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 $n$th 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 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:

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

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

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

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

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

To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are $10 \times 100$, $100 \times 5$, and $5 \times 50$, respectively. Multiplying according to the parenthesization $((A_1 A_2) A_3)$ performs $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute the $10 \times 5$ matrix product $A_1 A_2$, plus another $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications to multiply this matrix by $A_3$, for a total of 7500 scalar multiplications. Multiplying according to the alternative parenthesization $(A_1 (A_2 A_3))$ performs $100 \cdot 5 \cdot 50 = 25{,}000$ scalar multiplications to compute the $100 \times 50$ 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 $A_1A_2 \cdots A_n$ 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 $k$th and $(k + 1)$st matrices for any $k = 1, 2, \ldots, n - 1$. Thus, we obtain the recurrence

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

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)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                          // chain length 1
    m[i, i] = 0
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 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 $i \le j$, 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.

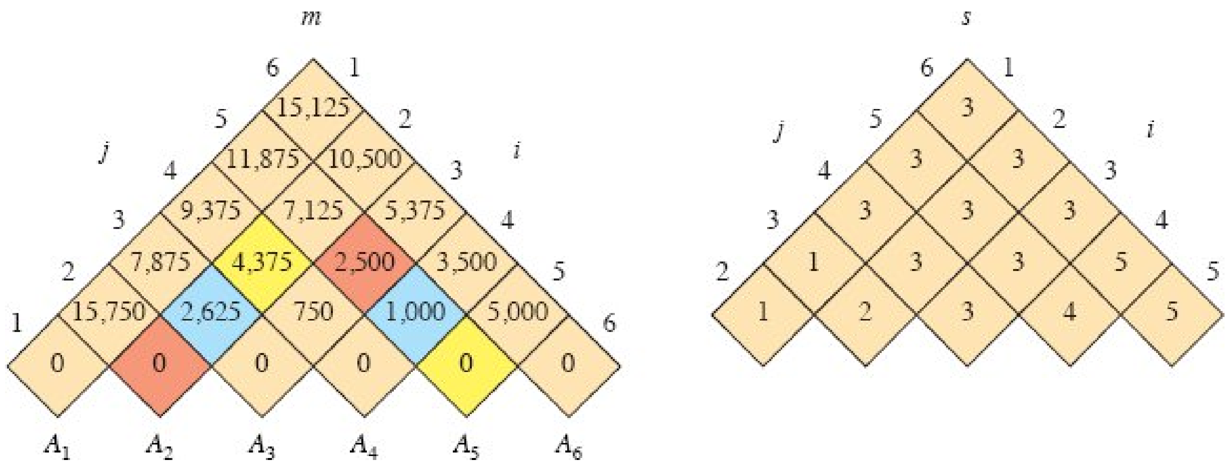


**Figure 14.5** The $m$ and $s$ tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

| 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

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

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

**Exercises**

***14.2-1***

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is ⟨5, 10, 3, 12, 5, 50, 6⟩.

***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