

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  $i = 1, 2, \dots, n$ . 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  $\{A, C, G, T\}$ . (See Section C.1 for the definition of a string.) For example, the DNA of one organism may be  $S_1 = \text{ACCGGTCGAGTGC GCGGAAGCCGGCCGAA}$ , and the DNA of another organism may be  $S_2 = \text{GTCGTT CGGAATGCCGTTGCTCTGTAAA}$ . 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  $X = \langle x_1, x_2, \dots, x_m \rangle$ , another sequence  $Z = \langle z_1, z_2, \dots, z_k \rangle$  is a **subsequence** of  $X$  if there exists a strictly increasing sequence  $\langle i_1, i_2, \dots, i_k \rangle$  of indices of  $X$  such that for all  $j = 1, 2, \dots, k$ , 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  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$ , 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  $\{1, 2, \dots, m\}$  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  $X = \langle x_1, x_2, \dots, x_m \rangle$ , we define the  $i$ th **prefix** of  $X$ , for  $i = 0, 1, \dots, m$ , as  $X_i = \langle x_1, x_2, \dots, x_i \rangle$ . 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  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  be sequences, and let  $Z = \langle z_1, z_2, \dots, z_k \rangle$  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  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$ . 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

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

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  $0 \leq i \leq m$  and  $0 \leq j \leq n$ ), dynamic programming can compute the solutions bottom up.

The procedure **LCS-LENGTH** on the next page takes two sequences  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  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              $b[i, j] = "\nwarrow"$ 
11         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
12              $c[i, j] = c[i - 1, j]$ 
13              $b[i, j] = "\uparrow"$ 
14         else  $c[i, j] = c[i, j - 1]$ 
15              $b[i, j] = "\leftarrow"$ 
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  $b[i, j] == "\nwarrow"$ 
4      PRINT-LCS( $b, X, i - 1, j - 1$ )
5      print  $x_i$        // same as  $y_j$ 
6  elseif  $b[i, j] == "\uparrow"$ 
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  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$ . Begin at  $b[m, n]$  and trace through the table by following the arrows. Each “ $\nwarrow$ ” 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.

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

**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  $\text{PRINT-LCS}(b, X, m, n)$ . 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  $\text{PRINT-LCS}$ . (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  $X = \langle x_1, x_2, \dots, x_m \rangle$  and  $Y = \langle y_1, y_2, \dots, y_n \rangle$  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.)