K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: E-2 Roll No.: 16010123325

Experiment No. __10__

Grade: AA / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: Study, Implementation, and Analysis of the Longest Common Subsequence Algorithm.

Objective: To compute longest common subsequence for the given two strings.

CO to be achieved:

CO 2	Analyze and solve problems for divide and conquer strategy, greedy method, dynamic programming approach and backtracking and branch & bound policies.
CO 3	Analyze and solve problems for different string matching algorithms.

Books/ Journals/ Websites referred:

1. **Ellis horowitz, Sarataj Sahni, S.Rajsekaran," Fundamentals of computer algorithm", University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein," Introduction to algortihtms",2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.**

Pre Lab/ Prior Concepts:
Data structures, Concepts of algorithm analysis

Historical Profile:
Given 2 sequences, $X = x1, ..., xm$ and $Y = y1, ..., yn$, 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.

New Concepts to be learned:
String matching algorithm, Dynamic programming approach for LCS, Applications of LCS.

Recursive **Formulation:**

Define $c[i, j]$ = length of LCS of X_i and Y_j.
Final answer will be computed with $c[m, n]$.

$c[i, j] = 0$
if $i=0$ or $j=0$.
$c[i, j] = c[i-1, j-1] + 1$
if $i,j>0$ and $x_i=y_j$

$c[i, j] = \max(c[i-1, j], c[i, j-1])$
if $i, j > 0$ and $x_i <> y_j$

Algorithm: Longest Common Subsequence

Compute length of optimal solution-

```
LCS-LENGTH ( X , Y, m, n)
for i ← 1 to m
    do c[i, 0] ← 0
for j ← 0 to n
    do c[0, j ] ← 0
for i ← 1 to m
    do for j ← 1 to n
            do if xi = y j
             then c[i, j ] ← c[i − 1, j − 1] + 1
                  b[i, j ] ← "≈"
              else if c[i − 1, j ] ≥ c[i, j − 1]
                   then c[i, j ] ← c[i − 1, j ]
                        b[i, j ] ← "↑"
                   else c[i, j ] ← c[i, j − 1]
                        b[i, j ] ← "←"
return c and b
```

Print the solution-

```
PRINT-LCS(b, X , i, j )
  if i = 0 or j = 0
    then return
  if b[i, j ] = "≈"
    then PRINT-LCS(b, X , i − 1, j − 1)
            print xi
  elseif b[i, j ] = "↑"
    then PRINT-LCS(b, X , i − 1, j )
  else PRINT-LCS(b, X , i, j − 1)
```

Initial call is PRINT-LCS(b, X, m, n).
$b[i, j]$ points to table entry whose subproblem we used in solving LCS of X_i and Y_j. When $b[i, j] = \approx$, we have extended LCS by one character. So longest com- mon subsequence = entries with $\approx$ in them.

Example: LCS computation

Longest Common Subsequence

Q. ACGTACG , ACTACG

		A	C	G	T	A	C	G
	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1
C	0	1	2	2	2	2	2	2
T	0	1	2	2	3	3	3	3
A	0	1	2	2	3	4	4	4
C	0	1	2	2	3	4	5	5
G	0	1	2	3	3	4	5	6

ACTACG

Sequence → ACTACG

If (x[i] == y[j]), c[i][j] = 1 + c[i-1][j-1]
otherwise c[i][j] = max(c[i][j-1], c[j-1][i])

Time Complexity: O(m*n)

m → length of string 1
n → length of string2

8
21/4/25.

Analysis of LCS computation

Time Complexity: O(m * n)

Space Complexity: O(m * n)

Code:

```
import java.util.Scanner;

class lcs {
    static void lcs(String S1, String S2, int m, int n) {
      int[][] table = new int[m + 1][n + 1];

      for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
          if (i == 0 || j == 0)
            table[i][j] = 0;
          else if (S1.charAt(i - 1) == S2.charAt(j - 1))
            table[i][j] = table[i - 1][j - 1] + 1;
          else
            table[i][j] = Math.max(table[i - 1][j], table[i][j - 1]);
        }
      }

      int index = table[m][n];
      int temp = index;

      char[] lcs = new char[index + 1];
      lcs[index] = '\0';

      int i = m, j = n;
      while (i > 0 && j > 0) {
        if (S1.charAt(i - 1) == S2.charAt(j - 1)) {
          lcs[index - 1] = S1.charAt(i - 1);

          i--;
          j--;
          index--;
        }

        else if (table[i - 1][j] > table[i][j - 1])
          i--;
        else
          j--;
      }

      // Printing the sub sequences
      System.out.print("S1 : " + S1 + "\nS2 : " + S2 + "\nLCS: ");
      for (int k = 0; k <= temp; k++)
        System.out.print(lcs[k]);
      System.out.println("");
    }

    public static void main(String[] args) {
```

```
    Scanner sc = new Scanner(System.in);
      System.out.println("Enter the first string: ");
      String S1 = sc.nextLine();
      System.out.println("Enter the second string: ");
      String S2 = sc.nextLine();

    int m = S1.length();
    int n = S2.length();
    lcs(S1, S2, m, n);
  }
}
```

Output:

```
Enter the first string:
ACGTACG
Enter the second string:
ACTACG
S1 : ACGTACG
S2 : ACTACG
LCS: ACTACG
```

Algorithm:

The dynamic programming (DP) approach is a more efficient way to solve the longest common subsequence problem. It avoids redundant calculations by storing the results of subproblems in a table, which can be reused.

This approach builds a 2D table where the entry at dp[i][j] represents the length of the LCS of the first i characters of the first string and the first j characters of the second string.

Advantages:

The DP approach has a time complexity of O(m * n) and space complexity of O(m * n), making it much more efficient than the recursive approach.

CONCLUSION:
The above experiment highlights implementation of LCS in Java using dynamic programming (DP) approach.