Longest common subsequence(LCS):

Problem Statement:

Given two sequences, $X = < x_1, x_2 \ldots .x_m >$ and $Y = < y_1, y_2 \ldots .y_n >$. Find a subsequence common to both whose length is longest. A subsequence doesn't have to be consecutive, but it has to be in order.

Brute-force algorithm:

For every subsequence of *X*, check whether it.s a subsequence of *Y* .
Time: $\Theta(n*2^m)$.
• 2^m subsequences of *X* to check.
• Each subsequence takes $\Theta(n)$ time to check: scan *Y* for first letter, from there scan for second, and so on.

Notation:

X_i = prefix $\langle x_1, \ldots, x_i \rangle$

Y_i = prefix $\langle y_1, \ldots, y_i \rangle$

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ 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 $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n \Rightarrow Z$ is an LCS of X and Y_{n-1}.

Step 1:

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.
- 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.

The characterization of this theorem 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.

Again from the theorem,

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.

// The above theorem implies that there are either one or two subproblems to examine when finding an LCS of $X = < x_1 , x_2 \ldots .x_m >$ and $Y = < y_1 , y_2 \ldots .y_n >$.

Step 2: (A recursive 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:

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

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_{m-1} and Y_{n-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.

Note: 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.

Step 3: (Computing the length of an LCS)

Based on above recurrence equation, we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences.

Since there are only Θ(mn) distinct subproblems, however, we can use dynamic programming to compute the solutions bottom up.

The following procedure takes two sequences $X = < x_1 , x_2 \ldots .x_m >$ and $Y = < y_1 , y_2 \ldots .y_n >$ as inputs.

It stores the c[i, j] values in a table c[0 . . m, 0 . . 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 . . m, 1 . . 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].

```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                      then c[i, j] ← c[i − 1, j]
14                           b[i, j] ← "↑"
15                      else c[i, j] ← c[i, j − 1]
16                           b[i, j] ← "←"
17  return c and b
```

Illustrative Example:

The c and b tables computed by DP procedure on the sequences
X = <A, B, C, B, D, A, B> and
Y = <B, D, C, A, B, A>.

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].

	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

The entry 4 in c[7, 6] (the lower right-hand corner of the table) represents the length of an LCS <B, C, B, A> 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.