we'll see how to compute the values of $root[i, j]$, the construction of an optimal binary search tree from these values is left as Exercise 14.5-1.

Step 3: Computing the expected search cost of an optimal binary search tree

At this point, you may have noticed some similarities between our characterizations of optimal binary search trees and matrix-chain multiplication. For both problem domains, the subproblems consist of contiguous index subranges. A direct, recursive implementation of equation (14.14) would be just as inefficient as a direct, recursive matrix-chain multiplication algorithm. Instead, you can store the $e[i, j]$ values in a table $e[1 : n+1, 0 : n]$. The first index needs to run to $n+1$ rather than n because in order to have a subtree containing only the dummy key d_n, you need to compute and store $e[n+1, n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0, you need to compute and store $e[1, 0]$. Only the entries $e[i, j]$ for which $j \ge i - 1$ are filled in. The table $root[i, j]$ records the root of the subtree containing keys $k_i, \ldots, k_j$ and uses only the entries for which $1 \le i \le j \le n$.

One other table makes the dynamic-programming algorithm a little faster. Instead of computing the value of $w(i, j)$ from scratch every time you compute $e[i, j]$, which would take $\Theta(j - i)$ additions, store these values in a table $w[1 : n+1, 0 : n]$. For the base case, compute $w[i, i-1] = q_{i-1}$ for $1 \le i \le n+1$. For $j \ge i$, compute

$$w[i, j] = w[i, j-1] + p_j + q_j \,. \tag{14.15}$$

Thus, you can compute the $\Theta(n^2)$ values of $w[i, j]$ in $\Theta(1)$ time each.

The OPTIMAL-BST procedure on the next page takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and it returns the tables e and $root$. From the description above and the similarity to the MATRIX-CHAIN-ORDER procedure

in Section 14.2, you should find the operation of this procedure to be fairly straightforward. The **for** loop of lines 2–4 initializes the values of $e[i, i-1]$ and $w[i, i-1]$. Then the **for** loop of lines 5–14 uses the recurrences (14.14) and (14.15) to compute $e[i, j]$ and $w[i, j]$ for all $1 \le i \le j \le n$. In the first iteration, when $l = 1$, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1, 2, \ldots, n$. The second iteration, with $l = 2$, computes $e[i, i+1]$ and $w[i, i+1]$ for $i = 1, 2, \ldots, n-1$, and so on. The innermost **for** loop, in lines 10–14, tries each candidate index r to determine which key k_r to use as the root of an optimal binary search tree containing keys $k_i, \ldots, k_j$. This **for** loop saves the current value of the index r in $root[i, j]$ whenever it finds a better key to use as the root.

```
OPTIMAL-BST(p, q, n)
 1  let e[1 : n + 1, 0 : n], w[1 : n + 1, 0 : n],
          and root[1 : n, 1 : n] be new tables
 2  for i = 1 to n + 1                    // base cases
 3      e[i, i − 1] = q_{i−1}             // equation (14.14)
 4      w[i, i − 1] = q_{i−1}
 5  for l = 1 to n
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          e[i, j] = ∞
 9          w[i, j] = w[i, j − 1] + p_j + q_j          // equation (14.15)
10          for r = i to j                             // try all possible roots r
11              t = e[i, r − 1] + e[r + 1, j] + w[i, j]  // equation (14.14)
12              if t < e[i, j]                         // new minimum?
13                  e[i, j] = t
14                  root[i, j] = r
15  return e and root
```

Figure 14.10 shows the tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by the procedure OPTIMAL-BST on the key distribution shown in Figure 14.9. As in the matrix-chain multiplication example of Figure 14.5, the tables are rotated to make the diagonals run horizontally. OPTIMAL-BST computes the rows from bottom to top and from left to right within each row.

The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER. Its running time is $O(n^3)$, since its **for** loops are nested three deep and each loop index takes on at most n values. The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within at most 1 in all directions. Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time.

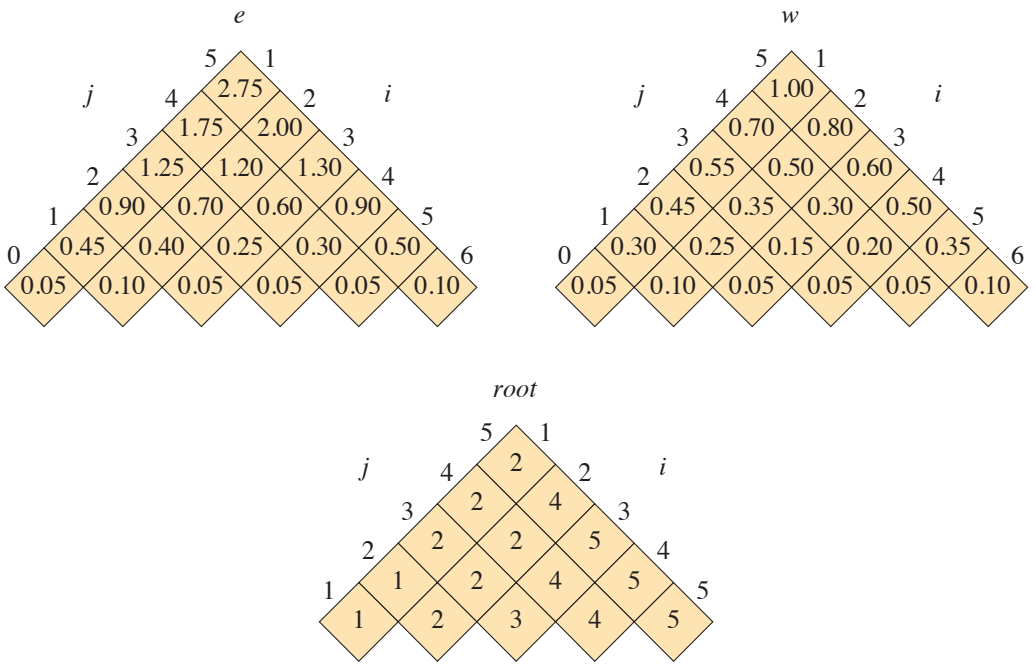


Figure 14.10 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 14.9. The tables are rotated so that the diagonals run horizontally.

Exercises

14.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST$(root, n)$ which, given the table $root[1:n, 1:n]$, outputs the structure of an optimal binary search tree. For the example in Figure 14.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 14.9(b).

14.5-2

Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

14.5-3

Suppose that instead of maintaining the table $w[i, j]$, you computed the value of $w(i, j)$ directly from equation (14.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

★ ### *14.5-4*

Knuth [264] has shown that there are always roots of optimal subtrees such that $root[i, j-1] \le root[i, j] \le root[i+1, j]$ for all $1 \le i < j \le n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Problems

14-1 Longest simple path in a directed acyclic graph

You are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. The ***weight*** of a path is the sum of the weights of the edges in the path. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What is the running time of your algorithm?

14-2 Longest palindrome subsequence

A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

14-3 Bitonic euclidean traveling-salesperson problem

In the ***euclidean traveling-salesperson problem***, you are given a set of n points in the plane, and your goal is to find the shortest closed tour that connects all n points.

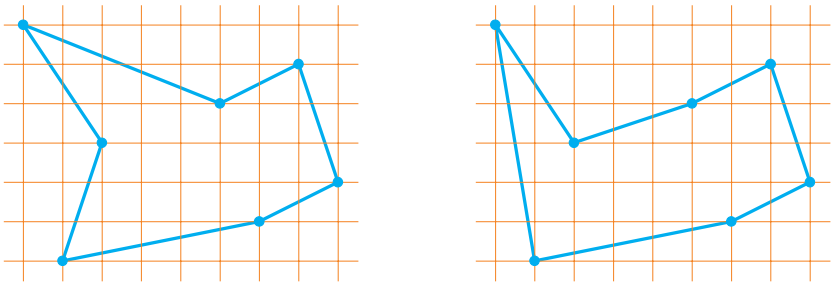


Figure 14.11 Seven points in the plane, shown on a unit grid. **(a)** The shortest closed tour, with length approximately 24.89. This tour is not bitonic. **(b)** The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

Figure 14.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).

J. L. Bentley has suggested simplifying the problem by considering only ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 14.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. (*Hint:* Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

14-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width). The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters, which are to be printed neatly on a number of lines that hold a maximum of M characters each. No word exceeds the line length, so that $l_i \leq M$ for $i = 1, 2, \ldots, n$. The criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and exactly one space appears between words, then the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. The goal is to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly. Analyze the running time and space requirements of your algorithm.

14-5 Edit distance

In order to transform a source string of text $x[1:m]$ to a target string $y[1:n]$, you can perform various transformation operations. The goal is, given x and y, to produce a series of transformations that changes x to y. An array z—assumed to be large enough to hold all the characters it needs—holds the intermediate results. Initially, z is empty, and at termination, you should have $z[j] = y[j]$ for $j = 1, 2, \ldots, n$. The procedure for solving this problem maintains current indices i into x and j into z, and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. Every character in x must be examined during the transformation, which means that at the end of the sequence of transformation operations, $i = m + 1$.

You may choose from among six transformation operations, each of which has a constant cost that depends on the operation:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$ and has cost Q_C.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$ and has cost Q_R.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$ and has cost Q_D.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x and has cost Q_I.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order: setting $z[j] = x[i + 1]$ and $z[j + 1] = x[i]$, and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i + 1]$ and has cost Q_T.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation. It has cost Q_K.

Figure 14.12 gives one way to transform the source string `algorithm` to the target string `altruistic`. Several other sequences of transformation operations can transform `algorithm` to `altruistic`.

Assume that $Q_C < Q_D + Q_I$ and $Q_R < Q_D + Q_I$, since otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming `algorithm` to `altruistic` is $3Q_C + Q_R + Q_D + 4Q_I + Q_T + Q_K$.

a. Given two sequences $x[1:m]$ and $y[1:n]$ and the costs of the transformation operations, the ***edit distance*** from x to y is the cost of the least expensive op-

Operation	x	z
initial strings	a`lgorithm`	_
copy	a`l`g`orithm`	a`_`
copy	a`lg`o`rithm`	a`l_`
replace by `t`	a`lgo`r`ithm`	a`lt_`
delete	a`lgor`i`thm`	a`lt_`
copy	a`lgori`t`hm`	a`ltr_`
insert `u`	a`lgori`t`hm`	a`ltru_`
insert `i`	a`lgori`t`hm`	a`ltrui_`
insert `s`	a`lgori`t`hm`	a`ltruis_`
twiddle	a`lgorit`h`m`	a`ltruisti_`
insert `c`	a`lgorit`h`m`	a`ltruistic_`
kill	a`lgorithm_`	a`ltruistic_`

Figure 14.12 A sequence of operations that transforms the source `algorithm` to the target string `altruistic`. The underlined characters are $x[i]$ and $z[j]$ after the operation.

eration sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1:m]$ to $y[1:n]$ and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [405, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x =$ `GATCGGCAT` and $y =$ `CAATGTGAATC`, one alignment is

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

A + under a position indicates a score of $+1$ for that position, a – indicates a score of -1, and a $*$ indicates a score of -2, so that this alignment has a total score of $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explain how to cast the problem of finding an optimal alignment as an edit-distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

14-6 Planning a company party

Professor Blutarsky is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure, that is, the supervisor relation forms a tree rooted at the president. The human resources department has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Blutarsky is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.3. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

14-7 Viterbi algorithm

Dynamic programming on a directed graph can play a part in speech recognition. A directed graph $G = (V, E)$ with labeled edges forms a formal model of a person speaking a restricted language. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. Each directed path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model, with the label of a path being the concatenation of the labels of the edges on that path.

a. Describe an efficient algorithm that, given an edge-labeled directed graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (*Hint:* You may find concepts from Chapter 20 useful.)

Now suppose that every edge $(u, v) \in E$ has an associated nonnegative probability $p(u, v)$ of being traversed, so that the corresponding sound is produced. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. Think of

the probability of a path beginning at vertex v_0 as the probability that a "random walk" beginning at v_0 follows the specified path, where the edge leaving a vertex u is taken randomly, according to the probabilities of the available edges leaving u.

b. Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at vertex v_0 and having label s. Analyze the running time of your algorithm.

14-8 Image compression by seam carving

Suppose that you are given a color picture consisting of an $m \times n$ array $A[1:m, 1:n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. You want to compress this picture slightly, by removing one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid incongruous visual effects, however, the pixels removed in two adjacent rows must lie in either the same column or adjacent columns. In this way, the pixels removed form a "seam" from the top row to the bottom row, where successive pixels in the seam are adjacent vertically or diagonally.

a. Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.

b. Suppose now that along with each pixel $A[i, j]$, you are given a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Define the disruption measure of a seam as the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

14-9 Breaking a string

A certain string-processing programming language allows you to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose that you want to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that you want to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If you program the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If you program the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the

second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, you could break first at 8 (costing 20), then break the left piece at 2 (costing another 8), and finally the right piece at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given an array $L[1:m]$ containing the break points for a string of n characters, compute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

14-10 Planning an investment strategy

Your knowledge of algorithms helps you obtain an exciting job with a hot startup, along with a $10,000 signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use your investment manager, G. I. Luvcache, to manage your signing bonus. The company that Luvcache works with requires you to observe the following rules. It offers n different investments, numbered 1 through n. In each year j, investment i provides a return rate of r_{ij}. In other words, if you invest d dollars in investment i in year j, then at the end of year j, you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investment. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$. You pay the fee once per year at the end of the year, and it is the same amount, f_2, whether you move money in and out of only one investment, or in and out of many investments.

a. The problem, as stated, allows you to invest your money in multiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)

b. Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.

c. Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?

d. Suppose that Luvcache's company imposes the additional restriction that, at any point, you can have no more than \$15,000 in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

14-11 Inventory planning

The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i, the company knows the demand d_i, that is, the number of machines that it will sell. Let $D = \sum_{i=1}^{n} d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if the company is holding any unsold machines at the end of a month, it must pay inventory costs. The company can hold up to D machines, with the cost for holding j machines given as a function $h(j)$ for $j = 1, 2, \ldots, D$ that monotonically increases with j.

Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polynomial in n and D.

14-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of $\$X$ to spend on free agents. You are allowed to spend less than $\$X$, but the owner will fire you if you spend any more than $\$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available.[10] Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

---

[10] Although there are nine positions on a baseball team, N is not necessarily equal to 9 because some general managers have particular ways of thinking about positions. For example, a general manager might consider right-handed pitchers and left-handed pitchers to be separate "positions," as well as starting pitchers, long relief pitchers (relief pitchers who can pitch several innings), and short relief pitchers (relief pitchers who normally pitch at most only one inning).

To determine how valuable a player is going to be, you decide to use a sabermetric statistic[11] known as "WAR," or "wins above replacement." A player with a higher WAR is more valuable than a player with a lower WAR. It is not necessarily more expensive to sign a player with a higher WAR than a player with a lower WAR, because factors other than a player's value determine how much it costs to sign them.

For each available free-agent player p, you have three pieces of information:

- the player's position,
- $p.cost$, the amount of money it costs to sign the player, and
- $p.war$, the player's WAR.

Devise an algorithm that maximizes the total WAR of the players you sign while spending no more than $\$X$. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total WAR of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

Chapter notes

Bellman [44] began the systematic study of dynamic programming in 1955, publishing a book about it in 1957. The word "programming," both here and in linear programming, refers to using a tabular solution method. Although optimization techniques incorporating elements of dynamic programming were known earlier, Bellman provided the area with a solid mathematical basis.

Galil and Park [172] classify dynamic-programming algorithms according to the size of the table and the number of other table entries each entry depends on. They call a dynamic-programming algorithm tD/eD if its table size is $O(n^t)$ and each entry depends on $O(n^e)$ other entries. For example, the matrix-chain multiplication algorithm in Section 14.2 is $2D/1D$, and the longest-common-subsequence algorithm in Section 14.4 is $2D/0D$.

The MATRIX-CHAIN-ORDER algorithm on page 378 is by Muraoka and Kuck [339]. Hu and Shing [230, 231] give an $O(n \lg n)$-time algorithm for the matrix-chain multiplication problem.

The $O(mn)$-time algorithm for the longest-common-subsequence problem appears to be a folk algorithm. Knuth [95] posed the question of whether subquadratic

[11] ***Sabermetrics*** is the application of statistical analysis to baseball records. It provides several ways to compare the relative values of individual players.

algorithms for the LCS problem exist. Masek and Paterson [316] answered this question in the affirmative by giving an algorithm that runs in $O(mn/\lg n)$ time, where $n \leq m$ and the sequences are drawn from a set of bounded size. For the special case in which no element appears more than once in an input sequence, Szymanski [425] shows how to solve the problem in $O((n+m)\lg(n+m))$ time. Many of these results extend to the problem of computing string edit distances (Problem 14-5).

An early paper on variable-length binary encodings by Gilbert and Moore [181], which had applications to constructing optimal binary search trees for the case in which all probabilities p_i are 0, contains an $O(n^3)$-time algorithm. Aho, Hopcroft, and Ullman [5] present the algorithm from Section 14.5. Splay trees [418], which modify the tree in response to the search queries, come within a constant factor of the optimal bounds without being initialized with the frequencies. Exercise 14.5-4 is due to Knuth [264]. Hu and Tucker [232] devised an algorithm for the case in which all probabilities p_i are 0 that uses $O(n^2)$ time and $O(n)$ space. Subsequently, Knuth [261] reduced the time to $O(n \lg n)$.

Problem 14-8 is due to Avidan and Shamir [30], who have posted on the web a wonderful video illustrating this image-compression technique.

15 Greedy Algorithms

Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step. For many optimization problems, using dynamic programming to determine the best choices is overkill, and simpler, more efficient algorithms will do. A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice leads to a globally optimal solution. This chapter explores optimization problems for which greedy algorithms provide optimal solutions. Before reading this chapter, you should read about dynamic programming in Chapter 14, particularly Section 14.3.

Greedy algorithms do not always yield optimal solutions, but for many problems they do. We first examine, in Section 15.1, a simple but nontrivial problem, the activity-selection problem, for which a greedy algorithm efficiently computes an optimal solution. We'll arrive at the greedy algorithm by first considering a dynamic-programming approach and then showing that an optimal solution can result from always making greedy choices. Section 15.2 reviews the basic elements of the greedy approach, giving a direct approach for proving greedy algorithms correct. Section 15.3 presents an important application of greedy techniques: designing data-compression (Huffman) codes. Finally, Section 15.4 shows that in order to decide which blocks to replace when a miss occurs in a cache, the "furthest-in-future" strategy is optimal if the sequence of block accesses is known in advance.

The greedy method is quite powerful and works well for a wide range of problems. Later chapters will present many algorithms that you can view as applications of the greedy method, including minimum-spanning-tree algorithms (Chapter 21), Dijkstra's algorithm for shortest paths from a single source (Section 22.3), and a greedy set-covering heuristic (Section 35.3). Minimum-spanning-tree algorithms furnish a classic example of the greedy method. Although you can read this chapter and Chapter 21 independently of each other, you might find it useful to read them together.

15.1 An activity-selection problem

Our first example is the problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Imagine that you are in charge of scheduling a conference room. You are presented with a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to reserve the conference room, and the room can serve only one activity at a time. Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \le s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$. (Assume that if your staff needs time to change over the room from one activity to the next, the changeover time is built into the intervals.) In the ***activity-selection problem***, your goal is to select a maximum-size subset of mutually compatible activities. Assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \le f_2 \le f_3 \le \cdots \le f_{n-1} \le f_n \,. \tag{15.1}$$

(We'll see later the advantage that this assumption provides.) For example, consider the set of activities in Figure 15.1. The subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities. It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger. In fact, $\{a_1, a_4, a_8, a_{11}\}$ is a largest subset of mutually compatible activities, and another largest subset is $\{a_2, a_4, a_9, a_{11}\}$.

We'll see how to solve this problem, proceeding in several steps. First we'll explore a dynamic-programming solution, in which you consider several choices when determining which subproblems to use in an optimal solution. We'll then observe that you need to consider only one choice—the greedy choice—and that when you make the greedy choice, only one subproblem remains. Based on these observations, we'll develop a recursive greedy algorithm to solve the activity-selection problem. Finally, we'll complete the process of developing a greedy solution by converting the recursive algorithm to an iterative one. Although the steps we go through in this section are slightly more involved than is typical when developing a greedy algorithm, they illustrate the relationship between greedy algorithms and dynamic programming.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Figure 15.1 A set $\{a_1, a_2, \ldots, a_{11}\}$ of activities. Activity a_i has start time s_i and finish time f_i.

The optimal substructure of the activity-selection problem

Let's verify that the activity-selection problem exhibits optimal substructure. Denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that you want to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k. By including a_k in an optimal solution, you are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts). Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.

The usual cut-and-paste argument shows that an optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If you could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then you could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. You would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

This way of characterizing optimal substructure suggests that you can solve the activity-selection problem by dynamic programming. Let's denote the size of an optimal solution for the set S_{ij} by $c[i, j]$. Then, the dynamic-programming approach gives the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 .$$

Of course, if you do not know that an optimal solution for the set S_{ij} includes activity a_k, you must examine all activities in S_{ij} to find which one to choose, so that

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset , \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset . \end{cases} \tag{15.2}$$

You can then develop a recursive algorithm and memoize it, or you can work bottom-up and fill in table entries as you go along. But you would be overlooking another important characteristic of the activity-selection problem that you can use to great advantage.

Making the greedy choice

What if you could choose an activity to add to an optimal solution without having to first solve all the subproblems? That could save you from having to consider all the choices inherent in recurrence (15.2). In fact, for the activity-selection problem, you need to consider only one choice: the greedy choice.

What is the greedy choice for the activity-selection problem? Intuition suggests that you should choose an activity that leaves the resource available for as many other activities as possible. Of the activities you end up choosing, one of them must be the first one to finish. Intuition says, therefore, choose the activity in S with the earliest finish time, since that leaves the resource available for as many of the activities that follow it as possible. (If more than one activity in S has the earliest finish time, then choose any such activity.) In other words, since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity a_1. Choosing the first activity to finish is not the only way to think of making a greedy choice for this problem. Exercise 15.1-3 asks you to explore other possibilities.

Once you make the greedy choice, you have only one remaining subproblem to solve: finding activities that start after a_1 finishes. Why don't you have to consider activities that finish before a_1 starts? Because $s_1 < f_1$, and because f_1 is the earliest finish time of any activity, no activity can have a finish time less than or equal to s_1. Thus, all activities that are compatible with activity a_1 must start after a_1 finishes.

Furthermore, we have already established that the activity-selection problem exhibits optimal substructure. Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes. If you make the greedy choice of activity a_1, then S_1 remains as the only subproblem to solve.[1] Optimal substructure says that if a_1 belongs to an optimal solution, then an optimal solution to the original problem consists of activity a_1 and all the activities in an optimal solution to the subproblem S_1.

One big question remains: Is this intuition correct? Is the greedy choice—in which you choose the first activity to finish—always part of some optimal solution? The following theorem shows that it is.

---

[1] We sometimes refer to the sets S_k as subproblems rather than as just sets of activities. The context will make it clear whether we are referring to S_k as a set of activities or as a subproblem whose input is that set.

Theorem 15.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m belongs to some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = (A_k - \{a_j\}) \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are compatible, which follows because the activities in A_k are compatible, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

Although you might be able to solve the activity-selection problem with dynamic programming, Theorem 15.1 says that you don't need to. Instead, you can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain. Moreover, because you always choose the activity with the earliest finish time, the finish times of the activities that you choose must strictly increase. You can consider each activity just once overall, in monotonically increasing order of finish times.

An algorithm to solve the activity-selection problem does not need to work bottom-up, like a table-based dynamic-programming algorithm. Instead, it can work top-down, choosing an activity to put into the optimal solution that it constructs and then solving the subproblem of choosing activities from those that are compatible with those already chosen. Greedy algorithms typically have this top-down design: make a choice and then solve a subproblem, rather than the bottom-up technique of solving subproblems before making a choice.

A recursive greedy algorithm

Now that you know you can bypass the dynamic-programming approach and instead use a top-down, greedy algorithm, let's see a straightforward, recursive procedure to solve the activity-selection problem. The procedure RECURSIVE-ACTIVITY-SELECTOR on the following page takes the start and finish times of the activities, represented as arrays s and f,[2] the index k that defines the subproblem S_k it is to solve, and the size n of the original problem. It returns a maximum-

[2] Because the pseudocode takes s and f as arrays, it indexes into them with square brackets rather than with subscripts.

size set of mutually compatible activities in S_k. The procedure assumes that the n input activities are already ordered by monotonically increasing finish time, according to equation (15.1). If not, you can first sort them into this order in $O(n \lg n)$ time, breaking ties arbitrarily. In order to start, add the fictitious activity a_0 with $f_0 = 0$, so that subproblem S_0 is the entire set of activities S. The initial call, which solves the entire problem, is RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return ∅
```

Figure 15.2 shows how the algorithm operates on the activities in Figure 15.1. In a given recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n), the **while** loop of lines 2–3 looks for the first activity in S_k to finish. The loop examines $a_{k+1}, a_{k+2}, \ldots, a_n$, until it finds the first activity a_m that is compatible with a_k, which means that $s_m \geq f_k$. If the loop terminates because it finds such an activity, line 5 returns the union of $\{a_m\}$ and the maximum-size subset of S_m returned by the recursive call RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n). Alternatively, the loop may terminate because $m > n$, in which case the procedure has examined all activities in S_k without finding one that is compatible with a_k. In this case, $S_k = \emptyset$, and so line 6 returns $\emptyset$.

Assuming that the activities have already been sorted by finish times, the running time of the call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$ is $\Theta(n)$. To see why, observe that over all recursive calls, each activity is examined exactly once in the **while** loop test of line 2. In particular, activity a_i is examined in the last call made in which $k < i$.

An iterative greedy algorithm

The recursive procedure can be converted to an iterative one because the procedure RECURSIVE-ACTIVITY-SELECTOR is almost "tail recursive" (see Problem 7-5): it ends with a recursive call to itself followed by a union operation. It is usually a straightforward task to transform a tail-recursive procedure to an iterative form. In fact, some compilers for certain programming languages perform this task automatically.

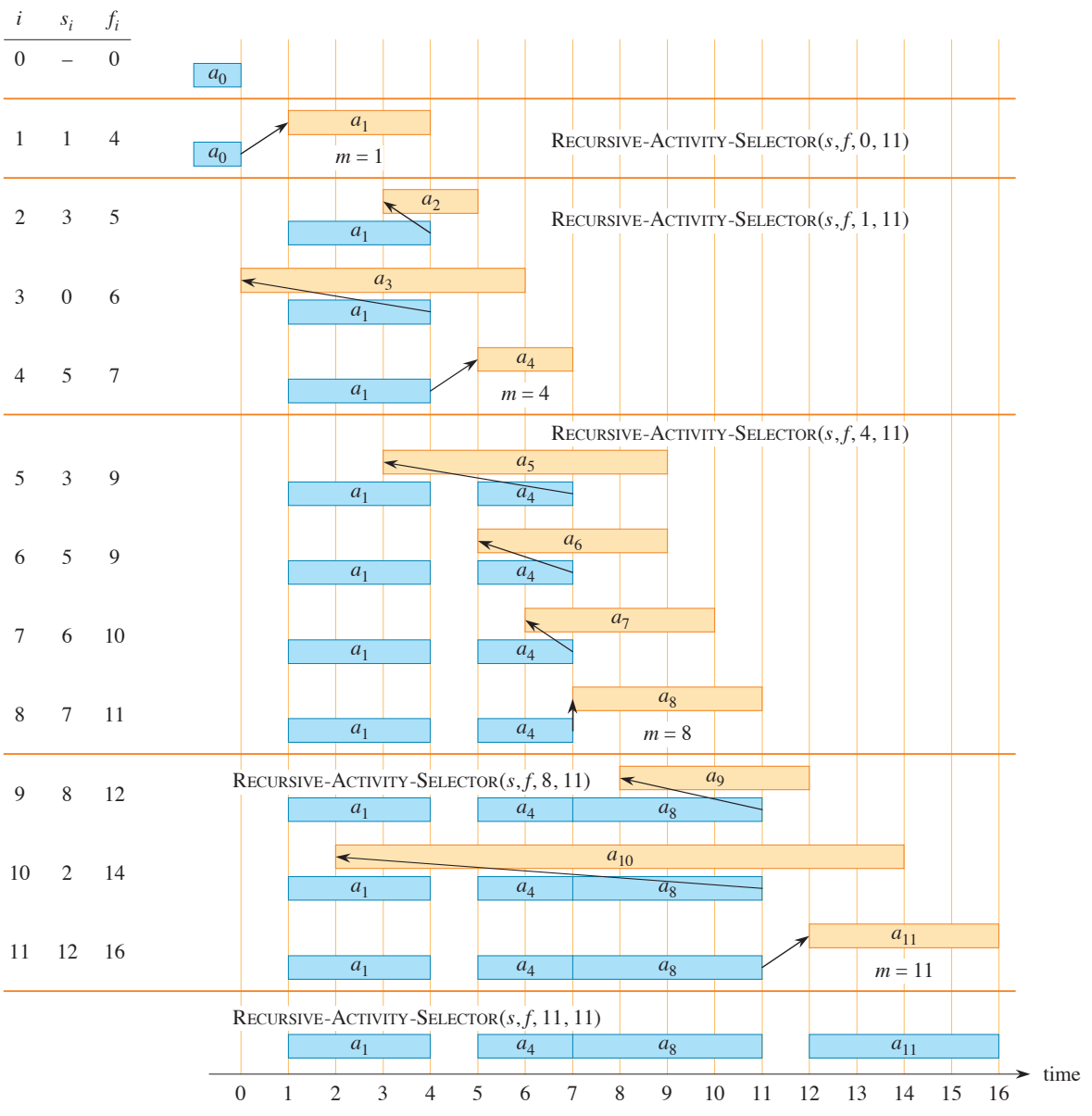


Figure 15.2 The operation of RECURSIVE-ACTIVITY-SELECTOR on the 11 activities from Figure 15.1. Activities considered in each recursive call appear between horizontal lines. The fictitious activity a_0 finishes at time 0, and the initial call RECURSIVE-ACTIVITY-SELECTOR(s, f, 0, 11), selects activity a_1. In each recursive call, the activities that have already been selected are blue, and the activity shown in tan is being considered. If the starting time of an activity occurs before the finish time of the most recently added activity (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is selected. The last recursive call, RECURSIVE-ACTIVITY-SELECTOR(s, f, 11, 11), returns $\emptyset$. The resulting set of selected activities is $\{a_1, a_4, a_8, a_{11}\}$.

The procedure GREEDY-ACTIVITY-SELECTOR is an iterative version of the procedure RECURSIVE-ACTIVITY-SELECTOR. It, too, assumes that the input activities are ordered by monotonically increasing finish time. It collects selected activities into a set A and returns this set when it is done.

```
GREEDY-ACTIVITY-SELECTOR(s, f, n)
1  A = {a_1}
2  k = 1
3  for m = 2 to n
4      if s[m] ≥ f[k]          // is a_m in S_k?
5          A = A ∪ {a_m}       // yes, so choose it
6          k = m               // and continue from there
7  return A
```

The procedure works as follows. The variable k indexes the most recent addition to A, corresponding to the activity a_k in the recursive version. Since the procedure considers the activities in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A. That is,

$$f_k = \max\{f_i : a_i \in A\} \ . \tag{15.3}$$

Lines 1–2 select activity a_1, initialize A to contain just this activity, and initialize k to index this activity. The **for** loop of lines 3–6 finds the earliest activity in S_k to finish. The loop considers each activity a_m in turn and adds a_m to A if it is compatible with all previously selected activities. Such an activity is the earliest in S_k to finish. To see whether activity a_m is compatible with every activity currently in A, it suffices by equation (15.3) to check (in line 4) that its start time s_m is not earlier than the finish time f_k of the activity most recently added to A. If activity a_m is compatible, then lines 5–6 add activity a_m to A and set k to m. The set A returned by the call GREEDY-ACTIVITY-SELECTOR(s, f) is precisely the set returned by the initial call RECURSIVE-ACTIVITY-SELECTOR$(s, f, 0, n)$.

Like the recursive version, GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Exercises

15.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (15.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (15.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

15.1-2

Suppose that instead of always selecting the first activity to finish, you instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

15.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

15.1-4

You are given a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. You wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the ***interval-graph coloring problem***. It is modeled by an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

15.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, the goal is to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

15.2 Elements of the greedy strategy

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm makes the choice that seems best at the moment. This heuristic strategy does not always produce an optimal solution, but as in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods.

The process that we followed in Section 15.1 to develop a greedy algorithm was a bit more involved than is typical. It consisted of the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution. (For the activity-selection problem, we formulated recurrence (15.2), but bypassed developing a recursive algorithm based solely on this recurrence.)
3. Show that if you make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

These steps highlighted in great detail the dynamic-programming underpinnings of a greedy algorithm. For example, the first cut at the activity-selection problem defined the subproblems S_{ij}, where both i and j varied. We then found that if you always make the greedy choice, you can restrict the subproblems to be of the form S_k.

An alternative approach is to fashion optimal substructure with a greedy choice in mind, so that the choice leaves just one subproblem to solve. In the activity-selection problem, start by dropping the second subscript and defining subproblems of the form S_k. Then prove that a greedy choice (the first activity a_m to finish in S_k), combined with an optimal solution to the remaining set S_m of compatible activities, yields an optimal solution to S_k. More generally, you can design greedy algorithms according to the following sequence of steps:

1. Cast the optimization problem as one in which you make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
3. Demonstrate optimal substructure by showing that, having made the greedy choice, what remains is a subproblem with the property that if you combine an

optimal solution to the subproblem with the greedy choice you have made, you arrive at an optimal solution to the original problem.

Later sections of this chapter will use this more direct process. Nevertheless, beneath every greedy algorithm, there is almost always a more cumbersome dynamic-programming solution.

How can you tell whether a greedy algorithm will solve a particular optimization problem? No way works all the time, but the greedy-choice property and optimal substructure are the two key ingredients. If you can demonstrate that the problem has these properties, then you are well on the way to developing a greedy algorithm for it.

Greedy-choice property

The first key ingredient is the ***greedy-choice property***: you can assemble a globally optimal solution by making locally optimal (greedy) choices. In other words, when you are considering which choice to make, you make the choice that looks best in the current problem, without considering results from subproblems.

Here is where greedy algorithms differ from dynamic programming. In dynamic programming, you make a choice at each step, but the choice usually depends on the solutions to subproblems. Consequently, you typically solve dynamic-programming problems in a bottom-up manner, progressing from smaller subproblems to larger subproblems. (Alternatively, you can solve them top down, but memoizing. Of course, even though the code works top down, you still must solve the subproblems before making a choice.) In a greedy algorithm, you make whatever choice seems best at the moment and then solve the subproblem that remains. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems before making the first choice, a greedy algorithm makes its first choice before solving any subproblems. A dynamic-programming algorithm proceeds bottom up, whereas a greedy strategy usually progresses top down, making one greedy choice after another, reducing each given problem instance to a smaller one.

Of course, you need to prove that a greedy choice at each step yields a globally optimal solution. Typically, as in the case of Theorem 15.1, the proof examines a globally optimal solution to some subproblem. It then shows how to modify the solution to substitute the greedy choice for some other choice, resulting in one similar, but smaller, subproblem.

You can usually make the greedy choice more efficiently than when you have to consider a wider set of choices. For example, in the activity-selection problem, assuming that the activities were already sorted in monotonically increasing order by finish times, each activity needed to be examined just once. By preprocessing

the input or by using an appropriate data structure (often a priority queue), you often can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal substructure

As we saw in Chapter 14, a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems. This property is a key ingredient of assessing whether dynamic programming applies, and it's also essential for greedy algorithms. As an example of optimal substructure, recall how Section 15.1 demonstrated that if an optimal solution to subproblem S_{ij} includes an activity a_k, then it must also contain optimal solutions to the subproblems S_{ik} and S_{kj}. Given this optimal substructure, we argued that if you know which activity to use as a_k, you can construct an optimal solution to S_{ij} by selecting a_k along with all activities in optimal solutions to the subproblems S_{ik} and S_{kj}. This observation of optimal substructure gave rise to the recurrence (15.2) that describes the value of an optimal solution.

You will usually use a more direct approach regarding optimal substructure when applying it to greedy algorithms. As mentioned above, you have the luxury of assuming that you arrived at a subproblem by having made the greedy choice in the original problem. All you really need to do is argue that an optimal solution to the subproblem, combined with the greedy choice already made, yields an optimal solution to the original problem. This scheme implicitly uses induction on the subproblems to prove that making the greedy choice at every step produces an optimal solution.

Greedy versus dynamic programming

Because both the greedy and dynamic-programming strategies exploit optimal substructure, you might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices or, conversely, you might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtle differences between the two techniques, let's investigate two variants of a classical optimization problem.

The ***0-1 knapsack problem*** is the following. A thief robbing a store wants to take the most valuable load that can be carried in a knapsack capable of carrying at most W pounds of loot. The thief can choose to take any subset of n items in the store. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. Which items should the thief take? (We call this the 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. The thief cannot take a fractional amount of an item or take an item more than once.)

In the ***fractional knapsack problem***, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot and an item in the fractional knapsack problem as more like gold dust.

Both knapsack problems exhibit the optimal-substructure property. For the 0-1 problem, if the most valuable load weighing at most W pounds includes item j, then the remaining load must be the most valuable load weighing at most $W - w_j$ pounds that the thief can take from the $n - 1$ original items excluding item j. For the comparable fractional problem, if if the most valuable load weighing at most W pounds includes weight w of item j, then the remaining load must be the most valuable load weighing at most $W - w$ pounds that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Although the problems are similar, a greedy strategy works to solve the fractional knapsack problem, but not the 0-1 problem. To solve the fractional problem, first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and the thief can still carry more, then the thief takes as much as possible of the item with the next greatest value per pound, and so forth, until reaching the weight limit W. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg n)$ time. You are asked to prove that the fractional knapsack problem has the greedy-choice property in Exercise 15.2-1.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 15.3(a). This example has three items and a knapsack that can hold 50 pounds. Item 1 weighs 10 pounds and is worth \$60. Item 2 weighs 20 pounds and is worth \$100. Item 3 weighs 30 pounds and is worth \$120. Thus, the value per pound of item 1 is \$6 per pound, which is greater than the value per pound of either item 2 (\$5 per pound) or item 3 (\$4 per pound). The greedy strategy, therefore, would take item 1 first. As you can see from the case analysis in Figure 15.3(b), however, the optimal solution takes items 2 and 3, leaving item 1 behind. The two possible solutions that take item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 15.3(c). Taking item 1 doesn't work in the 0-1 problem, because the thief is unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the load. In the 0-1 problem, when you consider whether to include an item in the knapsack, you must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item before you can make the choice. The problem formulated in this way gives rise to many overlapping sub-

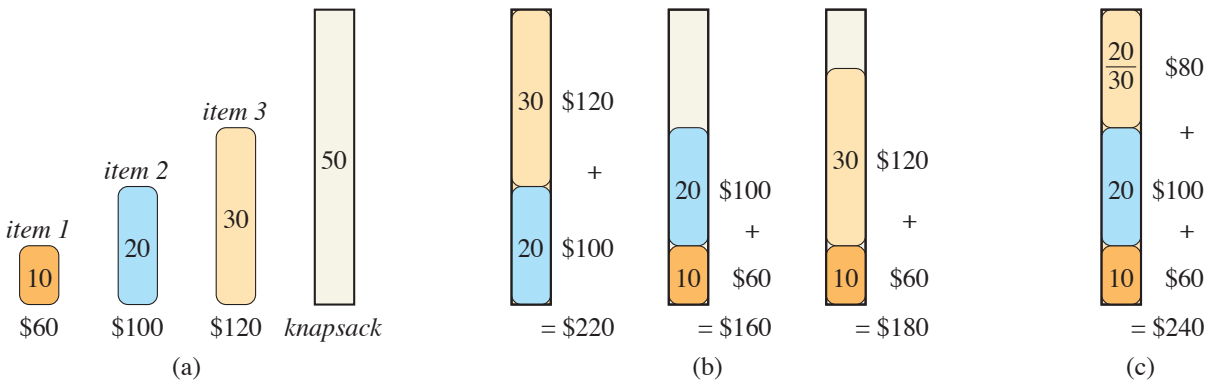


Figure 15.3 An example showing that the greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

problems—a hallmark of dynamic programming, and indeed, as Exercise 15.2-2 asks you to show, you can use dynamic programming to solve the 0-1 problem.

Exercises

15.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

15.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(n\,W)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in the knapsack.

15.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

15.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. The professor plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water and can skate m miles before

running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. The professor has an official North Dakota state map, which shows all the places along U.S. 2 to refill water and the distances between these locations.

The professor's goal is to minimize the number of water stops along the route across the state. Give an efficient method by which the professor can determine which water stops to make. Prove that your strategy yields an optimal solution, and give its running time.

15.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \ldots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

★ ***15.2-6***

Show how to solve the fractional knapsack problem in $O(n)$ time.

15.2-7

You are given two sets A and B, each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the ith element of set A, and let b_i be the ith element of set B. You then receive a payoff of $\prod_{i=1}^{n} a_i^{b_i}$. Give an algorithm that maximizes your payoff. Prove that your algorithm maximizes the payoff, and state its running time, omitting the time for reordering the sets.

15.3 Huffman codes

Huffman codes compress data well: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. The data arrive as a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (its frequency) to build up an optimal way of representing each character as a binary string.

Suppose that you have a 100,000-character data file that you wish to store compactly and you know that the 6 distinct characters in the file occur with the frequencies given by Figure 15.4. The character `a` occurs 45,000 times, the character `b` occurs 13,000 times, and so on.

You have many options for how to represent such a file of information. Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short)

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Figure 15.4 A character-coding problem. A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated. With each character represented by a 3-bit codeword, encoding the file requires 300,000 bits. With the variable-length code shown, the encoding requires only 224,000 bits.

in which each character is represented by a unique binary string, which we call a ***codeword***. If you use a ***fixed-length code***, you need $\lceil \lg n \rceil$ bits to represent $n \geq 2$ characters. For 6 characters, therefore, you need 3 bits: a = 000, b = 001, c = 010, d = 011, e = 100, and f = 101. This method requires 300,000 bits to encode the entire file. Can you do better?

A ***variable-length code*** can do considerably better than a fixed-length code. The idea is simple: give frequent characters short codewords and infrequent characters long codewords. Figure 15.4 shows such a code. Here, the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

to represent the file, a savings of approximately 25%. In fact, this is an optimal character code for this file, as we shall see.

Prefix-free codes

We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called ***prefix-free codes***. Although we won't prove it here, a prefix-free code can always achieve the optimal data compression among any character code, and so we suffer no loss of generality by restricting our attention to prefix-free codes.

Encoding is always simple for any binary character code: just concatenate the codewords representing each character of the file. For example, with the variable-length prefix-free code of Figure 15.4, the 4-character file face has the encoding $1100 \cdot 0 \cdot 100 \cdot 1101 = 110001001101$, where "$\cdot$" denotes concatenation.

Prefix-free codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. You can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string 100011001101 parses uniquely as $100 \cdot 0 \cdot 1100 \cdot 1101$, which decodes to cafe.

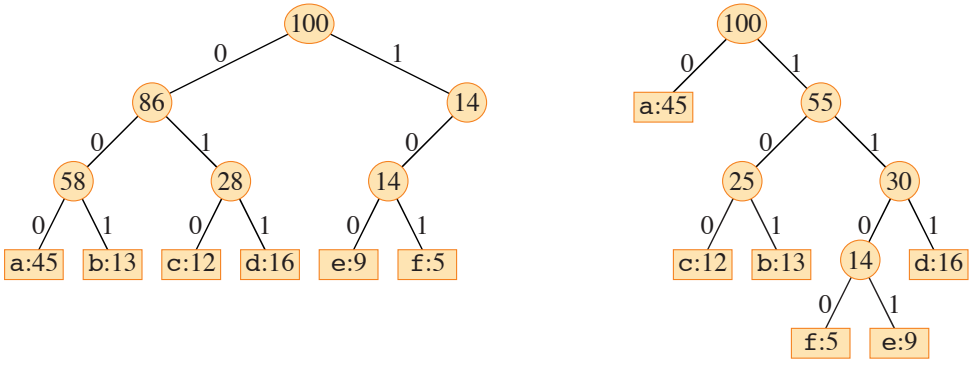


Figure 15.5 Trees corresponding to the coding schemes in Figure 15.4. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. All frequencies are in thousands. **(a)** The tree corresponding to the fixed-length code a = 000, b = 001, c = 010, d = 011, e = 100, f = 101. **(b)** The tree corresponding to the optimal prefix-free code a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100.

The decoding process needs a convenient representation for the prefix-free code so that you can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. Interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." Figure 15.5 shows the trees for the two codes of our example. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.

An optimal code for a file is always represented by a *full* binary tree, in which every nonleaf node has two children (see Exercise 15.3-2). The fixed-length code in our example is not optimal since its tree, shown in Figure 15.5(a), is not a full binary tree: it contains codewords beginning with 10, but none beginning with 11. Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix-free code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C| - 1$ internal nodes (see Exercise B.5-3 on page 1175).

Given a tree T corresponding to a prefix-free code, we can compute the number of bits required to encode a file. For each character c in the alphabet C, let the attribute $c.freq$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is thus

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c) \,, \tag{15.4}$$

which we define as the ***cost*** of the tree T.

Constructing a Huffman code

Huffman invented a greedy algorithm that constructs an optimal prefix-free code, called a ***Huffman code*** in his honor. In line with our observations in Section 15.2, its proof of correctness relies on the greedy-choice property and optimal substructure. Rather than demonstrating that these properties hold and then developing pseudocode, we present the pseudocode first. Doing so will help clarify how the algorithm makes greedy choices.

The procedure HUFFMAN assumes that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. The algorithm builds the tree T corresponding to an optimal code in a bottom-up manner. It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the *freq* attribute, to identify the two least-frequent objects to merge together. The result of merging two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN(C)
 1  n = |C|
 2  Q = C
 3  for i = 1 to n − 1
 4       allocate a new node z
 5       x = EXTRACT-MIN(Q)
 6       y = EXTRACT-MIN(Q)
 7       z.left = x
 8       z.right = y
 9       z.freq = x.freq + y.freq
10       INSERT(Q, z)
11  return EXTRACT-MIN(Q)      // the root of the tree is the only node left
```

For our example, Huffman's algorithm proceeds as shown in Figure 15.6. Since the alphabet contains 6 letters, the initial queue size is $n = 6$, and 5 merge steps build the tree. The final tree represents the optimal prefix-free code. The codeword for a letter is the sequence of edge labels on the simple path from the root to the letter.

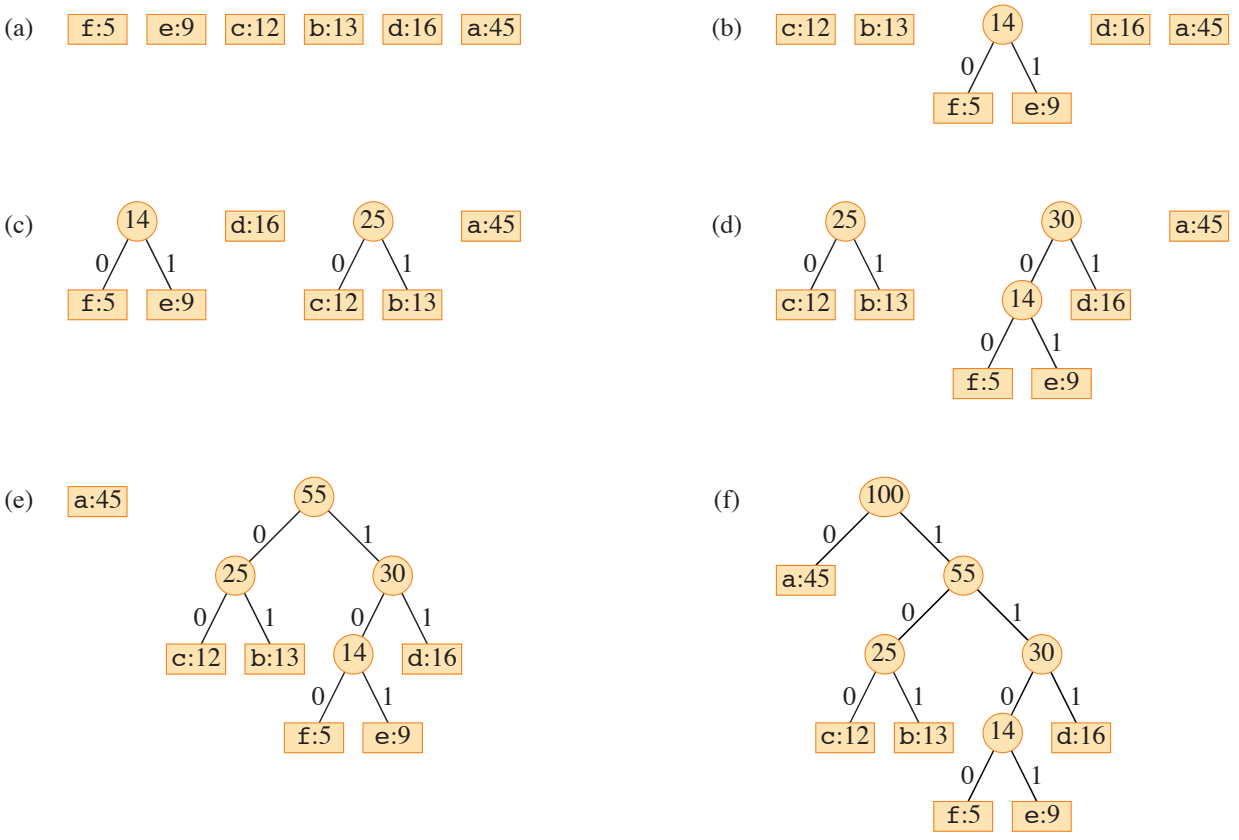


Figure 15.6 The steps of Huffman's algorithm for the frequencies given in Figure 15.4. Each part shows the contents of the queue sorted into increasing order by frequency. Each step merges the two trees with the lowest frequencies. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter. **(a)** The initial set of $n = 6$ nodes, one for each letter. **(b)–(e)** Intermediate stages. **(f)** The final tree.

The HUFFMAN procedure works as follows. Line 2 initializes the min-priority queue Q with the characters in C. The **for** loop in lines 3–10 repeatedly extracts the two nodes x and y of lowest frequency from the queue and replaces them in the queue with a new node z representing their merger. The frequency of z is computed as the sum of the frequencies of x and y in line 9. The node z has x as its left child and y as its right child. (This order is arbitrary. Switching the left and right child of any node yields a different code of the same cost.) After $n - 1$ mergers, line 11 returns the one node left in the queue, which is the root of the code tree.

The algorithm produces the same result without the variables x and y, assigning the values returned by the EXTRACT-MIN calls directly to $z.left$ and $z.right$ in lines 7 and 8, and changing line 9 to $z.freq = z.left.freq + z.right.freq$. We'll use the node names x and y in the proof of correctness, however, so we leave them in.

The running time of Huffman's algorithm depends on how the min-priority queue Q is implemented. Let's assume that it's implemented as a binary min-heap (see Chapter 6). For a set C of n characters, the BUILD-MIN-HEAP procedure discussed in Section 6.3 can initialize Q in line 2 in $O(n)$ time. The **for** loop in lines 3–10 executes exactly $n - 1$ times, and since each heap operation runs in $O(\lg n)$ time, the loop contributes $O(n \lg n)$ to the running time. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.

Correctness of Huffman's algorithm

To prove that the greedy algorithm HUFFMAN is correct, we'll show that the problem of determining an optimal prefix-free code exhibits the greedy-choice and optimal-substructure properties. The next lemma shows that the greedy-choice property holds.

Lemma 15.2 (Optimal prefix-free codes have the greedy-choice property)
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix-free code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof The idea of the proof is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree. In such a tree, the codewords for x and y have the same length and differ only in the last bit.

Let a and b be any two characters that are sibling leaves of maximum depth in T. Without loss of generality, assume that $a.freq \le b.freq$ and $x.freq \le y.freq$. Since $x.freq$ and $y.freq$ are the two lowest leaf frequencies, in order, and $a.freq$ and $b.freq$ are two arbitrary frequencies, in order, we have $x.freq \le a.freq$ and $y.freq \le b.freq$.

In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$, but $x.freq = b.freq$ implies that $a.freq = b.freq = x.freq = y.freq$ (see Exercise 15.3-1), and the lemma would be trivially true. Therefore, assume that $x.freq \ne b.freq$, which means that $x \ne b$.

As Figure 15.7 shows, imagine exchanging the positions in T of a and x to produce a tree T', and then exchanging the positions in T' of b and y to produce a

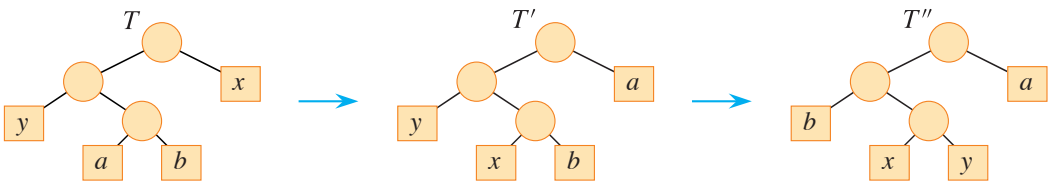


Figure 15.7 An illustration of the key step in the proof of Lemma 15.2. In the optimal tree T, leaves a and b are two siblings of maximum depth. Leaves x and y are the two characters with the lowest frequencies. They appear in arbitrary positions in T. Assuming that $x \neq b$, swapping leaves a and x produces tree T', and then swapping leaves b and y produces tree T''. Since each swap does not increase the cost, the resulting tree T'' is also an optimal tree.

tree T'' in which x and y are sibling leaves of maximum depth. (Note that if $x = b$ but $y \neq a$, then tree T'' does not have x and y as sibling leaves of maximum depth. Because we assume that $x \neq b$, this situation cannot occur.) By equation (15.4), the difference in cost between T and T' is

$$\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \geq 0 \,,
\end{aligned}$$

because both $a.freq - x.freq$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.freq - x.freq$ is nonnegative because x is a minimum-frequency leaf, and $d_T(a) - d_T(x)$ is nonnegative because a is a leaf of maximum depth in T. Similarly, exchanging y and b does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T') \leq B(T)$, and since T is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus, T'' is an optimal tree in which x and y appear as sibling leaves of maximum depth, from which the lemma follows. ■

Lemma 15.2 implies that the process of building up an optimal tree by mergers can, without loss of generality, begin with the greedy choice of merging together those two characters of lowest frequency. Why is this a greedy choice? We can view the cost of a single merger as being the sum of the frequencies of the two items being merged. Exercise 15.3-4 shows that the total cost of the tree constructed equals the sum of the costs of its mergers. Of all possible mergers at each step, HUFFMAN chooses the one that incurs the least cost.

The next lemma shows that the problem of constructing optimal prefix-free codes has the optimal-substructure property.

Lemma 15.3 (Optimal prefix-free codes have the optimal-substructure property)
Let C be a given alphabet with frequency $c.freq$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = (C - \{x, y\}) \cup \{z\}$. Define $freq$ for all characters in C' with the same values as in C, along with $z.freq = x.freq + y.freq$. Let T' be any tree representing an optimal prefix-free code for alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix-free code for the alphabet C.

Proof We first show how to express the cost $B(T)$ of tree T in terms of the cost $B(T')$ of tree T', by considering the component costs in equation (15.4). For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$\begin{aligned} x.freq \cdot d_T(x) + y.freq \cdot d_T(y) &= (x.freq + y.freq)(d_{T'}(z) + 1) \\ &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq) , \end{aligned}$$

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

or, equivalently,

$$B(T') = B(T) - x.freq - y.freq .$$

We now prove the lemma by contradiction. Suppose that T does not represent an optimal prefix-free code for C. Then there exists an optimal tree T'' such that $B(T'') < B(T)$. Without loss of generality (by Lemma 15.2), T'' has x and y as siblings. Let T''' be the tree T'' with the common parent of x and y replaced by a leaf z with frequency $z.freq = x.freq + y.freq$. Then

$$\begin{aligned} B(T''') &= B(T'') - x.freq - y.freq \\ &< B(T) - x.freq - y.freq \\ &= B(T') , \end{aligned}$$

yielding a contradiction to the assumption that T' represents an optimal prefix-free code for C'. Thus, T must represent an optimal prefix-free code for the alphabet C. ■

Theorem 15.4
Procedure HUFFMAN produces an optimal prefix-free code.

Proof Immediate from Lemmas 15.2 and 15.3. ■

Exercises

15.3-1

Explain why, in the proof of Lemma 15.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

15.3-2

Prove that a non-full binary tree cannot correspond to an optimal prefix-free code.

15.3-3

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

`a`:1 `b`:1 `c`:2 `d`:3 `e`:5 `f`:8 `g`:13 `h`:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

15.3-4

Prove that the total cost $B(T)$ of a full binary tree T for a code equals the sum, over all internal nodes, of the combined frequencies of the two children of the node.

15.3-5

Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

15.3-6

Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

15.3-7

A data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

15.3-8

Show that no lossless (invertible) compression scheme can guarantee that for every input file, the corresponding output file is shorter. (*Hint:* Compare the number of possible files with the number of possible encoded files.)

15.4 Offline caching

Computer systems can decrease the time to access data by storing a subset of the main memory in the ***cache***: a small but faster memory. A cache organizes data into ***cache blocks*** typically comprising 32, 64, or 128 bytes. You can also think of main memory as a cache for disk-resident data in a virtual-memory system. Here, the blocks are called ***pages***, and 4096 bytes is a typical size.

As a computer program executes, it makes a sequence of memory requests. Say that there are n memory requests, to data in blocks $b_1, b_2, \ldots, b_n$, in that order. The blocks in the access sequence might not be distinct, and indeed, any given block is usually accessed multiple times. For example, a program that accesses four distinct blocks p, q, r, s might make a sequence of requests to blocks $s, q, s, q, q, s, p, p, r, s, s, q, p, r, q$. The cache can hold up to some fixed number k of cache blocks. It starts out empty before the first request. Each request causes at most one block to enter the cache and at most one block to be evicted from the cache. Upon a request for block b_i, any one of three scenarios may occur:

1. Block b_i is already in the cache, due to a previous request for the same block. The cache remains unchanged. This situation is known as a ***cache hit***.
2. Block b_i is not in the cache at that time, but the cache contains fewer than k blocks. In this case, block b_i is placed into the cache, so that the cache contains one more block than it did before the request.
3. Block b_i is not in the cache at that time and the cache is full: it contains k blocks. Block b_i is placed into the cache, but before that happens, some other block in the cache must be evicted from the cache in order to make room.

The latter two situations, in which the requested block is not already in the cache, are called ***cache misses***. The goal is to minimize the number of cache misses or, equivalently, to maximize the number of cache hits, over the entire sequence of n requests. A cache miss that occurs while the cache holds fewer than k blocks—that is, as the cache is first being filled up—is known as a ***compulsory miss***, since no prior decision could have kept the requested block in the cache. When a cache miss occurs and the cache is full, ideally the choice of which block to evict should allow for the smallest possible number of cache misses over the entire sequence of future requests.

Typically, caching is an online problem. That is, the computer has to decide which blocks to keep in the cache without knowing the future requests. Here, however, let's consider the offline version of this problem, in which the computer knows in advance the entire sequence of n requests and the cache size k, with a goal of minimizing the total number of cache misses.

To solve this offline problem, you can use a greedy strategy called ***furthest-in-future***, which chooses to evict the block in the cache whose next access in the request sequence comes furthest in the future. Intuitively, this strategy makes sense: if you're not going to need something for a while, why keep it around? We'll show that the furthest-in-future strategy is indeed optimal by showing that the offline caching problem exhibits optimal substructure and that furthest-in-future has the greedy-choice property.

Now, you might be thinking that since the computer usually doesn't know the sequence of requests in advance, there is no point in studying the offline problem. Actually, there is. In some situations, you do know the sequence of requests in advance. For example, if you view the main memory as the cache and the full set of data as residing on disk (or a solid-state drive), there are algorithms that plan out the entire set of reads and writes in advance. Furthermore, we can use the number of cache misses produced by an optimal algorithm as a baseline for comparing how well online algorithms perform. We'll do just that in Section 27.3.

Offline caching can even model real-world problems. For example, consider a scenario where you know in advance a fixed schedule of n events at known locations. Events may occur at a location multiple times, not necessarily consecutively. You are managing a group of k agents, you need to ensure that you have one agent at each location when an event occurs, and you want to minimize the number of times that agents have to move. Here, the agents are like the blocks, the events are like the requests, and moving an agent is akin to a cache miss.

Optimal substructure of offline caching

To show that the offline problem exhibits optimal substructure, let's define the subproblem (C, i) as processing requests for blocks $b_i, b_{i+1}, \ldots, b_n$ with cache configuration C at the time that the request for block b_i occurs, that is, C is a subset of the set of blocks such that $|C| \le k$. A solution to subproblem (C, i) is a sequence of decisions that specifies which block to evict (if any) upon each request for blocks $b_i, b_{i+1}, \ldots, b_n$. An optimal solution to subproblem (C, i) minimizes the number of cache misses.

Consider an optimal solution S to subproblem (C, i), and let C' be the contents of the cache after processing the request for block b_i in solution S. Let S' be the subsolution of S for the resulting subproblem $(C', i + 1)$. If the request for b_i results in a cache hit, then the cache remains unchanged, so that $C' = C$. If the request for block b_i results in a cache miss, then the contents of the cache change, so that $C' \neq C$. We claim that in either case, S' is an optimal solution to subproblem $(C', i + 1)$. Why? If S' is not an optimal solution to subproblem $(C', i + 1)$, then there exists another solution S'' to subproblem $(C', i + 1)$ that makes fewer cache misses than S'. Combining S'' with the decision of S at the request for

block b_i yields another solution that makes fewer cache misses than S, which contradicts the assumption that S is an optimal solution to subproblem (C, i).

To quantify a recursive solution, we need a little more notation. Let $R_{C,i}$ be the set of all cache configurations that can immediately follow configuration C after processing a request for block b_i. If the request results in a cache hit, then the cache remains unchanged, so that $R_{C,i} = \{C\}$. If the request for b_i results in a cache miss, then there are two possibilities. If the cache is not full ($|C| < k$), then the cache is filling up and the only choice is to insert b_i into the cache, so that $R_{C,i} = \{C \cup \{b_i\}\}$. If the cache is full ($|C| = k$) upon a cache miss, then $R_{C,i}$ contains k potential configurations: one for each candidate block in C that could be evicted and replaced by block b_i. In this case, $R_{C,i} = \{(C - \{x\}) \cup \{b_i\} : x \in C\}$. For example, if $C = \{p, q, r\}$, $k = 3$, and block s is requested, then $R_{C,i} = \{\{p, q, s\}, \{p, r, s\}, \{q, r, s\}\}$.

Let $miss(C, i)$ denote the minimum number of cache misses in a solution for subproblem (C, i). Here is a recurrence for $miss(C, i)$:

$$miss(C, i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C \ , \\ 1 & \text{if } i = n \text{ and } b_n \notin C \ , \\ miss(C, i+1) & \text{if } i < n \text{ and } b_i \in C \ , \\ 1 + \min\{miss(C', i+1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C \ . \end{cases}$$

Greedy-choice property

To prove that the furthest-in-future strategy yields an optimal solution, we need to show that optimal offline caching exhibits the greedy-choice property. Combined with the optimal-substructure property, the greedy-choice property will prove that furthest-in-future produces the minimum possible number of cache misses.

Theorem 15.5 (Optimal offline caching has the greedy-choice property)
Consider a subproblem (C, i) when the cache C contains k blocks, so that it is full, and a cache miss occurs. When block b_i is requested, let $z = b_m$ be the block in C whose next access is furthest in the future. (If some block in the cache will never again be referenced, then consider any such block to be block z, and add a dummy request for block $z = b_m = b_{n+1}$.) Then evicting block z upon a request for block b_i is included in some optimal solution for the subproblem (C, i).

Proof Let S be an optimal solution to subproblem (C, i). If S evicts block z upon the request for block b_i, then we are done, since we have shown that some optimal solution includes evicting z.

So now suppose that optimal solution S evicts some other block x when block b_i is requested. We'll construct another solution S' to subproblem (C, i) which, upon

the request for b_i, evicts block z instead of x and induces no more cache misses than S does, so that S' is also optimal. Because different solutions may yield different cache configurations, denote by $C_{S,j}$ the configuration of the cache under solution S just before the request for some block b_j, and likewise for solution S' and $C_{S',j}$. We'll show how to construct S' with the following properties:

1. For $j = i+1, \ldots, m$, let $D_j = C_{S,j} \cap C_{S',j}$. Then, $|D_j| \geq k-1$, so that the cache configurations $C_{S,j}$ and $C_{S',j}$ differ by at most one block. If they differ, then $C_{S,j} = D_j \cup \{z\}$ and $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$.
2. For each request of blocks $b_i, \ldots, b_{m-1}$, if solution S has a cache hit, then solution S' also has a cache hit.
3. For all $j > m$, the cache configurations $C_{S,j}$ and $C_{S',j}$ are identical.
4. Over the sequence of requests for blocks $b_i, \ldots, b_m$, the number of cache misses produced by solution S' is at most the number of cache misses produced by solution S.

We'll prove inductively that these properties hold for each request.

1. We proceed by induction on j, for $j = i+1, \ldots, m$. For the base case, the initial caches $C_{S,i}$ and $C_{S',i}$ are identical. Upon the request for block b_i, solution S evicts x and solution S' evicts z. Thus, cache configurations $C_{S,i+1}$ and $C_{S',i+1}$ differ by just one block, $C_{S,i+1} = D_{i+1} \cup \{z\}$, $C_{S',i+1} = D_{i+1} \cup \{x\}$, and $x \neq z$.

   The inductive step defines how solution S' behaves upon a request for block b_j for $i+1 \leq j \leq m-1$. The inductive hypothesis is that property 1 holds when b_j is requested. Because $z = b_m$ is the block in $C_{S,i}$ whose next reference is furthest in the future, we know that $b_j \neq z$. We consider several scenarios:

   - If $C_{S,j} = C_{S',j}$ (so that $|D_j| = k$), then solution S' makes the same decision upon the request for b_j as S makes, so that $C_{S,j+1} = C_{S',j+1}$.
   - If $|D_j| = k-1$ and $b_j \in D_j$, then both caches already contain block b_j, and both solutions S and S' have cache hits. Therefore, $C_{S,j+1} = C_{S,j}$ and $C_{S',j+1} = C_{S',j}$.
   - If $|D_j| = k-1$ and $b_j \notin D_j$, then because $C_{S,j} = D_j \cup \{z\}$ and $b_j \neq z$, solution S has a cache miss. It evicts either block z or some block $w \in D_j$.
     - If solution S evicts block z, then $C_{S,j+1} = D_j \cup \{b_j\}$. There are two cases, depending on whether $b_j = y$:
       - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Thus, $C_{S,j+1} = C_{S',j+1}$.
       - If $b_j \neq y$, then solution S' has a cache miss. It evicts block y, so that $C_{S',j+1} = D_j \cup \{b_j\}$, and again $C_{S,j+1} = C_{S',j+1}$.

- If solution S evicts some block $w \in D_j$, then $C_{S,j+1} = (D_j - \{w\}) \cup \{b_j, z\}$. Once again, there are two cases, depending on whether $b_j = y$:
  - If $b_j = y$, then solution S' has a cache hit, so that $C_{S',j+1} = C_{S',j} = D_j \cup \{b_j\}$. Since $w \in D_j$ and w was not evicted by solution S', we have $w \in C_{S',j+1}$. Therefore, $w \notin D_{j+1}$ and $b_j \in D_{j+1}$, so that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$. Thus, $C_{S,j+1} = D_{j+1} \cup \{z\}$, $C_{S',j+1} = D_{j+1} \cup \{w\}$, and because $w \neq z$, property 1 holds when block b_{j+1} is requested. (In other words, block w replaces block y in property 1.)
  - If $b_j \neq y$, then solution S' has a cache miss. It evicts block w, so that $C_{S',j+1} = (D_j - \{w\}) \cup \{b_j, y\}$. Therefore, we have that $D_{j+1} = (D_j - \{w\}) \cup \{b_j\}$ and so $C_{S,j+1} = D_{j+1} \cup \{z\}$ and $C_{S',j+1} = D_{j+1} \cup \{y\}$.

2. In the above discussion about maintaining property 1, solution S may have a cache hit in only the first two cases, and solution S' has a cache hit in these cases if and only if S does.

3. If $C_{S,m} = C_{S',m}$, then solution S' makes the same decision upon the request for block $z = b_m$ as S makes, so that $C_{S,m+1} = C_{S',m+1}$. If $C_{S,m} \neq C_{S',m}$, then by property 1, $C_{S,m} = D_m \cup \{z\}$ and $C_{S',m} = D_m \cup \{y\}$, where $y \neq z$. In this case, solution S has a cache hit, so that $C_{S,m+1} = C_{S,m} = D_m \cup \{z\}$. Solution S' evicts block y and brings in block z, so that $C_{S',m+1} = D_m \cup \{z\} = C_{S,m+1}$. Thus, regardless of whether or not $C_{S,m} = C_{S',m}$, we have $C_{S,m+1} = C_{S',m+1}$, and starting with the request for block b_{m+1}, solution S' simply makes the same decisions as S.

4. By property 2, upon the requests for blocks $b_i, \ldots, b_{m-1}$, whenever solution S has a cache hit, so does S'. Only the request for block $b_m = z$ remains to be considered. If S has a cache miss upon the request for b_m, then regardless of whether S' has a cache hit or a cache miss, we are done: S' has at most the same number of cache misses as S.

   So now suppose that S has a cache hit and S' has a cache miss upon the request for b_m. We'll show that there exists a request for at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ in which the request results in a cache miss for S and a cache hit for S', thereby compensating for what happens upon the request for block b_m. The proof is by contradiction. Assume that no request for blocks $b_{i+1}, \ldots, b_{m-1}$ results in a cache miss for S and a cache hit for S'.

   We start by observing that once the caches $C_{S,j}$ and $C_{S'j}$ are equal for some $j > i$, they remain equal thereafter. Observe also that if $b_m \in C_{S,m}$ and $b_m \notin C_{S',m}$, then $C_{S,m} \neq C_{S',m}$. Therefore, solution S cannot have evicted block z upon the requests for blocks $b_i, \ldots, b_{m-1}$, for if it had, then these two

cache configurations would be equal. The remaining possibility is that upon each of these requests, we had $C_{S,j} = D_j \cup \{z\}$, $C_{S',j} = D_j \cup \{y\}$ for some block $y \neq z$, and solution S evicted some block $w \in D_j$. Moreover, since none of these requests resulted in a cache miss for S and a cache hit for S', the case of $b_j = y$ never occurred. That is, for every request of blocks $b_{i+1}, \ldots, b_{m-1}$, the requested block b_j was never the block $y \in C_{S',j} - C_{S,j}$. In these cases, after processing the request, we had $C_{S',j+1} = D_{j+1} \cup \{y\}$: the difference between the two caches did not change. Now, let's go back to the request for block b_i, where afterward, we had $C_{S',i+1} = D_{i+1} \cup \{x\}$. Because every succeeding request until requesting block b_m did not change the difference between the caches, we had $C_{S',j} = D_j \cup \{x\}$ for $j = i + 1, \ldots, m$.

By definition, block $z = b_m$ is requested after block x. That means at least one of blocks $b_{i+1}, \ldots, b_{m-1}$ is block x. But for $j = i + 1, \ldots, m$, we have $x \in C_{S',j}$ and $x \notin C_{S,j}$, so that at least one of these requests had a cache hit for S' and a cache miss for S, a contradiction. We conclude that if solution S has a cache hit and solution S' has a cache miss upon the request for block b_m, then some earlier request had the opposite result, and so solution S' produces no more cache misses than solution S. Since S is assumed to be optimal, S' is optimal as well. ■

Along with the optimal-substructure property, Theorem 15.5 tells us that the furthest-in-future strategy yields the minimum number of cache misses.

Exercises

15.4-1

Write pseudocode for a cache manager that uses the furthest-in-future strategy. It should take as input a set C of blocks in the cache, the number of blocks k that the cache can hold, a sequence $b_1, b_2, \ldots, b_n$ of requested blocks, and the index i into the sequence for the block b_i being requested. For each request, it should print out whether a cache hit or cache miss occurs, and for each cache miss, it should also print out which block, if any, is evicted.

15.4-2

Real cache managers do not know the future requests, and so they often use the past to decide which block to evict. The ***least-recently-used***, or ***LRU***, strategy evicts the block that, of all blocks currently in the cache, was the least recently requested. (You can think of LRU as "furthest-in-past.") Give an example of a request sequence in which the LRU strategy is not optimal, by showing that it induces more cache misses than the furthest-in-future strategy does on the same request sequence.

15.4-3

Professor Croesus suggests that in the proof of Theorem 15.5, the last clause in property 1 can change to $C_{S',j} = D_j \cup \{x\}$ or, equivalently, require the block y given in property 1 to always be the block x evicted by solution S upon the request for block b_i. Show where the proof breaks down with this requirement.

15.4-4

This section has assumed that at most one block is placed into the cache whenever a block is requested. You can imagine, however, a strategy in which multiple blocks may enter the cache upon a single request. Show that for every solution that allows multiple blocks to enter the cache upon each request, there is another solution that brings in only one block upon each request and is at least as good.

Problems

15-1 Coin changing

Consider the problem of making change for n cents using the smallest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

b. Suppose that the available coins are in denominations that are powers of c: the denominations are $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

d. Give an $O(nk)$-time algorithm that makes change for any set of k different coin denominations using the smallest number of coins, assuming that one of the coins is a penny.

15-2 Scheduling to minimize average completion time

You are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete. Let C_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n) \sum_{i=1}^{n} C_i$. For example, suppose that there are two tasks a_1 and a_2 with $p_1 = 3$ and $p_2 = 5$, and consider the schedule

in which a_2 runs first, followed by a_1. Then we have $C_2 = 5$, $C_1 = 8$, and the average completion time is $(5+8)/2 = 6.5$. If task a_1 runs first, however, then we have $C_1 = 3$, $C_2 = 8$, and the average completion time is $(3+8)/2 = 5.5$.

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run nonpreemptively, that is, once task a_i starts, it must run continuously for p_i units of time until it is done. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its ***release time*** b_i. Suppose also that tasks may be ***preempted***, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $b_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

Chapter notes

Much more material on greedy algorithms can be found in Lawler [276] and Papadimitriou and Steiglitz [353]. The greedy algorithm first appeared in the combinatorial optimization literature in a 1971 article by Edmonds [131].

The proof of correctness of the greedy algorithm for the activity-selection problem is based on that of Gavril [179].

Huffman codes were invented in 1952 [233]. Lelewer and Hirschberg [294] surveys data-compression techniques known as of 1987.

The furthest-in-future strategy was proposed by Belady [41], who suggested it for virtual-memory systems. Alternative proofs that furthest-in-future is optimal appear in articles by Lee et al. [284] and Van Roy [443].

16 Amortized Analysis

Imagine that you join Buff's Gym. Buff charges a membership fee of \$60 per month, plus \$3 for every time you use the gym. Because you are disciplined, you visit Buff's Gym every day during the month of November. On top of the \$60 monthly charge for November, you pay another $3 \times \$30 = \$90$ that month. Although you can think of your fees as a flat fee of \$60 and another \$90 in daily fees, you can think about it in another way. All together, you pay \$150 over 30 days, or an average of \$5 per day. When you look at your fees in this way, you are ***amortizing*** the monthly fee over the 30 days of the month, spreading it out at \$2 per day.

You can do the same thing when you analyze running times. In an ***amortized analysis***, you average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, you show that if you average over a sequence of operations, then the average cost of an operation is small, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved. An amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 16.1 starts with aggregate analysis, in which you determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. You take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 16.2 covers the accounting method, in which you determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as "pre-

paid credit" on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 16.3 discusses the potential method, which is like the accounting method in that you determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the "potential energy" of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We'll use use two examples in this chapter to examine each of these three methods. One is a stack with the additional operation MULTIPOP, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation INCREMENT.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not—appear in the code. If, for example, you assign a credit to an object x when using the accounting method, you have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When you perform an amortized analysis, you often gain insight into a particular data structure, and this insight can help you optimize the design. For example, Section 16.4 will use the potential method to analyze a dynamically expanding and contracting table.

16.1 Aggregate analysis

In ***aggregate analysis***, you show that for all n, a sequence of n operations takes $T(n)$ *worst-case* time in total. In the worst case, the average cost, or ***amortized cost***, per operation is therefore $T(n)/n$. This amortized cost applies to each operation, even when there are several types of operations in the sequence. The other two methods we shall study in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

As the first example of aggregate analysis, let's analyze stacks that have been augmented with a new operation. Section 10.1.3 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S.

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

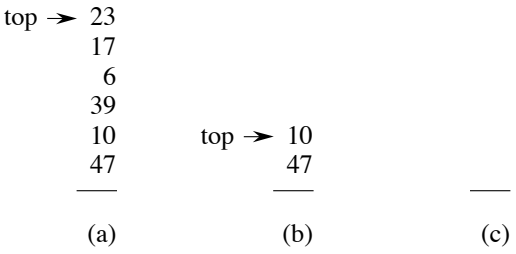


Figure 16.1 The action of MULTIPOP on a stack S, shown initially in **(a)**. The top 4 objects are popped by MULTIPOP$(S, 4)$, whose result is shown in **(b)**. The next operation is MULTIPOP$(S, 7)$, which empties the stack—shown in **(c)**—since fewer than 7 objects remained.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each to be 1. The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$.

Now let's add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. Of course, the procedure assumes that k is positive, and otherwise, the MULTIPOP operation leaves the stack unchanged. In the pseudocode for MULTIPOP, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise. Figure 16.1 shows an example of MULTIPOP.

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k − 1
```

What is the running time of MULTIPOP(S, k) on a stack of s objects? The actual running time is linear in the number of POP operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number min $\{s, k\}$ of objects popped off the stack. Each iteration of the loop makes one call to POP in line 2. Thus, the total cost of MULTIPOP is min $\{s, k\}$, and the actual running time is a linear function of this cost.

Now let's analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since the sequence contains at most n MULTIPOP operations costing $O(n)$ each.

Although this analysis is correct, the $O(n^2)$ result, which came from considering the worst-case cost of each operation individually, is not tight.

Yes, a single MULTIPOP might be expensive, but an aggregate analysis shows that any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack has an upper bound on its cost of $O(n)$. Why? An object cannot be popped from the stack unless it was first pushed. Therefore, the number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n. For any value of n, any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time. Averaging over the n operations gives an average cost per operation of $O(n)/n = O(1)$. Aggregate analysis assigns the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

To recap: although the average cost, and hence the running time, of a stack operation is $O(1)$, the analysis did not rely on probabilistic reasoning. Instead, the analysis yielded a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded that the average cost per operation—that is, the amortized cost—is $O(1)$.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing a k-bit binary counter that counts upward from 0. An array $A[0:k-1]$ of bits represents the counter. A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Initially, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \ldots, k-1$. To add 1 (modulo 2^k) to the value in the counter, call the INCREMENT procedure.

```
INCREMENT(A, k)
1  i = 0
2  while i < k and A[i] == 1
3       A[i] = 0
4       i = i + 1
5  if i < k
6       A[i] = 1
```

Figure 16.2 shows what happens to a binary counter when INCREMENT is called 16 times, starting with the initial value 0 and ending with the value 16. Each iteration of the **while** loop in lines 2–4 adds a 1 into position i. If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 16.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded in blue. The running cost for flipping bits is shown at the right. The total cost is always less than twice the total number of INCREMENT operations.

position $i + 1$ during the next iteration of the loop. Otherwise, the loop ends, and then, if $i < k$, $A[i]$ must be 0, so that line 6 adds a 1 into position i, flipping the 0 to a 1. If the loop ends with $i = k$, then the call of INCREMENT flipped all k bits from 1 to 0. The cost of each INCREMENT operation is linear in the number of bits flipped.

As with the stack example, a cursory analysis yields a bound that is correct but not tight. A single execution of INCREMENT takes $\Theta(k)$ time in the worst case, in which all the bits in array A are 1. Thus, a sequence of n INCREMENT operations on an initially zero counter takes $O(nk)$ time in the worst case.

Although a single call of INCREMENT might flip all k bits, not all bits flip upon each call. (Note the similarity to MULTIPOP, where a single call might pop many objects, but not every call pops many objects.) As Figure 16.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of n INCREMENT operations on an initially zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times. Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT operations. In general, for $i = 0, 1, \ldots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter. For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip. The total number

of flips in the sequence is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

by equation (A.7) on page 1142. Thus, a sequence of n INCREMENT operations on an initially zero counter takes $O(n)$ time in the worst case. The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$.

Exercises

16.1-1
If the set of stack operations includes a MULTIPUSH operation, which pushes k items onto the stack, does the $O(1)$ bound on the amortized cost of stack operations continue to hold?

16.1-2
Show that if a DECREMENT operation is included in the k-bit counter example, n operations can cost as much as $\Theta(nk)$ time.

16.1-3
Use aggregate analysis to determine the amortized cost per operation for a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise.

16.2 The accounting method

In the ***accounting method*** of amortized analysis, you assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount that you charge an operation is its ***amortized cost***. When an operation's amortized cost exceeds its actual cost, you assign the difference to specific objects in the data structure as ***credit***. Credit can help pay for later operations whose amortized cost is less than their actual cost. Thus, you can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

You must choose the amortized costs of operations carefully. If you want to use amortized costs to show that in the worst case the average cost per operation is

small, you must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. Moreover, as in aggregate analysis, the upper bound must apply to all sequences of operations. Let's denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\hat{c}_i$. Then you need to have

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i \tag{16.1}$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$. By inequality (16.1), the total credit associated with the data structure must be nonnegative at all times. If you ever allowed the total credit to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred. In that case, for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, you must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, we return to the stack example. Recall that the actual costs of the operations were

PUSH	1 ,
POP	1 ,
MULTIPOP	$\min\{s, k\}$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH	2 ,
POP	0 ,
MULTIPOP	0 .

The amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable, and thus all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

Now let's see how to pay for any sequence of stack operations by charging the amortized costs. Let \$1 represent each unit of cost. At first, the stack is empty. Recall the analogy of Section 10.1.3 between the stack data structure and a stack of plates in a cafeteria. Upon pushing a plate onto the stack, use \$1 to pay the

actual cost of the push, leaving a credit of \$1 (out of the \$2 charged). Place that \$1 of credit on top of the plate. At any point in time, every plate on the stack has \$1 of credit on it.

The \$1 stored on the plate serves to prepay the cost of popping the plate from the stack. A POP operation incurs no charge: pay the actual cost of popping a plate by taking the \$1 of credit off the plate. Thus, by charging the PUSH operation a little bit more, we can view the POP operation as free.

Moreover, the MULTIPOP operation also incurs no charge, since it's just repeated POP operations, each of which is free. If a MULTIPOP operation pops k plates, then the actual cost is paid by the k dollars stored on the k plates. Because each plate on the stack has \$1 of credit on it, and the stack always has a nonnegative number of plates, the amount of credit is always nonnegative. Thus, for *any* sequence of n PUSH, POP, and MULTIPOP operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a binary counter

As another illustration of the accounting method, let's analyze the INCREMENT operation on a binary counter that starts at 0. Recall that the running time of this operation is proportional to the number of bits flipped, which serves as the cost for this example. Again, we'll use \$1 to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, the amortized cost to set a 0-bit to 1 is \$2. When a bit is set to 1, \$1 of the \$2 pays to actually set the bit. The second \$1 resides on the bit as credit to be used later if and when the bit is reset to 0. At any point in time, every 1-bit in the counter has \$1 of credit on it, and thus resetting a bit to 0 can be viewed as costing nothing, and the \$1 on the bit prepays for the reset.

Here is how to determine the amortized cost of INCREMENT. The cost of resetting the bits to 0 within the **while** loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit to 1, in line 6, and therefore the amortized cost of an INCREMENT operation is at most \$2. The number of 1-bits in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Exercises

16.2-1

You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automat-

ically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

16.2-2
Redo Exercise 16.1-3 using an accounting method of analysis.

16.2-3
You wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(n)$ time on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

16.3 The potential method

Instead of representing prepaid work as credit stored with specific objects in the data structure, the ***potential method*** of amortized analysis represents the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations. The potential applies to the data structure as a whole rather than to specific objects within the data structure.

The potential method works as follows. Starting with an initial data structure D_0, a sequence of n operations occurs. For each $i = 1, 2, \ldots, n$, let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}. A ***potential function*** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the ***potential*** associated with D_i. The ***amortized cost*** $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \tag{16.2}$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. By equation (16.2), the total amortized cost of the n operations is

$$\begin{aligned} \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \tag{16.3}$$

The second equation follows from equation (A.12) on page 1143 because the $\Phi(D_i)$ terms telescope.

If you can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$. In practice, you don't always know how many operations might be performed. Therefore, if you require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then you guarantee, as in the accounting method, that you've paid in advance. It's usually simplest to just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all i. (See Exercise 16.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is positive, then the amortized cost $\hat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases. If the potential difference is negative, then the amortized cost represents an undercharge to the ith operation, and the decrease in the potential pays for the actual cost of the operation.

The amortized costs defined by equations (16.2) and (16.3) depend on the choice of the potential function Φ. Different potential functions may yield different amortized costs, yet still be upper bounds on the actual costs. You will often find tradeoffs that you can make in choosing a potential function. The best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to the example of the stack operations PUSH, POP, and MULTIPOP. We define the potential function Φ on a stack to be the number of objects in the stack. The potential of the empty initial stack D_0 is $\Phi(D_0) = 0$. Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) \, . \end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Now let's compute the amortized costs of the various stack operations. If the ith operation on a stack containing s objects is a PUSH operation, then the potential difference is

$$\begin{aligned} \Phi(D_i) - \Phi(D_{i-1}) &= (s+1) - s \\ &= 1 \, . \end{aligned}$$

By equation (16.2), the amortized cost of this PUSH operation is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 \\ &= 2 .\end{aligned}$$

Suppose that the ith operation on the stack of s objects is MULTIPOP(S, k), which causes $k' = \min\{s, k\}$ objects to be popped off the stack. The actual cost of the operation is k', and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k' .$$

Thus, the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing a binary counter

As another example of the potential method, we revisit incrementing a k-bit binary counter. This time, the potential of the counter after the ith INCREMENT operation is defined to be the number of 1-bits in the counter after the ith operation, which we'll denote by b_i.

Here is how to compute the amortized cost of an INCREMENT operation. Suppose that the ith INCREMENT operation resets t_i bits to 0. The actual cost c_i of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1. If $b_i = 0$, then the ith operation had reset all k bits to 0, and so $b_{i-1} = t_i = k$. If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$. In either case, $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

The amortized cost is therefore

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

If the counter starts at 0, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENT operations is $O(n)$.

The potential method provides a simple and clever way to analyze the counter even when it does not start at 0. The counter starts with b_0 1-bits, and after n INCREMENT operations it has b_n 1-bits, where $0 \leq b_0, b_n \leq k$. Rewrite equation (16.3) as

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

Since $\Phi(D_0) = b_0$, $\Phi(D_n) = b_n$, and $\hat{c}_i \leq 2$ for all $1 \leq i \leq n$, the total actual cost of n INCREMENT operations is

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

In particular, $b_0 \leq k$ means that as long as $k = O(n)$, the total actual cost is $O(n)$. In other words, if at least $n = \Omega(k)$ INCREMENT operations occur, the total actual cost is $O(n)$, no matter what initial value the counter contains.

Exercises

16.3-1

Suppose you have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

16.3-2

Redo Exercise 16.1-3 using a potential method of analysis.

16.3-3

Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever be stored in the heap.

16.3-4
What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

16.3-5
Show how to implement a queue with two ordinary stacks (Exercise 10.1-7) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

16.3-6
Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT(S, x) inserts x into S.

DELETE-LARGER-HALF(S) deletes the largest $\lceil |S| / 2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

16.4 Dynamic tables

When you design an application that uses a table, you do not always know in advance how many items the table will hold. You might allocate space for the table, only to find out later that it is not enough. The program must then reallocate the table with a larger size and copy all items stored in the original table over into the new, larger table. Similarly, if many items have been deleted from the table, it might be worthwhile to reallocate the table with a smaller size. This section studies this problem of dynamically expanding and contracting a table. Amortized analyses will show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, you'll see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Let's assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single ***slot***, that is, a space for one item. Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant: it could be a stack (Section 10.1.3), a heap (Chapter 6), a hash table (Chapter 11), or something else.

It is convenient to use a concept introduced in Section 11.2, where we analyzed hashing. The ***load factor*** $\alpha(T)$ of a nonempty table T is defined as the number of items stored in the table divided by the size (number of slots) of the table. An empty table (one with no slots) has size 0, and its load factor is defined to be 1. If the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

We start by analyzing a dynamic table that allows only insertion and then move on to the more general case that supports both insertion and deletion.

16.4.1 Table expansion

Let's assume that storage for a table is allocated as an array of slots. A table fills up when all slots have been used or, equivalently, when its load factor is 1.[1] In some software environments, upon an attempt to insert an item into a full table, the only alternative is to abort with an error. The scenario in this section assumes, however, that the software environment, like many modern ones, provides a memory-management system that can allocate and free blocks of storage on request. Thus, upon inserting an item into a full table, the system can ***expand*** the table by allocating a new table with more slots than the old table had. Because the table must always reside in contiguous memory, the system must allocate a new array for the larger table and then copy items from the old table into the new table.

A common heuristic allocates a new table with twice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least $1/2$, and thus the amount of wasted space never exceeds half the total space in the table.

The TABLE-INSERT procedure on the following page assumes that T is an object representing the table. The attribute $T.table$ contains a pointer to the block of storage representing the table, $T.num$ contains the number of items in the table, and $T.size$ gives the total number of slots in the table. Initially, the table is empty: $T.num = T.size = 0$.

There are two types of insertion here: the TABLE-INSERT procedure itself and the ***elementary insertion*** into a table in lines 6 and 10. We can analyze the running time of TABLE-INSERT in terms of the number of elementary insertions by assigning a cost of 1 to each elementary insertion. In most computing environments, the overhead for allocating an initial table in line 2 is constant and the overhead for allocating and freeing storage in lines 5 and 7 is dominated by the cost of transfer-

[1] In some situations, such as an open-address hash table, it's better to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 16.4-2.)

```
TABLE-INSERT(T, x)
 1  if T.size == 0
 2      allocate T.table with 1 slot
 3      T.size = 1
 4  if T.num == T.size
 5      allocate new-table with 2 · T.size slots
 6      insert all items in T.table into new-table
 7      free T.table
 8      T.table = new-table
 9      T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

ring items in line 6. Thus, the actual running time of TABLE-INSERT is linear in the number of elementary insertions. An ***expansion*** occurs when lines 5–9 execute.

Now, we'll use all three amortized analysis techniques to analyze a sequence of n TABLE-INSERT operations on an initially empty table. First, we need to determine the actual cost c_i of the ith operation. If the current table has room for the new item (or if this is the first operation), then $c_i = 1$, since the only elementary insertion performed is the one in line 10. If the current table is full, however, and an expansion occurs, then $c_i = i$: the cost is 1 for the elementary insertion in line 10 plus $i - 1$ for the items copied from the old table to the new table in line 6. For n operations, the worst-case cost of an operation is $O(n)$, which leads to an upper bound of $O(n^2)$ on the total running time for n operations.

This bound is not tight, because the table rarely expands in the course of n TABLE-INSERT operations. Specifically, the ith operation causes an expansion only when $i - 1$ is an exact power of 2. The amortized cost of an operation is in fact $O(1)$, as an aggregate analysis shows. The cost of the ith operation is

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is an exact power of 2 ,} \\ 1 & \text{otherwise .} \end{cases}$$

The total cost of n TABLE-INSERT operations is therefore

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \qquad \text{(by equation (A.6) on page 1142)} \\ &= 3n \ , \end{aligned}$$

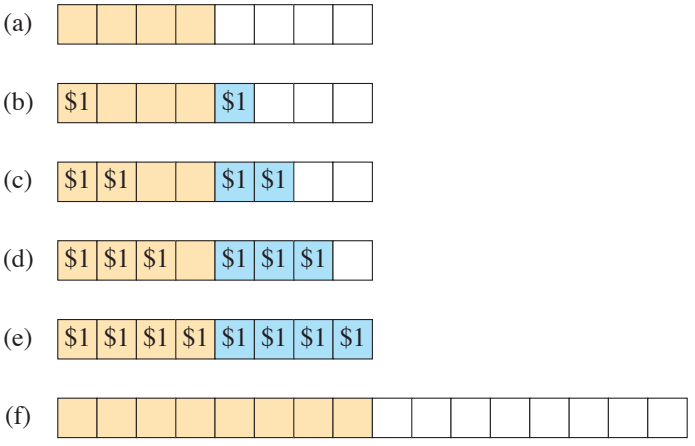


Figure 16.3 Analysis of table expansion by the accounting method. Each call of TABLE-INSERT charges \$3 as follows: \$1 to pay for the elementary insertion, \$1 on the item inserted as prepayment for it to be reinserted later, and \$1 on an item that was already in the table, also as prepayment for reinsertion. **(a)** The table immediately after an expansion, with 8 slots, 4 items (tan slots), and no stored credit. **(b)–(e)** After each of 4 calls to TABLE-INSERT, the table has one more item, with \$1 stored on the new item and \$1 stored on one of the 4 items that were present immediately after the expansion. Slots with these new items are blue. **(f)** Upon the next call to TABLE-INSERT, the table is full, and so it expands again. Each item had \$1 to pay for it to be reinserted. Now the table looks as it did in part (a), with no stored credit but 16 slots and 8 items.

because at most n operations cost 1 each and the costs of the remaining operations form a geometric series. Since the total cost of n TABLE-INSERT operations is bounded by $3n$, the amortized cost of a single operation is at most 3.

The accounting method can provide some intuition for why the amortized cost of a TABLE-INSERT operation should be 3. You can think of each item paying for three elementary insertions: inserting itself into the current table, moving itself the next time that the table expands, and moving some other item that was already in the table the next time that the table expands. For example, suppose that the size of the table is m immediately after an expansion, as shown in Figure 16.3 for $m = 8$. Then the table holds $m/2$ items, and it contains no credit. Each call of TABLE-INSERT charges \$3. The elementary insertion that occurs immediately costs \$1. Another \$1 resides on the item inserted as credit. The third \$1 resides as credit on one of the $m/2$ items already in the table. The table will not fill again until another $m/2 - 1$ items have been inserted, and thus, by the time the table contains m items and is full, each item has \$1 on it to pay for it to be reinserted it during the expansion.

Now, let's see how to use the potential method. We'll use it again in Section 16.4.2 to design a TABLE-DELETE operation that has an $O(1)$ amortized cost

as well. Just as the accounting method had no stored credit immediately after an expansion—that is, when $T.num = T.size/2$—let's define the potential to be 0 when $T.num = T.size/2$. As elementary insertions occur, the potential needs to increase enough to pay for all the reinsertions that will happen when the table next expands. The table fills after another $T.size/2$ calls of TABLE-INSERT, when $T.num = T.size$. The next call of TABLE-INSERT after these $T.size/2$ calls triggers an expansion with a cost of $T.size$ to reinsert all the items. Therefore, over the course of $T.size/2$ calls of TABLE-INSERT, the potential must increase from 0 to $T.size$. To achieve this increase, let's design the potential so that each call of TABLE-INSERT increases it by

$$\frac{T.size}{T.size/2} = 2 ,$$

until the table expands. You can see that the potential function

$$\Phi(T) = 2(T.num - T.size/2) \tag{16.4}$$

equals 0 immediately after the table expands, when $T.num = T.size/2$, and it increases by 2 upon each insertion until the table fills. Once the table fills, that is, when $T.num = T.size$, the potential $\Phi(T)$ equals $T.size$. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

To analyze the amortized costs of table operations, it is convenient to think in terms of the change in potential due to each operation. Letting Φ_i denote the potential after the ith operation, we can rewrite equation (16.2) as

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= c_i + \Delta\Phi_i , \end{aligned}$$

where $\Delta\Phi_i$ is the change in potential due to the ith operation. First, consider the case when the ith insertion does not cause the table to expand. In this case, $\Delta\Phi_i$ is 2. Since the actual cost c_i is 1, the amortized cost is

$$\begin{aligned} \hat{c}_i &= c_i + \Delta\Phi_i \\ &= 1 + 2 \\ &= 3 . \end{aligned}$$

Now, consider the change in potential when the table does expand during the ith insertion because it was full immediately before the insertion. Let num_i denote the number of items stored in the table after the ith operation and $size_i$ denote the total size of the table after the ith operation, so that $size_{i-1} = num_{i-1} = i - 1$

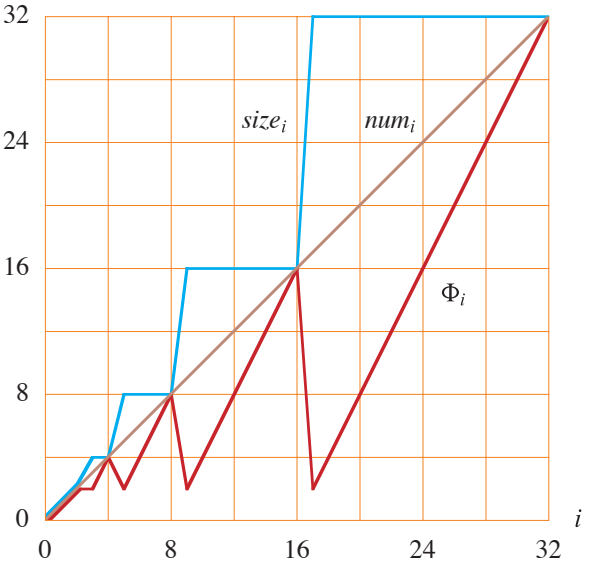


Figure 16.4 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table (the brown line), the number $size_i$ of slots in the table (the blue line), and the potential $\Phi_i = 2(num_i - size_i/2)$ (the red line), each being measured after the ith operation. Immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterward, the potential drops to 0, but it immediately increases by 2 upon insertion of the item that caused the expansion.

and therefore $\Phi_{i-1} = 2(size_{i-1} - size_{i-1}/2) = size_{i-1} = i - 1$. Immediately after the expansion, the potential goes down to 0, and then the new item is inserted, causing the potential to increase to $\Phi_i = 2$. Thus, when the ith insertion triggers an expansion, $\Delta\Phi_i = 2 - (i - 1) = 3 - i$. When the table expands in the ith TABLE-INSERT operation, the actual cost c_i equals i (to reinsert $i - 1$ items and insert the ith item), giving an amortized cost of

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi_i \\ &= i + (3 - i) \\ &= 3 .\end{aligned}$$

Figure 16.4 plots the values of num_i, $size_i$, and Φ_i against i. Notice how the potential builds to pay for expanding the table.

16.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. In order to limit the amount of wasted space, however, you might want to ***contract*** the table when the load factor becomes too small. Ta-

ble contraction is analogous to table expansion: when the number of items in the table drops too low, allocate a new, smaller table and then copy the items from the old table into the new one. You can then free the storage for the old table by returning it to the memory-management system. In order to not waste space, yet keep the amortized costs low, the insertion and deletion procedures should preserve two properties:

- the load factor of the dynamic table is bounded below by a positive constant, as well as above by 1, and
- the amortized cost of a table operation is bounded above by a constant.

The actual cost of each operation equals the number of elementary insertions or deletions.

You might think that if you double the table size upon inserting an item into a full table, then you should halve the size when deleting an item that would cause the table to become less than half full. This strategy does indeed guarantee that the load factor of the table never drops below $1/2$. Unfortunately, it can also cause the amortized cost of an operation to be quite large. Consider the following scenario. Perform n operations on a table T of size $n/2$, where n is an exact power of 2. The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$. At the end of this sequence of insertions, $T.num = T.size = n/2$. For the second $n/2$ operations, perform the following sequence:

insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

The first insertion causes the table to expand to size n. The two deletions that follow cause the table to contract back to size $n/2$. Two further insertions cause another expansion, and so forth. The cost of each expansion and contraction is $\Theta(n)$, and there are $\Theta(n)$ of them. Thus, the total cost of the n operations is $\Theta(n^2)$, making the amortized cost of an operation $\Theta(n)$.

The problem with this strategy is that after the table expands, not enough deletions occur to pay for a contraction. Likewise, after the table contracts, not enough insertions take place to pay for an expansion.

How can we solve this problem? Allow the load factor of the table to drop below $1/2$. Specifically, continue to double the table size upon inserting an item into a full table, but halve the table size when deleting an item causes the table to become less than $1/4$ full, rather than $1/2$ full as before. The load factor of the table is therefore bounded below by the constant $1/4$, and the load factor is $1/2$ immediately after a contraction.

An expansion or contraction should exhaust all the built-up potential, so that immediately after expansion or contraction, when the load factor is $1/2$, the table's potential is 0. Figure 16.5 shows the idea. As the load factor deviates from $1/2$, the

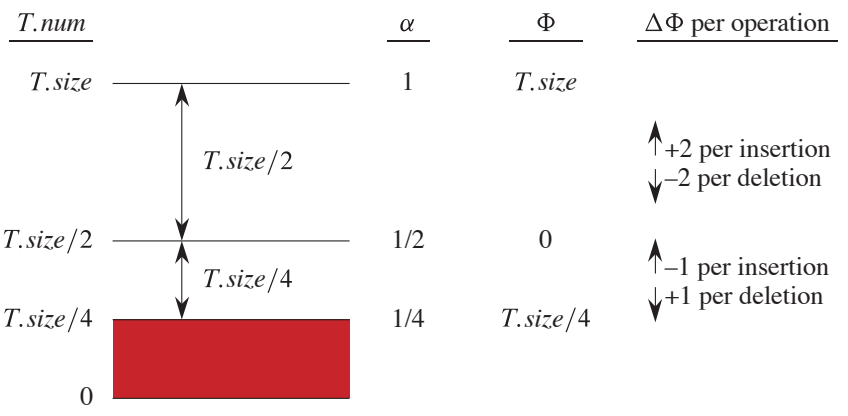


Figure 16.5 How to think about the potential function Φ for table insertion and deletion. When the load factor α is $1/2$, the potential is 0. In order to accumulate sufficient potential to pay for reinserting all $T.size$ items when the table fills, the potential needs to increase by 2 upon each insertion when $\alpha \geq 1/2$. Correspondingly, the potential decreases by 2 upon each deletion that leaves $\alpha \geq 1/2$. In order to accrue enough potential to cover the cost of reinserting all $T.size/4$ items when the table contracts, the potential needs to increase by 1 upon each deletion when $\alpha < 1/2$, and correspondingly the potential decreases by 1 upon each insertion that leaves $\alpha<1/2$. The red area represents load factors less than $1/4$, which are not allowed.

potential increases so that by the time an expansion or contraction occurs, the table has garnered sufficient potential to pay for copying all the items into the newly allocated table. Thus, the potential function should grow to $T.num$ by the time that the load factor has either increased to 1 or decreased to $1/4$. Immediately after either expanding or contracting the table, the load factor goes back to $1/2$ and the table's potential reduces back to 0.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT. We assume that if a contraction occurs during TABLE-DELETE, it occurs after the item is deleted from the table. The analysis assumes that whenever the number of items in the table drops to 0, the table occupies no storage. That is, if $T.num = 0$, then $T.size = 0$.

How do we design a potential function that gives constant amortized time for both insertion and deletion? When the load factor is at least $1/2$, the same potential function, $\Phi(T) = 2(T.num - T.size/2)$, that we used for insertion still works. When the table is at least half full, each insertion increases the potential by 2 if the table does not expand, and each deletion reduces the potential by 2 if it does not cause the load factor to drop below $1/2$.

What about when the load factor is less than $1/2$, that is, when $1/4 \leq \alpha(T) < 1/2$? As before, when $\alpha(T) = 1/2$, so that $T.num = T.size/2$, the potential $\Phi(T)$ should be 0. To get the load factor from $1/2$ down to $1/4$, $T.size/4$ deletions need

to occur, at which time $T.num = T.size/4$. To pay for all the reinsertions, the potential must increase from 0 to $T.size/4$ over these $T.size/4$ deletions. Therefore, for each call of TABLE-DELETE until the table contracts, the potential should increase by

$$\frac{T.size/4}{T.size/4} = 1 .$$

Likewise, when $\alpha < 1/2$, each call of TABLE-INSERT should decrease the potential by 1. When $1/4 \le \alpha(T) < 1/2$, the potential function

$$\Phi(T) = T.size/2 - T.num$$

produces this desired behavior.

Putting the two cases together, we get the potential function

$$\Phi(T) = \begin{cases} 2(T.num - T.size/2) & \text{if } \alpha(T) \ge 1/2 , \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2 . \end{cases} \tag{16.5}$$

The potential of an empty table is 0 and the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence. Figure 16.6 illustrates how the potential function behaves over a sequence of insertions and deletions.

Now, let's determine the amortized costs of each operation. As before, let num_i denote the number of items stored in the table after the ith operation, $size_i$ denote the total size of the table after the ith operation, $\alpha_i = num_i/size_i$ denote the load factor after the ith operation, Φ_i denote the potential after the ith operation, and $\Delta\Phi_i$ denote the change in potential due to the ith operation. Initially, $num_0 = 0$, $size_0 = 0$, and $\Phi_0 = 0$.

The cases in which the table does not expand or contract and the load factor does not cross $\alpha = 1/2$ are straightforward. As we have seen, if $\alpha_{i-1} \ge 1/2$ and the ith operation is an insertion that does not cause the table to expand, then $\Delta\Phi_i = 2$. Likewise, if the ith operation is a deletion and $\alpha_i \ge 1/2$, then $\Delta\Phi_i = -2$. Furthermore, if $\alpha_{i-1} < 1/2$ and the ith operation is a deletion that does not trigger a contraction, then $\Delta\Phi_i = 1$, and if the ith operation is an insertion and $\alpha_i < 1/2$, then $\Delta\Phi_i = -1$. In other words, if no expansion or contraction occurs and the load factor does not cross $\alpha = 1/2$, then

- if the load factor stays at or above $1/2$, then the potential increases by 2 for an insertion and decreases by 2 for a deletion, and
- if the load factor stays below $1/2$, then the potential increases by 1 for a deletion and decreases by 1 for an insertion.

In each of these cases, the actual cost c_i of the ith operation is just 1, and so

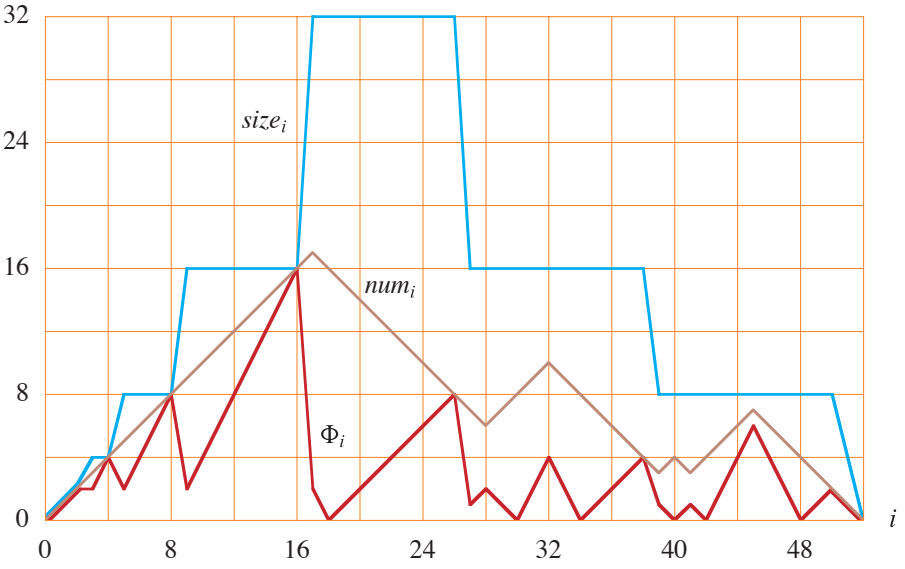


Figure 16.6 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table (the brown line), the number $size_i$ of slots in the table (the blue line), and the potential (the red line)

$$\Phi_i = \begin{cases} 2(num_i - size_i/2) & \text{if } \alpha_i \geq 1/2\,, \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2\,, \end{cases}$$

where $\alpha_i = num_i/size_i$, each measured after the ith operation. Immediately before an expansion or contraction, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table.

- if the ith operation is an insertion, its amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i$, which is $1 + 2 = 3$ if the load factor stays at or above $1/2$, and $1 + (-1) = 0$ if the load factor stays below $1/2$, and
- if the ith operation is a deletion, its amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i$, which is $1 + (-2) = -1$ if the load factor stays at or above $1/2$, and $1 + 1 = 2$ if the load factor stays below $1/2$.

Four cases remain: an insertion that takes the load factor from below $1/2$ to $1/2$, a deletion that takes the load factor from $1/2$ to below $1/2$, a deletion that causes the table to contract, and an insertion that causes the table to expand. We analyzed that last case at the end of Section 16.4.1 to show that its amortized cost is 3.

When the ith operation is a deletion that causes the table to contract, we have $num_{i-1} = size_{i-1}/4$ before the contraction, then the item is deleted, and finally $num_i = size_i/2 - 1$ after the contraction. Thus, by equation (16.5) we have

$$\begin{aligned}\Phi_{i-1} &= size_{i-1}/2 - num_{i-1} \\ &= size_{i-1}/2 - size_{i-1}/4 \\ &= size_{i-1}/4 \,,\end{aligned}$$

which also equals the actual cost c_i of deleting one item and copying $size_{i-1}/4 - 1$ items into the new, smaller table. Since $num_i = size_i/2 - 1$ after the operation has completed, $\alpha_i < 1/2$, and so

$$\begin{aligned}\Phi_i &= size_i/2 - num_i \\ &= 1 \,,\end{aligned}$$

giving $\Delta\Phi_i = 1 - size_{i-1}/4$. Therefore, when the ith operation is a deletion that triggers a contraction, its amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Delta\Phi_i \\ &= size_{i-1}/4 + (1 - size_{i-1}/4) \\ &= 1 \,.\end{aligned}$$

Finally, we handle the cases where the load factor fits one case of equation (16.5) before the operation and the other case afterward. We start with deletion, where we have $num_{i-1} = size_{i-1}/2$, so that $\alpha_{i-1} = 1/2$, beforehand, and $num_i = size_i/2 - 1$, so that $\alpha_i < 1/2$ afterward. Because $\alpha_{i-1} = 1/2$, we have $\Phi_{i-1} = 0$, and because $\alpha_i < 1/2$, we have $\Phi_i = size_i/2 - num_i = 1$. Thus we get that $\Delta\Phi_i = 1 - 0 = 1$. Since the ith operation is a deletion that does not cause a contraction, the actual cost c_i equals 1, and the amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i = 1 + 1 = 2$.

Conversely, if the ith operation is an insertion that takes the load factor from below $1/2$ to equaling $1/2$, the change in potential $\Delta\Phi_i$ equals -1. Again, the actual cost c_i is 1, and now the amortized cost $\widehat{c}_i$ is $c_i + \Delta\Phi_i = 1 + (-1) = 0$.

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of n operations on a dynamic table is $O(n)$.

Exercises

16.4-1

Using the potential method, analyze the amortized cost of the first table insertion.

16.4-2

You wish to implement a dynamic, open-address hash table. Why might you consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per

insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

16.4-3

Discuss how to use the accounting method to analyze both the insertion and deletion operations, assuming that the table doubles in size when its load factor exceeds 1 and the table halves in size when its load factor goes below $1/4$.

16.4-4

Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, you contract the table by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function

$$\Phi(T) = |2(T.num - T.size/2)| \ ,$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

Problems

16-1 Binary reflected Gray code

A ***binary Gray code*** represents a sequence of nonnegative integers in binary such that to go from one integer to the next, exactly one bit flips every time. The ***binary reflected Gray code*** represents a sequence of the integers 0 to $2^k - 1$ for some positive integer k according to the following recursive method:

- For $k = 1$, the binary reflected Gray code is $\langle 0, 1 \rangle$.
- For $k \geq 2$, first form the binary reflected Gray code for $k - 1$, giving the 2^{k-1} integers 0 to $2^{k-1} - 1$. Then form the reflection of this sequence, which is just the sequence in reverse. (That is, the jth integer in the sequence becomes the $(2^{k-1} - j - 1)$st integer in the reflection). Next, add 2^{k-1} to each of the 2^{k-1} integers in the reflected sequence. Finally, concatenate the two sequences.

For example, for $k = 2$, first form the binary reflected Gray code $\langle 0, 1 \rangle$ for $k = 1$. Its reflection is the sequence $\langle 1, 0 \rangle$. Adding $2^{k-1} = 2$ to each integer in the reflection gives the sequence $\langle 3, 2 \rangle$. Concatenating the two sequences gives $\langle 0, 1, 3, 2 \rangle$ or, in binary, $\langle 00, 01, 11, 10 \rangle$, so that each integer differs from its predecessor by exactly one bit. For $k = 3$, the reflection of the binary reflected Gray code for $k = 2$ is $\langle 2, 3, 1, 0 \rangle$ and adding $2^{k-1} = 4$ gives $\langle 6, 7, 5, 4 \rangle$. Concatenating produces the sequence $\langle 0, 1, 3, 2, 6, 7, 5, 4 \rangle$, which in binary is $\langle 000, 001, 011, 010, 110, 111, 101, 100 \rangle$. In the binary reflected Gray code, only one bit flips even when wrapping around from the last integer to the first.

a. Index the integers in a binary reflected Gray code from 0 to $2^k - 1$, and consider the ith integer in the binary reflected Gray code. To go from the $(i-1)$st integer to the ith integer in the binary reflected Gray code, exactly one bit flips. Show how to determine which bit flips, given the index i.

b. Assuming that given a bit number j, you can flip bit j of an integer in constant time, show how to compute the entire binary reflected Gray code sequence of 2^k numbers in $\Theta(2^k)$ time.

16-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. You can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that you wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n+1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \ldots, n_0 \rangle$. Maintain k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k-1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i \, 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times, assuming that the only operations are INSERT and SEARCH.

c. Describe how to implement DELETE. Analyze its worst-case and amortized running times, assuming that there can be DELETE, INSERT, and SEARCH operations.

16-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$, which gives the number of keys stored in the subtree rooted at x. Let α be a constant in the range $1/2 \le \alpha < 1$. We say that a given node x is ***α-balanced*** if $x.left.size \le \alpha \cdot x.size$ and $x.right.size \le \alpha \cdot x.size$. The tree as a whole is ***α-balanced*** if every node in the tree is α-balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

a. A $1/2$-balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$-balanced. Your algorithm should run in $\Theta(x.size)$ time, and it can use $O(x.size)$ auxiliary storage.

b. Show that performing a search in an n-node α-balanced binary search tree takes $O(\lg n)$ worst-case time.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that you implement INSERT and DELETE as usual for an n-node binary search tree, except that after every such operation, if any node in the tree is no longer α-balanced, then you "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$-balanced.

We'll analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T, define

$$\Delta(x) = |x.left.size - x.right.size| \ .$$

Define the potential of T as

$$\Phi(T) = c \sum_{x \in T : \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α.

c. Argue that any binary search tree has nonnegative potential and also that a $1/2$-balanced tree has potential 0.

d. Suppose that m units of potential can pay for rebuilding an m-node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α-balanced?

e. Show that inserting a node into or deleting a node from an n-node α-balanced tree costs $O(\lg n)$ amortized time.

16-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform ***structural modifications***: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n+1)$st node causes $\Omega(\lg n)$ color changes. Then describe a legal

red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Although the worst-case number of color changes per operation can be logarithmic, you will prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case.

b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are ***terminating***: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not. (*Hint:* Look at Figures 13.5, 13.6, and 13.7 in Sections 13.3 and 13.4.)

You will first analyze the structural modifications when only insertions are performed. Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T. Assume that one unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T. Argue that $\Phi(T') = \Phi(T) - 1$.

d. We can break the operation of the RB-INSERT procedure into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

Next you will prove that there are $O(m)$ structural modifications when both insertions and deletions occur. Define, for each node x,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red}, \\ 1 & \text{if } x \text{ is black and has no red children}, \\ 0 & \text{if } x \text{ is black and has one red child}, \\ 2 & \text{if } x \text{ is black and has two red children}. \end{cases}$$

Now redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x),$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T.

f. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

g. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

Chapter notes

Aho, Hopcroft, and Ullman [5] used aggregate analysis to determine the running time of operations on a disjoint-set forest. We'll analyze this data structure using the potential method in Chapter 19. Tarjan [430] surveys the accounting and potential methods of amortized analysis and presents several applications. He attributes the accounting method to several authors, including M. R. Brown, R. E. Tarjan, S. Huddleston, and K. Mehlhorn. He attributes the potential method to D. D. Sleator. The term "amortized" is due to D. D. Sleator and R. E. Tarjan.

Potential functions are also useful for proving lower bounds for certain types of problems. For each configuration of the problem, define a potential function that maps the configuration to a real number. Then determine the potential Φ_{init} of the initial configuration, the potential Φ_{final} of the final configuration, and the maximum change in potential $\Delta\Phi_{\text{max}}$ due to any step. The number of steps must therefore be at least $|\Phi_{\text{final}} - \Phi_{\text{init}}| / |\Delta\Phi_{\text{max}}|$. Examples of potential functions to prove lower bounds in I/O complexity appear in works by Cormen, Sundquist, and Wisniewski [105], Floyd [146], and Aggarwal and Vitter [3]. Krumme, Cybenko, and Venkataraman [271] applied potential functions to prove lower bounds on ***gossiping***: communicating a unique item from each vertex in a graph to every other vertex.