Solving “Longest common subsequence” problem using Dynamic programming

- 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}.**

- For example, the DNA of one organism may be,

S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and the DNA of another organism may be S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.

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.

We formalize this last notion of similarity as the longest-common-subsequence problem. A subsequence of a given sequence is just the given sequence with 0 or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a *subsequence* of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$.

LCS contd..

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

In the *longest-common-subsequence problem*, the input is two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, and the goal is to find a maximum-length common subsequence of X and Y. This section shows how to efficiently solve the LCS problem using dynamic programming.

Step 1: Characterizing a longest common subsequence

You can solve the LCS problem with a brute-force approach: enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence you find. Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of X. Because X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences.

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

Theorem 14.1 (Optimal substructure of an LCS)

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

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

Step 2: A recursive solution

Theorem 14.1 implies that you should examine either one or two subproblems when finding an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. If $x_m = y_n$, you need to find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then you have to solve two subproblems: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.

Whichever of these two LCSs is longer is an LCS of X and Y. Because these cases exhaust all possibilities, one of the optimal subproblem solutions must appear within an LCS of X and Y.

The LCS problem has the overlapping-subproblems property. Here's how. To find an LCS of X and Y, you might need to find the LCSs of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding an LCS of X_{m-1} and Y_{n-1}. Many other subproblems share subsubproblems.

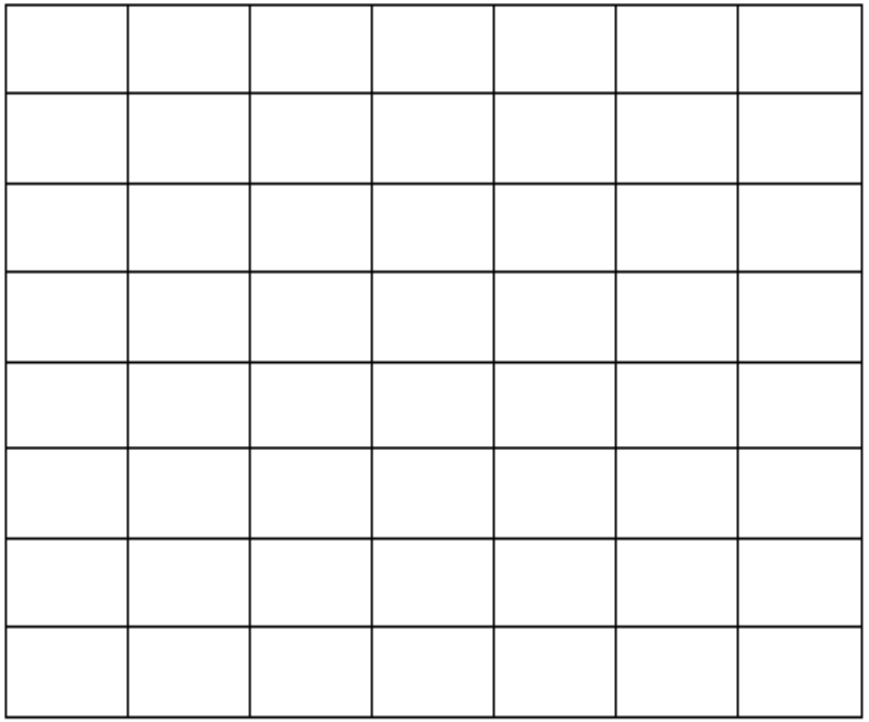
As in the matrix-chain multiplication problem, solving the LCS problem recursively involves establishing a recurrence for the value of an optimal solution. Let's define $c[i, j]$ to be the length of an LCS of the sequences X_i and Y_j. If either $i = 0$ or $j = 0$, one of the sequences has length 0,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

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

Step 3: Computing the length of an LCS

$X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$.

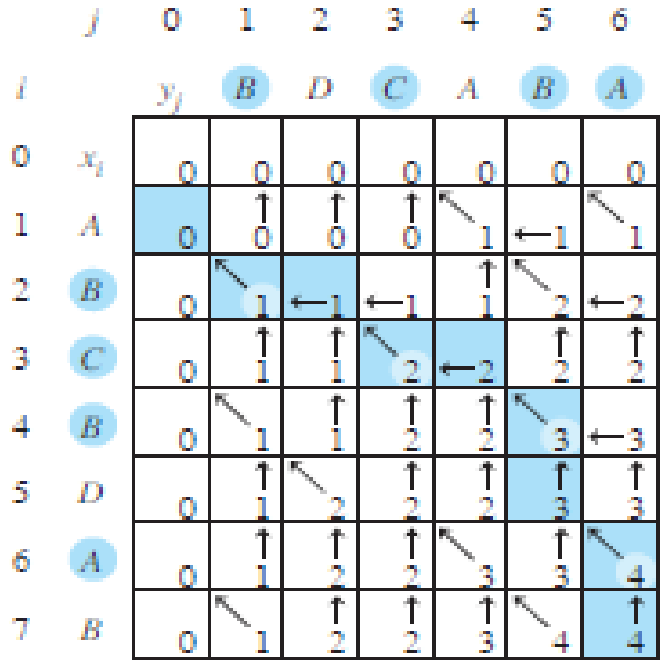
```
LCS-LENGTH(X, Y, m, n)
 1  let b[1:m, 1:n] and c[0:m, 0:n] be new tables
 2  for i = 1 to m
 3      c[i, 0] = 0
 4  for j = 0 to n
 5      c[0, j] = 0
 6  for i = 1 to m          // compute table entries in row-major order
 7      for j = 1 to n
 8          if x_i == y_j
 9              c[i, j] = c[i − 1, j − 1] + 1
10              b[i, j] = "↖"
11          elseif c[i − 1, j] ≥ c[i, j − 1]
12              c[i, j] = c[i − 1, j]
13              b[i, j] = "↑"
14          else c[i, j] = c[i, j − 1]
15              b[i, j] = "←"
16  return c and b
```



```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return            // the LCS has length 0
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i         // same as y_j
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Step 4: Constructing an LCS

With the b table returned by LCS-LENGTH, you can quickly construct an LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Begin at $b[m, n]$ and trace through the table by following the arrows. Each "↖" encountered in an entry $b[i, j]$ implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH found. This method gives you the elements of this LCS in reverse order. The recursive procedure PRINT-LCS prints out an LCS of X and Y in the proper, forward order.



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

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return                  // the LCS has length 0
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i               // same as y_j
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

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 procedure takes O(m+n) time, since it decrements at least one of i and j in each recursi call.