EXPERIMENT NO 7

Title: Implement Longest Common Subsequence Algorithm by using dynamic programming approach and analyze its complexity.

Aim: - Implementation of longest common subsequences for string matching using Dynamic Programming.

Lab Outcomes : CSL 401.3: Ability to implement the algorithm using Dynamic Programming approach..

Theory:-

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

If S1 = {B, C, D, A, A, C, D}

Then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. not strictly increasing sequence).

Let us understand LCS with an example.

If S1 = {B, C, D, A, A, C, D} and S2 = {A, C, D, B, A, C}

Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ...

Among these subsequences, {C, D, A, C} is the longest common subsequence. We are going to find this longest common subsequence using dynamic programming.

LCS function defined

Let two sequences be defined as follows: X = (x1, x2...xm) and Y = (y1, y2...yn). The prefixes of X are X1, 2,...m; the prefixes of Y are Y1, 2,...n. Let LCS (Xi, Yj) represent the set of longest common subsequence of prefixes Xi and Yj. This set of sequences is given by the following.

$$LCS(X_i, Y_j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & \text{if } x_i = y_j \\ \text{longest}(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & \text{if } x_i \neq y_j \end{cases}$$

Algorithm

```
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
```

```
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)
```

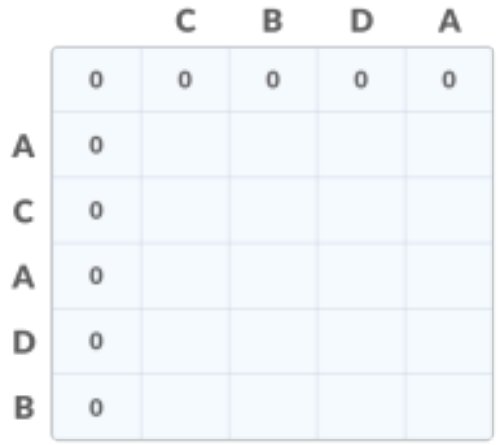
Example:-

Using Dynamic Programming to find the LCS

Let us take two sequences:

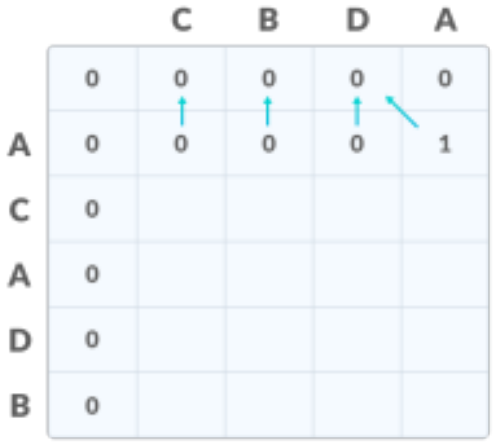
The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

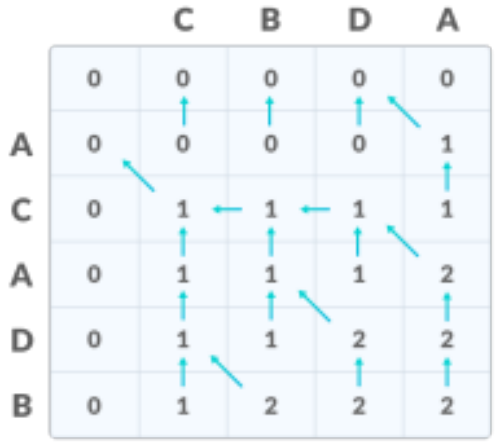


2. Fill each cell of the table using the following logic.

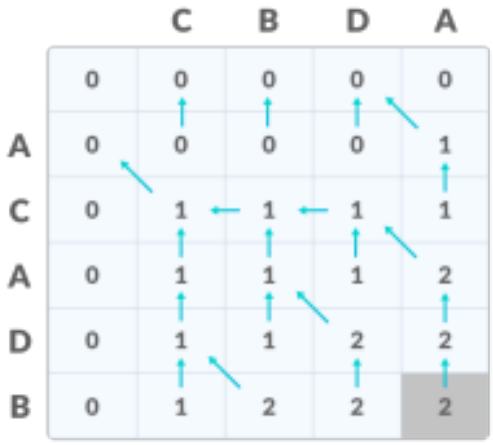
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell. 4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.



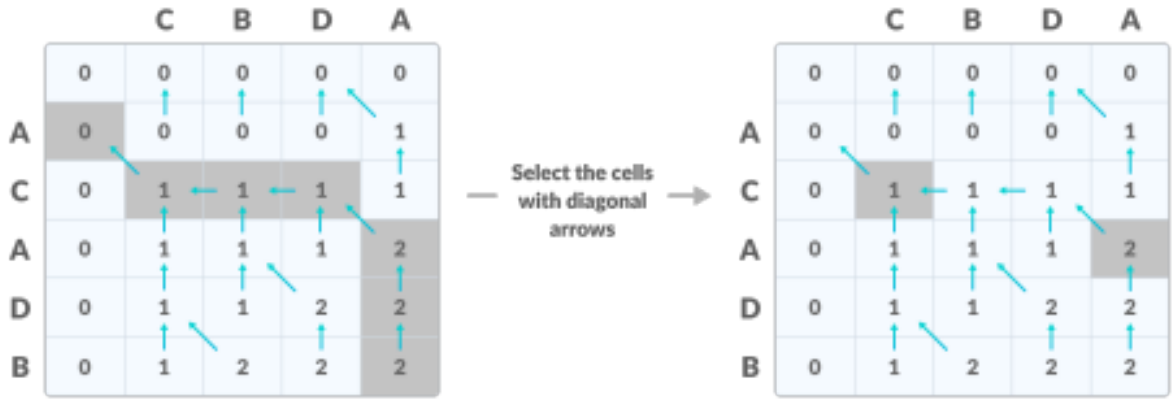
5. Step 2 is repeated until the table is filled.



6. The value in the last row and the last column is the length of the longest common subsequence.



7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Create a path according to the arrows

Thus, the longest common subsequence is CA.

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, the recursion algorithm has the complexity of 2^max(m, n).

Conclusion: - Thus we have implemented longest common subsequences for string matching using dynamic programming.