Dynamic Programming Longest Common Subsequence

Dr. P. Gayathri
Professor
SCOPE, VIT Vellore

Dynamic Programming (DP)

- method for solving optimization problems.
- Optimal solution for a problem is obtained by computing optimal solutions for its subproblems (optimal substructure property)
- **The idea:** Break the problem into subsub-problems, compute the solutions to the subsub-problems *once* and store the solutions, so that they can be reused repeatedly later (overlapping subproblems property).
- Memoization – Storing previously calculated values of subsub-problems.
- DP provides efficient solutions for some problems for which a brute force approach would be very slow.

Longest Common Subsequence Problem

- Given two strings, **S1** and **S2**, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings.
- Suppose we have a string W_1 = **abcd**
  - The following are some of the subsequences that can be created from the above string:
    - ab, bd, ac, ad, acd, bcd
  - The above are the subsequences as all the characters are written in increasing order with respect to their position.
  - If we write ca or da then it would be a wrong subsequence as characters are not appearing in the increasing order.
  - The total number of subsequences that would be possible is 2^n, where n is the number of characters in a string.
  - In the above string, n = 4. So the total number of subsequences = 16.

Brute Force Technique Time complexity

- Let S and T be 2 given strings. m = No. of characters in string S and n = No. of characters in string T.
- Since there are 2^m and 2^n subsequences for S and T respectively, the time complexity is $O(2^m + 2^n)$.

LCS problem using DP : Basic Formula

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i,j > 0 \text{ and } xi = yj \quad \text{--------> } b[i,j] = \text{“}\nwarrow\text{”} \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i,j > 0 \text{ and } xi \neq yj \quad \text{--------> } b[i,j] = \text{“}\uparrow\text{”} \ (or) \ \text{“}\leftarrow\text{”} \end{cases}$$

- *Note to fill b[i, j] :*

  if $c[i, j-1]$ is max then $b[i, j] =$ “$\leftarrow$”

  if $c[i-1, j]$ is max then $b[i, j] =$ “$\uparrow$”

  if ($c[i, j-1]$ and $c[i-1, j]$ are equal then $b[i, j] =$ “$\uparrow$”

LCS problem using DP : Example

- *Given Strings:*

X = A, B, C, B, D, A, B

Y = B, D, C, A, B, A

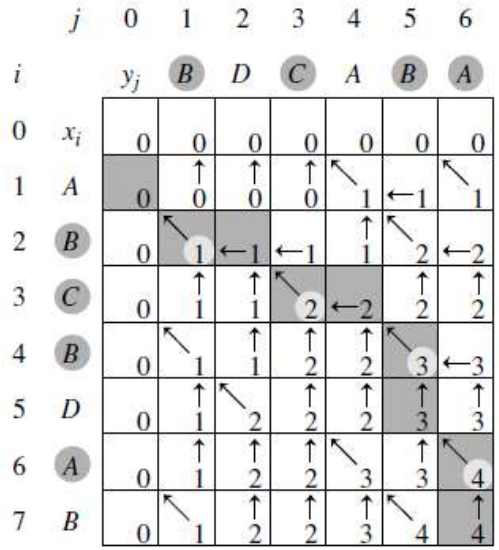
Solution:

- Length of LCS = 4
- Longest Subsequence = BCBA

Note:

Table size = m+1 rows and n+1 columns

Where, 'm' = number of characters in 1st string and 'n' = number of characters in 2nd string



	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.

Pseudocode and TC

```
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
```

The running time of this procedure is *O(mn)*.

The initial invocation is:
PRINT-LCS*(b, X, length*[*X*]*, length*[*Y*]*)*.

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

This procedure takes time *O(m + n)*, since at least one of *i* and *j* is decremented in each stage of the recursion.

Practice Problem

- *Given Strings:*

  X = a c b c f and Y = a b c d a f

Find the longest common subsequence and its length using DP.

Solution

- *Given Strings:*

  X = a c b c f

  Y = a b c d a f

- LCS length = 4
- LCS = abcf