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.

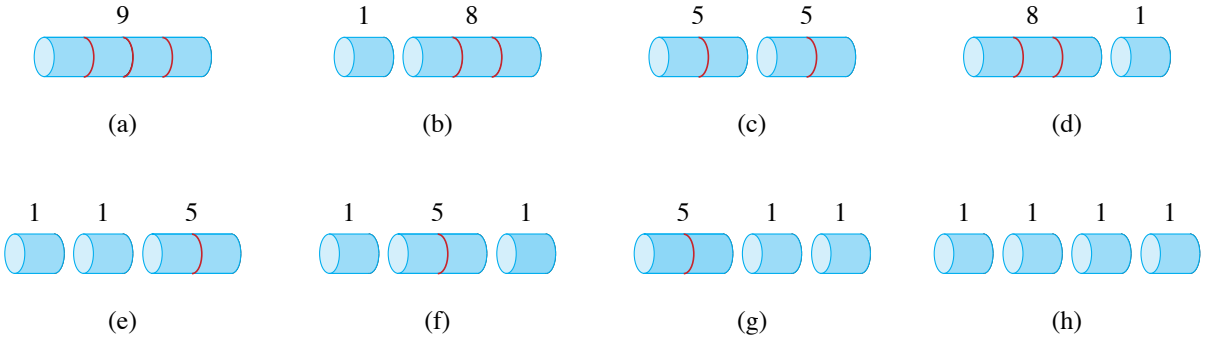


Figure 14.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 14.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

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

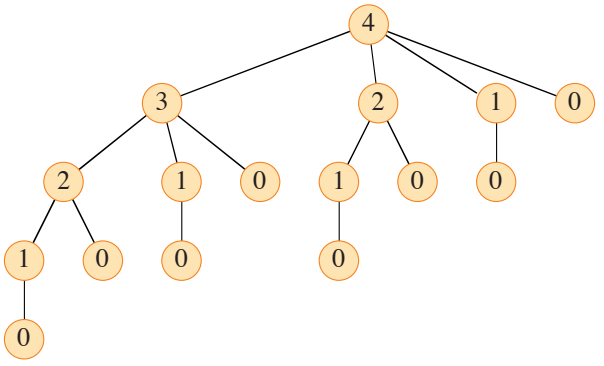


Figure 14.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and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

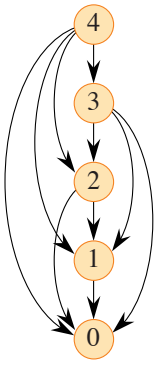


Figure 14.4 The subproblem graph for the rod-cutting problem with $n = 4$. The vertex labels give the sizes of the corresponding subproblems. A directed edge (x, y) indicates that solving subproblem x requires a solution to subproblem y. This graph is a reduced version of the recursion tree of Figure 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:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4))$,
$((A_1A_2)(A_3A_4))$,
$((A_1(A_2A_3))A_4)$,
$(((A_1A_2)A_3)A_4)$.

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 $((A_1A_2)A_3)$ performs $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the 10×5 matrix product A_1A_2, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. Multiplying according to the alternative parenthesization $(A_1(A_2A_3))$ performs $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the 100×50 matrix product A_2A_3, plus another $10 \cdot 100 \cdot 50 = 50{,}000$ scalar multiplications to multiply A_1 by this matrix, for a

total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the ***matrix-chain multiplication problem*** as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications. 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.

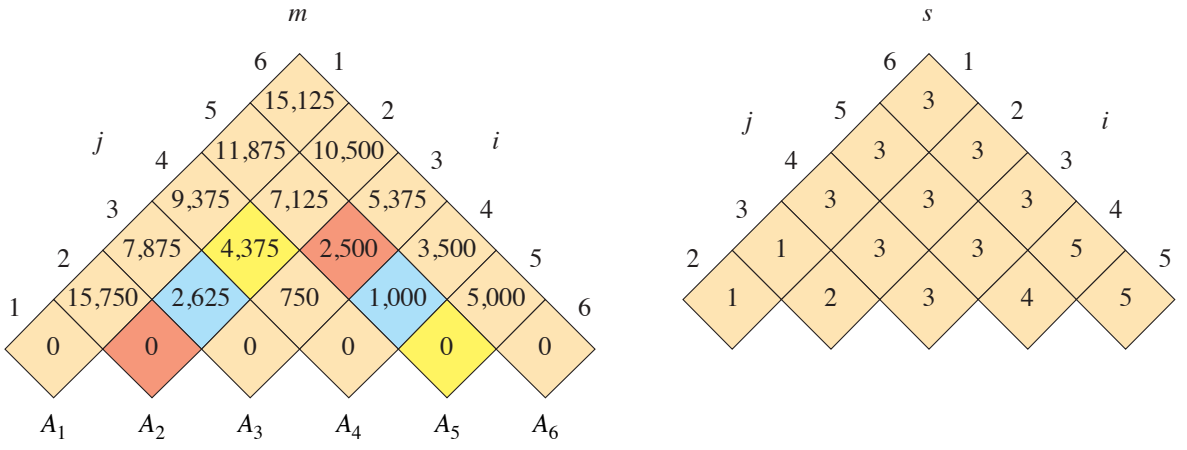


Figure 14.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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.

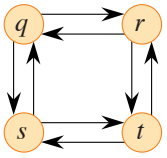


Figure 14.6 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath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

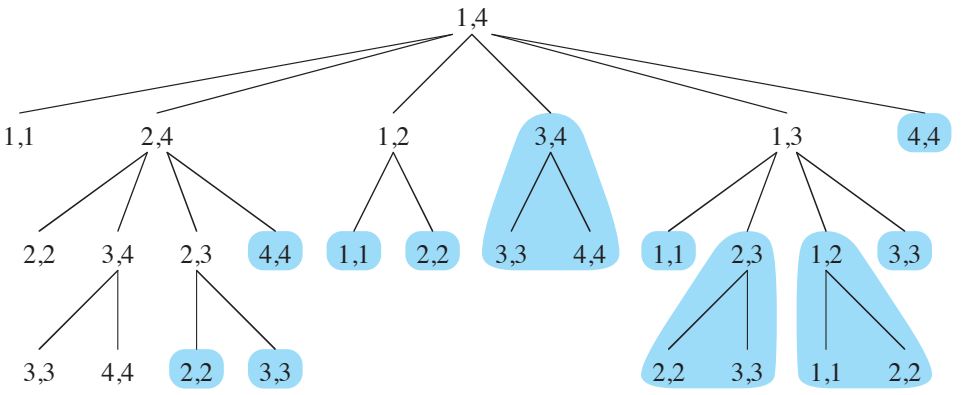


Figure 14.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a subtree shaded blue are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

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

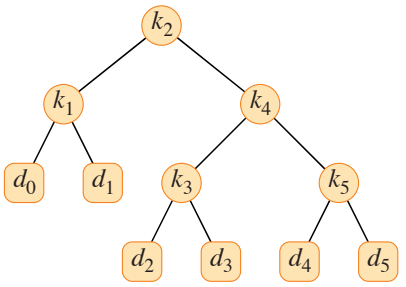
Suppose that you are designing a program to translate text from English to Latvian. For each occurrence of each English word in the text, you need to look up its Latvian equivalent. You can perform these lookup operations by building a binary search tree with n English words as keys and their Latvian 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?

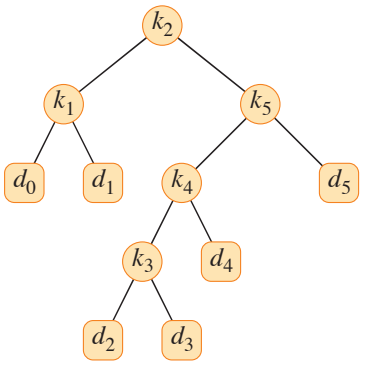
What you need is an ***optimal binary search tree***. Formally, given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys such that $k_1 < k_2 < \cdots < k_n$, build a binary search tree containing them. For each key k_i, you are given the probability p_i that any given search is 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.

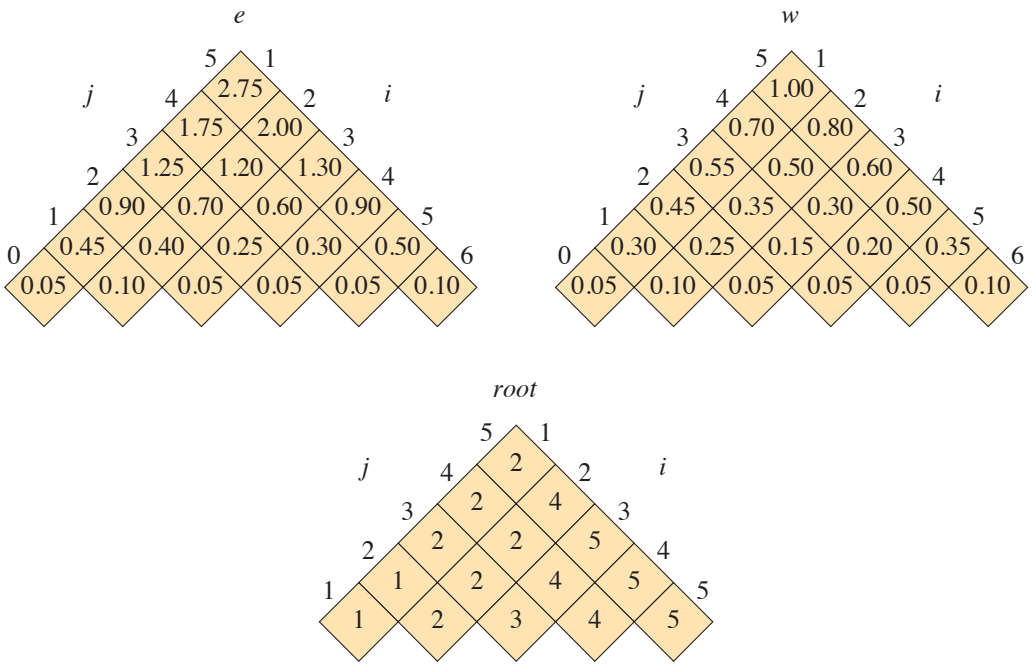


Figure 14.10 The tables $e[i, j]$, $w[i, j]$, and $root[i, j]$ computed by OPTIMAL-BST on the key distribution shown in Figure 14.9. The tables are rotated so that the diagonals run horizontally.

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.

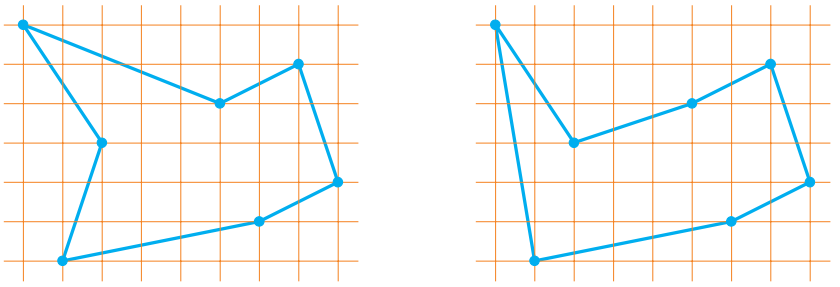


Figure 14.11 Seven points in the plane, shown on a unit grid. **(a)** The shortest closed tour, with length approximately 24.89. This tour is not bitonic. **(b)** The shortest bitonic tour for the same set of points. Its length is approximately 25.58.

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`	al_
replace by `t`	a`lgo`r`ithm`	alt_
delete	a`lgor`i`thm`	alt_
copy	a`lgori`t`hm`	altr_
insert `u`	a`lgori`t`hm`	altru_
insert `i`	a`lgori`t`hm`	altrui_
insert `s`	a`lgori`t`hm`	altruis_
twiddle	a`lgorit`h`m`	altruisti_
insert `c`	a`lgorit`h`m`	altruistic_
kill	`algorithm_`	`altruistic_`

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.