15.1-2
Use equations (15.8) and (15.9) and the substitution method to show that $r_i(j)$, the number of references made to $f_i[j]$ in a recursive algorithm, equals 2^{n-j}.

15.1-3
Using the result of Exercise 15.1-2, show that the total number of references to all $f_i[j]$ values, or $\sum_{i=1}^{2} \sum_{j=1}^{n} r_i(j)$, is exactly $2^{n+1} - 2$.

15.1-4
Together, the tables containing $f_i[j]$ and $l_i[j]$ values contain a total of $4n - 2$ entries. Show how to reduce the space requirements to a total of $2n + 2$ entries, while still computing f^* and still being able to print all the stations on a fastest way through the factory.

15.1-5
Professor Canty conjectures that there might exist some e_i, $a_{i,j}$, and $t_{i,j}$ values for which FASTEST-WAY produces $l_i[j]$ values such that $l_1[j] = 2$ and $l_2[j] = 1$ for some station number j. Assuming that all transfer costs $t_{i,j}$ are nonnegative, show that the professor is wrong.

15.2 Matrix-chain multiplication

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

$$A_1 A_2 \cdots A_n \ . \tag{15.10}$$

We can evaluate the expression (15.10) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. 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. Matrix multiplication is associative, and so all parenthesizations yield the same product. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct ways:

$$\begin{aligned}
&(A_1(A_2(A_3 A_4))) \ , \\
&(A_1((A_2 A_3)A_4)) \ , \\
&((A_1 A_2)(A_3 A_4)) \ , \\
&((A_1(A_2 A_3))A_4) \ , \\
&(((A_1 A_2)A_3)A_4) \ .
\end{aligned}$$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. The standard algorithm is given by the following pseudocode. The attributes *rows* and *columns* are the numbers of rows and columns in a matrix.

```
MATRIX-MULTIPLY($A$, $B$)
1  if $columns[A] \neq rows[B]$
2     then error "incompatible dimensions"
3     else for $i \leftarrow 1$ to $rows[A]$
4              do for $j \leftarrow 1$ to $columns[B]$
5                     do $C[i, j] \leftarrow 0$
6                        for $k \leftarrow 1$ to $columns[A]$
7                            do $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$
8          return $C$
```

We can multiply two matrices A and B only if they are ***compatible***: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications in line 7, which is pqr. In what follows, we shall express costs in terms of the number of scalar multiplications.

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

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

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

Counting the number of parenthesizations

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

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \,, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \,. \end{cases} \tag{15.11}$$

Problem 12-4 asked you to show that the solution to a similar recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n/n^{3/2})$. A simpler exercise (see Exercise 15.2-3) is to show that the solution to the recurrence (15.11) is $\Omega(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search is therefore a poor strategy for determining the optimal parenthesization of a matrix chain.

Step 1: The structure of an optimal parenthesization

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

would produce another parenthesization of $A_i A_{i+1} \cdots A_j$ whose cost was lower than the optimum: a contradiction. A similar observation holds for the parenthesization of the subchain $A_{k+1} A_{k+2} \cdots A_j$ in the optimal parenthesization of $A_i A_{i+1} \cdots A_j$: it must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

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

Step 2: A recursive solution

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

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

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

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

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\ . \end{cases} \quad (15.12)$$

The $m[i,j]$ values give the costs of optimal solutions to subproblems. To help us keep track of how to construct an optimal solution, let us define $s[i,j]$ to be a value of k at which we can split the product $A_i A_{i+1} \cdots A_j$ to obtain 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, it is a simple matter to write a recursive algorithm based on recurrence (15.12) to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \cdots A_n$. As we shall see in Section 15.3, however, this algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.

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

Instead of computing the solution to recurrence (15.12) recursively, we perform the third step of the dynamic-programming paradigm and compute the optimal cost by using a tabular, bottom-up approach. The following pseudocode assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, \ldots, n$. The input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $\mathit{length}[p] = n + 1$. The procedure uses an auxiliary table $m[1 \,..\, n, 1 \,..\, n]$ for storing the $m[i,j]$ costs and an auxiliary table $s[1 \,..\, n, 1 \,..\, n]$ that records which index of k achieved the optimal cost in computing $m[i,j]$. We will use the table s to construct an optimal solution.

In order to correctly implement the bottom-up approach, we must determine which entries of the table are used in computing $m[i,j]$. Equation (15.12) shows that the cost $m[i,j]$ of computing a matrix-chain product of $j - i + 1$ matrices depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices. That is, for $k = i, i+1, \ldots, j-1$, the matrix $A_{i..k}$ is a product of $k - i + 1 < j - i + 1$ matrices and the matrix $A_{k+1..j}$ is a product of $j - k < j - i + 1$ matrices. Thus, the algorithm should fill in the table m in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length.

```
MATRIX-CHAIN-ORDER($p$)
 1  $n \leftarrow length[p] - 1$
 2  for $i \leftarrow 1$ to $n$
 3       do $m[i, i] \leftarrow 0$
 4  for $l \leftarrow 2$ to $n$              $\rhd$ $l$ is the chain length.
 5       do for $i \leftarrow 1$ to $n - l + 1$
 6              do $j \leftarrow i + l - 1$
 7                 $m[i, j] \leftarrow \infty$
 8                 for $k \leftarrow i$ to $j - 1$
 9                     do $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$
10                        if $q < m[i, j]$
11                          then $m[i, j] \leftarrow q$
12                               $s[i, j] \leftarrow k$
13  return $m$ and $s$
```

The algorithm first computes $m[i, i] \leftarrow 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1) in lines 2–3. It then uses recurrence (15.12) to compute $m[i, i + 1]$ for $i = 1, 2, \ldots, n - 1$ (the minimum costs for chains of length $l = 2$) during the first execution of the loop in lines 4–12. The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i, j]$ cost computed in lines 9–12 depends only on table entries $m[i, k]$ and $m[k + 1, j]$ already computed.

Figure 15.3 illustrates this procedure on a chain of $n = 6$ matrices. Since we have defined $m[i, j]$ only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, the minimum cost $m[i, j]$ for multiplying a subchain $A_i A_{i+1} \cdots A_j$ of matrices can be found at the intersection of lines running northeast from A_i and northwest from A_j. Each horizontal row in the table contains the entries for matrix chains of the same length. MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row. An entry $m[i, j]$ is computed using the products $p_{i-1} p_k p_j$ for $k = i, i + 1, \ldots, j - 1$ and all entries southwest and southeast from $m[i, j]$.

A simple inspection of the nested loop structure of MATRIX-CHAIN-ORDER yields a running time of $O(n^3)$ for the algorithm. The loops are nested three deep, and each loop index (l, i, and k) takes on at most $n-1$ values. Exercise 15.2-4 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.

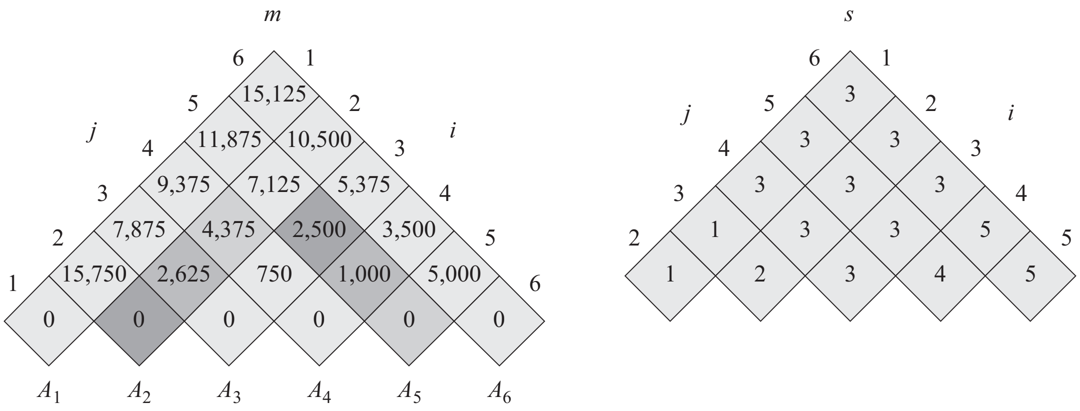


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13000\ , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\ , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11375 \end{cases}$$
$$= 7125\ .$$

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. It is not difficult to construct an optimal solution from the computed information stored in the table $s[1 \mathinner{..} n, 1 \mathinner{..} n]$. Each entry $s[i, j]$ records the value of k such that the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. Thus, we know that the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]} A_{s[1,n]+1..n}$. The earlier matrix multiplications can be computed recursively, since $s[1, s[1, n]]$ determines the last matrix multiplication in computing $A_{1..s[1,n]}$, and $s[s[1,n]+1,n]$ determines the last matrix multiplication in computing $A_{s[1,n]+1..n}$. The following recursive procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j. The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $\langle A_1, A_2, \ldots, A_n \rangle$.

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

In the example of Figure 15.3, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Exercises

15.2-1

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

15.2-2

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

15.2-3

Use the substitution method to show that the solution to the recurrence (15.11) is $\Omega(2^n)$.

15.2-4

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

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

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

15.2-5
Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

15.3 Elements of dynamic programming

Although we have just worked through two examples of the dynamic-programming method, you might still be wondering just when the method applies. From an engineering perspective, when should we look for a dynamic-programming solution to a problem? In this section, we examine the two key ingredients that an optimization problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems. We also look at a variant method, called memoization,[1] for taking advantage of the overlapping-subproblems property.

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. Whenever a problem exhibits optimal substructure, it is a good clue that dynamic programming might apply. (It also might mean that a greedy strategy applies, however. See Chapter 16.) In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.

We discovered optimal substructure in both of the problems we have examined in this chapter so far. In Section 15.1, we observed that the fastest way through station j of either line contained within it the fastest way through station $j-1$ on one line. In Section 15.2, we observed that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ 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] This is not a misspelling. The word really is *memoization*, not *memorization*. *Memoization* comes from *memo*, since the technique consists of recording a value so that we can look it up later.

1. You show that a solution to the problem consists of making a choice, such as choosing a preceding assembly-line station 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 the 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 subproblem solution 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 there is more than one subproblem, they are typically so similar that the cut-and-paste argument for one can be modified for the others with little effort.

To characterize the space of subproblems, a good rule of thumb is to try to keep the space as simple as possible, and then to expand it as necessary. For example, the space of subproblems that we considered for assembly-line scheduling was the fastest way from entry into the factory through stations $S_{1,j}$ and $S_{2,j}$. This subproblem space worked well, and there was no need to try a more general space of subproblems.

Conversely, suppose that we had tried to constrain our 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 $1 \leq k \leq j$. Unless we could guarantee that k always equals $j - 1$, we would find that we had subproblems of the form $A_1 A_2 \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$, and that the latter subproblem is not of the form $A_1 A_2 \cdots A_j$. For this problem, it was necessary to allow our subproblems to vary at "both ends," that is, to allow both i and j to vary in the subproblem $A_i A_{i+1} \cdots A_j$.

Optimal substructure varies across problem domains in two ways:

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

In assembly-line scheduling, an optimal solution uses just one subproblem, but we must consider two choices in order to determine an optimal solution. To find the fastest way through station $S_{i,j}$, we use *either* the fastest way through $S_{1,j-1}$ *or* the fastest way through $S_{2,j-1}$; whichever we use represents the one subproblem that we must optimally solve. Matrix-chain multiplication for the subchain $A_i A_{i+1} \cdots A_j$ serves as an example with two subproblems and $j - i$ choices. For a given matrix A_k at which we split the product, we have two subproblems—parenthesizing $A_i A_{i+1} \cdots A_k$ and parenthesizing $A_{k+1} A_{k+2} \cdots A_j$—and we must 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 we look at for each subproblem. In assembly-line scheduling, we had $\Theta(n)$ subproblems overall, and only two choices to examine for each, yielding a $\Theta(n)$ running time. For matrix-chain multiplication, there were $\Theta(n^2)$ subproblems overall, and in each we had at most $n - 1$ choices, giving an $O(n^3)$ running time.

Dynamic programming uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among subproblems as to which we 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 assembly-line scheduling, for example, first we solved the subproblems of finding the fastest way through stations $S_{1,j-1}$ and $S_{2,j-1}$, and then we chose one of these stations as the one preceding station $S_{i,j}$. The cost attributable to the choice itself depends on whether we switch lines between stations $j - 1$ and j; this cost is $a_{i,j}$ if we stay on the same line, and it is $t_{i',j-1} + a_{i,j}$, where $i' \neq i$, if we switch. 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 $p_{i-1} p_k p_j$.

In Chapter 16, we shall examine "greedy algorithms," which have many similarities to dynamic programming. In particular, problems to which greedy algorithms apply have optimal substructure. One salient difference between greedy algorithms and dynamic programming is that in greedy algorithms, we use optimal substructure in a top-down fashion. Instead of first finding optimal solutions to subproblems and then making a choice, greedy algorithms first make a choice—the choice that looks best at the time—and then solve a resulting subproblem.

Subtleties

One should be careful not to assume that optimal substructure applies when it does not. Consider the following two problems in which we are given a directed graph $G = (V, E)$ and vertices $u, v \in V$.

Unweighted shortest path:[2] 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. We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.

The unweighted shortest-path problem exhibits optimal substructure, as follows. 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.) Thus we can decompose the path $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$. Clearly, the number of edges in p is equal to the sum of the number of edges in p_1 and 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? We 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 $u \overset{p'_1}{\leadsto} w \overset{p_2}{\leadsto} v$ with fewer edges than p, thus contradicting p's optimality. Symmetrically, p_2 must be a shortest path from w to v. Thus, we can find a shortest path from u to v by considering all intermediate vertices w, finding a shortest path from u to w and a shortest path from w to v, and choosing an intermediate vertex w that yields the overall shortest path. In Section 25.2, we use a variant of this observation of optimal substructure to find a shortest path between every pair of vertices on a weighted, directed graph.

It is tempting 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 $u \overset{p}{\leadsto} v$ into subpaths $u \overset{p_1}{\leadsto} w \overset{p_2}{\leadsto} v$, 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 15.4 gives an example. Consider the path $q \rightarrow r \rightarrow t$, which is a longest simple path from q to t. Is $q \rightarrow r$ a longest simple path from q to r? No, for the path $q \rightarrow s \rightarrow t \rightarrow r$ is a simple path that is longer. Is $r \rightarrow t$ a longest simple path from r to t? No again, for the path $r \rightarrow q \rightarrow s \rightarrow t$ is a simple path that is longer.

This example shows that for longest simple paths, not only is optimal substructure lacking, but we cannot necessarily assemble a "legal" solution to the problem from solutions to subproblems. If we combine the longest simple paths $q \rightarrow s \rightarrow t \rightarrow r$ and $r \rightarrow q \rightarrow s \rightarrow t$, we get the path $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$,

[2] We use the term "unweighted" to distinguish this problem from that of finding shortest paths with weighted edges, which we shall see in Chapters 24 and 25. We can use the breadth-first search technique of Chapter 22 to solve the unweighted problem.

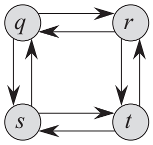


Figure 15.4 A directed graph showing that the problem of finding a longest simple path in an unweighted directed graph does not have optimal substructure. The path $q \to r \to t$ is a longest simple path from q to t, but the subpath $q \to r$ is not a longest simple path from q to r, nor is the subpath $r \to t$ a longest simple path from r to t.

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 it is unlikely that it can be solved in polynomial time.

What is it about the substructure of a longest simple path that is so different from that of a shortest path? Although two subproblems are used in a solution to a problem for both longest and shortest paths, 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 15.4, 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 choose the path $q \to s \to t \to r$, and so we have also used the vertices s and t. We can no longer use these vertices in the second subproblem, since the combination of the two solutions to subproblems would yield a path that is not simple. If we cannot use vertex t in the second problem, then we cannot solve it all, since t is required to be on the path that we find, and it is not the vertex at which we are "splicing" together the subproblem solutions (that vertex being r). Our use of vertices s and t in one subproblem solution prevents them from being used in the other subproblem solution. We must use at least one of them to solve the other subproblem, however, and we must use both of them to solve it optimally. Thus we say that these subproblems are not independent. Looked at another way, our use of resources in solving one subproblem (those resources being vertices) has rendered 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 $u \overset{p_1}{\leadsto} w$ and *any* shortest path $w \overset{p_2}{\leadsto} v$ 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 $u \overset{p_{ux}}{\leadsto} x \leadsto w$ and p_2 as $w \leadsto x \overset{p_{xv}}{\leadsto} v$. 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 $u \overset{p_{ux}}{\leadsto} x \overset{p_{xv}}{\leadsto} v$ from u to v. This path has 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.

Both problems examined in Sections 15.1 and 15.2 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 assembly-line scheduling, to determine the fastest way through station $S_{i,j}$, we look at the fastest ways through stations $S_{1,j-1}$ and $S_{2,j-1}$. Because our solution to the fastest way through station $S_{i,j}$ will include just one of these subproblem solutions, that subproblem is automatically independent of all others used in the solution.

Overlapping subproblems

The second ingredient that an optimization problem must have for dynamic programming to be applicable 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 over and over again, we say that the optimization problem has ***overlapping subproblems***.[3] In contrast, a problem for which a divide-and-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 15.1, we briefly examined how a recursive solution to assembly-line scheduling makes 2^{n-j} references to $f_i[j]$ for $j = 1, 2, \ldots, n$. Our tabular solution takes an exponential-time recursive algorithm down to linear time.

To illustrate the overlapping-subproblems property in greater detail, let us reexamine the matrix-chain multiplication problem. Referring back to Figure 15.3,

[3] 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.

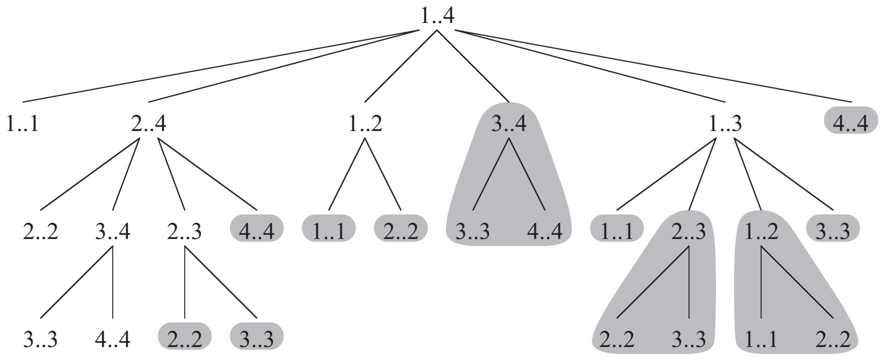


Figure 15.5 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN$(p, 1, 4)$. Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN$(p, 1, 4)$.

observe that MATRIX-CHAIN-ORDER repeatedly looks up the solution to subproblems in lower rows when solving subproblems in higher rows. For example, entry $m[3, 4]$ is referenced 4 times: during the computations of $m[2, 4]$, $m[1, 4]$, $m[3, 5]$, and $m[3, 6]$. If $m[3, 4]$ were recomputed each time, rather than just being looked up, the increase in running time would be dramatic. To see this, consider the following (inefficient) recursive procedure that 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 (15.12).

```
RECURSIVE-MATRIX-CHAIN$(p, i, j)$
if $i = j$
   then return 0
$m[i, j] \leftarrow \infty$
for $k \leftarrow i$ to $j - 1$
     do $q \leftarrow$ RECURSIVE-MATRIX-CHAIN$(p, i, k)$
                 $+$ RECURSIVE-MATRIX-CHAIN$(p, k + 1, j)$
                 $+ p_{i-1} p_k p_j$
        if $q < m[i, j]$
           then $m[i, j] \leftarrow q$
return $m[i, j]$
```

Figure 15.5 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, we can show that the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. Let $T(n)$ denote the time taken by RECURSIVE-

MATRIX-CHAIN to compute an optimal parenthesization of a chain of n matrices. If we assume that the execution of lines 1–2 and of lines 6–7 each take at least unit time, then inspection of the procedure yields the recurrence

$$\begin{aligned} T(1) &\ge 1 , \\ T(n) &\ge 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \qquad \text{for } n > 1 . \end{aligned}$$

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 $n-1$ 1's in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \ge 2 \sum_{i=1}^{n-1} T(i) + n . \qquad (15.13)$$

We shall prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we shall show that $T(n) \ge 2^{n-1}$ for all $n \ge 1$. The basis is easy, since $T(1) \ge 1 = 2^0$. Inductively, for $n \ge 2$ we have

$$\begin{aligned} T(n) &\ge 2 \sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2 \sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= (2^n - 2) + n \\ &\ge 2^{n-1} , \end{aligned}$$

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 with the bottom-up, dynamic-programming algorithm. The latter is more efficient because it takes advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ different subproblems, and the dynamic-programming algorithm solves each exactly once. The recursive algorithm, on the other hand, must repeatedly resolve each subproblem each 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 different subproblems is small, it is a good idea to see if dynamic programming can be made to work.

Reconstructing an optimal solution

As a practical matter, we often store which choice we made in each subproblem in a table so that we do not have to reconstruct this information from the costs that we stored. In assembly-line scheduling, we stored in $l_i[j]$ the station preceding $S_{i,j}$ in a fastest way through $S_{i,j}$. Alternatively, having filled in the entire $f_i[j]$ table, we could determine which station precedes $S_{1,j}$ in a fastest way through $S_{i,j}$ with a little extra work. If $f_1[j] = f_1[j-1] + a_{1,j}$, then station $S_{1,j-1}$ precedes $S_{1,j}$ in a fastest way through $S_{1,j}$. Otherwise, it must be the case that $f_1[j] = f_2[j-1] + t_{2,j-1} + a_{1,j}$, and so $S_{2,j-1}$ precedes $S_{1,j}$. For assembly-line scheduling, reconstructing the predecessor stations takes only $O(1)$ time per station, even without the $l_i[j]$ table.

For matrix-chain multiplication, however, the table $s[i, j]$ saves us a significant amount of work when reconstructing an optimal solution. Suppose that we did not maintain the $s[i, j]$ table, having filled in only the table $m[i, j]$ containing optimal subproblem costs. There are $j-i$ choices in 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 we chose for a solution to a given problem. By storing in $s[i, j]$ the index of the matrix at which we split the product $A_i A_{i+1} \cdots A_j$, we can reconstruct each choice in $O(1)$ time.

Memoization

There is a variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy. The idea is to ***memoize*** the natural, but inefficient, recursive algorithm. As in ordinary dynamic programming, we 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 during the execution of the recursive algorithm, its solution is computed and then stored in the table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned.[4]

Here is a memoized version of RECURSIVE-MATRIX-CHAIN:

[4] This approach presupposes that the set of all possible subproblem parameters is known and that the relation between table positions and subproblems is established. Another approach is to memoize by using hashing with the subproblem parameters as keys.

```
Memoized-Matrix-Chain($p$)
1  $n \leftarrow length[p] - 1$
2  for $i \leftarrow 1$ to $n$
3       do for $j \leftarrow i$ to $n$
4              do $m[i, j] \leftarrow \infty$
5  return Lookup-Chain($p, 1, n$)
```

```
Lookup-Chain($p, i, j$)
1  if $m[i, j] < \infty$
2     then return $m[i, j]$
3  if $i = j$
4     then $m[i, j] \leftarrow 0$
5     else for $k \leftarrow i$ to $j - 1$
6              do $q \leftarrow$ Lookup-Chain($p, i, k$)
                        + Lookup-Chain($p, k + 1, j$) + $p_{i-1} p_k p_j$
7                 if $q < m[i, j]$
8                    then $m[i, j] \leftarrow q$
9  return $m[i, j]$
```

Memoized-Matrix-Chain, like Matrix-Chain-Order, 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. When the call Lookup-Chain(p, i, j) is executed, if $m[i, j] < \infty$ in line 1, the procedure simply returns the previously computed cost $m[i, j]$ (line 2). Otherwise, the cost is computed as in Recursive-Matrix-Chain, stored in $m[i, j]$, and returned. (The value ∞ is convenient to use for an unfilled table entry since it is the value used to initialize $m[i, j]$ in line 3 of Recursive-Matrix-Chain.) Thus, Lookup-Chain(p, i, j) always returns the value of $m[i, j]$, but it computes it only if this is the first time that Lookup-Chain has been called with the parameters i and j.

Figure 15.5 illustrates how Memoized-Matrix-Chain saves time compared to Recursive-Matrix-Chain. Shaded subtrees represent values that are looked up rather than computed.

Like the dynamic-programming algorithm Matrix-Chain-Order, the procedure Memoized-Matrix-Chain runs in $O(n^3)$ time. Each of $\Theta(n^2)$ table entries is initialized once in line 4 of Memoized-Matrix-Chain. We can categorize the calls of Lookup-Chain into two types:

1. calls in which $m[i, j] = \infty$, so that lines 3–9 are executed, 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.

In summary, the matrix-chain multiplication problem can be solved by either a top-down, memoized algorithm or a bottom-up, dynamic-programming algorithm in $O(n^3)$ time. Both methods take advantage of the overlapping-subproblems property. There are only $\Theta(n^2)$ different subproblems in total, and either of these methods computes the solution to each subproblem 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 a top-down memoized algorithm by a constant factor, because there is no overhead for recursion and less overhead for maintaining the table. Moreover, there are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce time or space requirements even further. Alternatively, if some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercises

15.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

15.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization is ineffective in speeding up a good divide-and-conquer algorithm such as MERGE-SORT.

15.3-3

Consider a variant of the matrix-chain multiplication problem in which 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?

15.3-4
Describe how assembly-line scheduling has overlapping subproblems.

15.3-5
As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that it is not always necessary to solve all the subproblems in order to find an optimal solution. She suggests that an optimal solution to the matrix-chain multiplication problem can be found by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) *before* solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

15.4 Longest common subsequence

In biological applications, we often want to compare the DNA of two (or more) different organisms. A strand of DNA consists of a string of molecules called ***bases***, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by their initial letters, a strand of DNA can be expressed as a string over the finite set $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. (See Appendix C for a definition of a string.) For example, the DNA of one organism may be $S_1 =$ ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, while the DNA of another organism may be $S_2 =$ GTCGTTCGGAATGCCGTTGCTCTGTAAA. One goal of comparing two strands of DNA is to determine how “similar” the two strands are, as some measure of how closely related the two organisms are. Similarity can be and is defined in many different ways. For example, we can say that two DNA strands are similar if one is a substring of the other. (Chapter 32 explores algorithms to solve this problem.) In our example, neither S_1 nor S_2 is a substring of the other. Alternatively, we could say that two strands are similar if the number of changes needed to turn one into the other is small. (Problem 15-3 looks at this notion.) Yet another way to measure the similarity of strands S_1 and S_2 is by finding a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively. The longer the strand S_3 we can find, the more similar S_1 and S_2 are. In our example, the longest strand S_3 is GTCGTCGGAAGCCGGCCGAA.

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

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

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

Step 1: Characterizing a longest common subsequence

A brute-force approach to solving the LCS problem is to enumerate all subsequences of X and check each subsequence to see if it is also a subsequence of Y, keeping track of the longest subsequence found. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. There are 2^m subsequences of X, so this approach requires exponential time, making it impractical for long sequences.

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

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

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

Proof (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$.

Now, the prefix Z_{k-1} is a length-$(k-1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there is 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 characterization of Theorem 15.1 shows that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution also has the overlapping-subproblems property, as we shall see in a moment.

Step 2: A recursive solution

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

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

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

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \,. \end{cases} \tag{15.14}$$

Observe that in this recursive formulation, a condition in the problem restricts which subproblems we may consider. When $x_i = y_j$, we can and should consider the subproblem of finding the LCS of X_{i-1} and Y_{j-1}. Otherwise, we instead consider the two subproblems of finding the 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 assembly-line scheduling and matrix-chain multiplication—no subproblems were ruled out due to conditions in the problem. Finding the LCS is not the only dynamic-programming algorithm that rules out subproblems based on conditions in the problem. For example, the edit-distance problem (see Problem 15-3) has this characteristic.

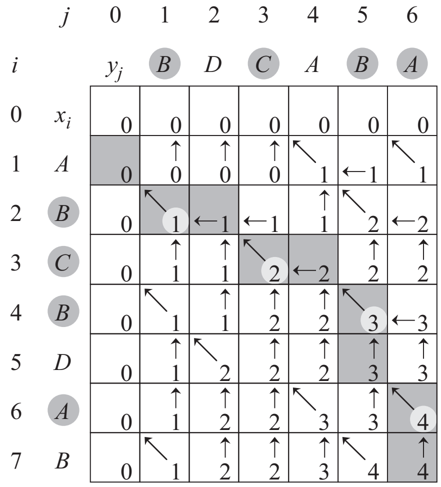
Step 3: Computing the length of an LCS

Based on equation (15.14), we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. Since there are only $\Theta(mn)$ distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

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

```
LCS-LENGTH($X, Y$)
for $i \leftarrow 1$ to $m$
```

```
LCS-LENGTH($X, Y$)
$m \leftarrow length[X]$
$n \leftarrow length[Y]$
for $i \leftarrow 1$ to $m$
     do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $n$
     do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ to $m$
     do for $j \leftarrow 1$ to $n$
            do if $x_i = y_j$
                  then $c[i, j] \leftarrow c[i-1, j-1] + 1$
                       $b[i, j] \leftarrow$ "↖"
                  else if $c[i-1, j] \geq c[i, j-1]$
                          then $c[i, j] \leftarrow c[i-1, j]$
                               $b[i, j] \leftarrow$ "↑"
                          else $c[i, j] \leftarrow c[i, j-1]$
                               $b[i, j] \leftarrow$ "←"
return $c$ and $b$
```



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

Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

Figure 15.6 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 $O(mn)$, since each table entry takes $O(1)$ time to compute.

Step 4: Constructing an LCS

The b table returned by LCS-LENGTH can be used to quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. We simply begin at $b[m, n]$ and trace through the table following the arrows. Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS. The elements of the LCS are encountered in reverse order by this method. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial invocation is PRINT-LCS$(b, X, \mathit{length}[X], \mathit{length}[Y])$.

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

For the b table in Figure 15.6, this procedure prints "$BCBA$." The procedure takes time $O(m + n)$, since at least one of i and j is decremented in each stage of the recursion.

Improving the code

Once you have developed an algorithm, you will often find that you can improve on the time or space it uses. This is especially true of straightforward dynamic-programming algorithms. 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.

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

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

Exercises

15.4-1

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

15.4-2

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

15.4-3

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

15.4-4

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

15.4-5

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

15.4-6 ⋆

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

15.5 Optimal binary search trees

Suppose that we are designing a program to translate text from English to French. For each occurrence of each English word in the text, we need to look up its French equivalent. One way to perform these lookup operations is to build a binary search tree with n English words as keys and French equivalents as satellite data. Because we will search the tree for each individual word in the text, we want the total time spent searching to be as low as possible. We could ensure an $O(\lg n)$ search time per occurrence by using a red-black tree or any other balanced binary search tree. Words appear with different frequencies, however, and it may be the case that a frequently used word such as “the” appears far from the root while a rarely used word such as “mycophagist” 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 is one plus the depth of the node containing the key. We