
15.4 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, guanine, cytosine, and thymine. Representing each of these bases by its initial letter, we can express a strand of DNA as a string over the finite set $\{A, C, G, T\}$. (See Appendix C 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 $S_2 = \text{GTCGTTCCGAATGCCGTTGCTCTGTAAA}$. One reason to compare 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 15-5 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 $\text{GTCGTCGGAAGCCGGCCGAA}$.

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 $X = \langle x_1, x_2, \dots, x_m \rangle$, 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 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**, we are given two sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ and wish 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

In a brute-force approach to solving the LCS problem, we would 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 we 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 shall see, the natural classes of subproblems correspond to pairs of “prefixes” of the two input sequences. To be precise, given a sequence $X = \langle x_1, x_2, \dots, x_m \rangle$, 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 15.1 (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 $x_m \neq y_n$, 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 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 15.1 characterizes longest common subsequences tells us 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 recur-

sive solution also has the overlapping-subproblems property, as we shall see in a moment.

Step 2: A recursive solution

Theorem 15.1 implies that we 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 $x_m = y_n$, 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 LCSs 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 appear 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 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, 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, and so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j - 1], c[i - 1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases} \quad (15.9)$$

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 an LCS of X_{i-1} and Y_{j-1} . Otherwise, we 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 ruled out no 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 15-5) has this characteristic.

Step 3: Computing the length of an LCS

Based on equation (15.9), we could easily 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, however, we can use dynamic programming to compute the solutions bottom up.

Procedure `LCS-LENGTH` 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. 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 us construct 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)

```

1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5       $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7       $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9      for  $j = 1$  to  $n$ 
10         if  $x_i == y_j$ 
11              $c[i, j] = c[i - 1, j - 1] + 1$ 
12              $b[i, j] = "\nwarrow"$ 
13         elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14              $c[i, j] = c[i - 1, j]$ 
15              $b[i, j] = "\uparrow"$ 
16         else  $c[i, j] = c[i, j - 1]$ 
17              $b[i, j] = "\leftarrow"$ 
18  return  $c$  and  $b$ 

```

Figure 15.8 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.

Step 4: Constructing an LCS

The b table returned by `LCS-LENGTH` enables us to quickly construct an LCS of $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$. We simply begin at $b[m, n]$ and trace through the table by following the arrows. Whenever we encounter a “ \nwarrow ” in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that `LCS-LENGTH`

		j	0	1	2	3	4	5	6
i	y_j		B	D	C	A	B	A	
	x_i		0	0	0	0	0	0	0
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 15.8 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; the sequence is shaded. Each “↖” on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

found. With this method, we encounter the elements of this LCS in reverse order. The following recursive procedure prints out an LCS of X and Y in the proper, forward order. The initial call is PRINT-LCS($b, X, X.length, Y.length$).

```

PRINT-LCS( $b, X, i, j$ )
1  if  $i == 0$  or  $j == 0$ 
2    return
3  if  $b[i, j] == \text{“}\swarrow\text{”}$ 
4    PRINT-LCS( $b, X, i - 1, j - 1$ )
5    print  $x_i$ 
6  elseif  $b[i, j] == \text{“}\uparrow\text{”}$ 
7    PRINT-LCS( $b, X, i - 1, j$ )
8  else PRINT-LCS( $b, X, i, j - 1$ )

```

For the b table in Figure 15.8, this procedure prints $BCBA$. The procedure takes time $O(m + n)$, 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, 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, as Exercise 15.4-4 asks you to show, we can use only slightly more than the space for one row of c to compute the length of an LCS.) 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.

Reference :

Text Book-T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein : Introduction to Algorithms, Third Edition, The MIT Press Cambridge, Massachusetts London, England.