**Batch: Roll No.:**

**Experiment No. \_\_\_\_**

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

**Signature of the Staff In-charge with date**

|  |
| --- |
| **Title: Implementation of Longest Common Subsequence String Matching 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.Rajasekaran,” Fundamentals of computer algorithm”, University Press**
2. **T.H.Cormen ,C.E.Leiserson,R.L.Rivest and C.Stein,” Introduction to algorithms”,2nd Edition ,MIT press/McGraw Hill,2001**
3. **http://www.math.utah.edu/~alfeld/queens/queens.**
4. **https://www.programiz.com/dsa/longest-common-subsequence**
5. **https://www.gyaanibuddy.com/assignments/assignment-detail/longest-common-subsequence-algorithm/**
6. **https://www.codingninjas.com/codestudio/library/longest-common-subsequence**

**Pre Lab/ Prior Concepts:**

Data structures, Concepts of algorithm analysis

**Historical Profile:**

Given 2 sequences, *X* = *x*1 *, ..., xm*  and *Y* = *y*1 *, ... , 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 *Xi* 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 xi=yj

c[i, j]= max(c[i − 1, j ], c[i, j − 1])

if i, j > 0 and xi <> yj

**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*)*

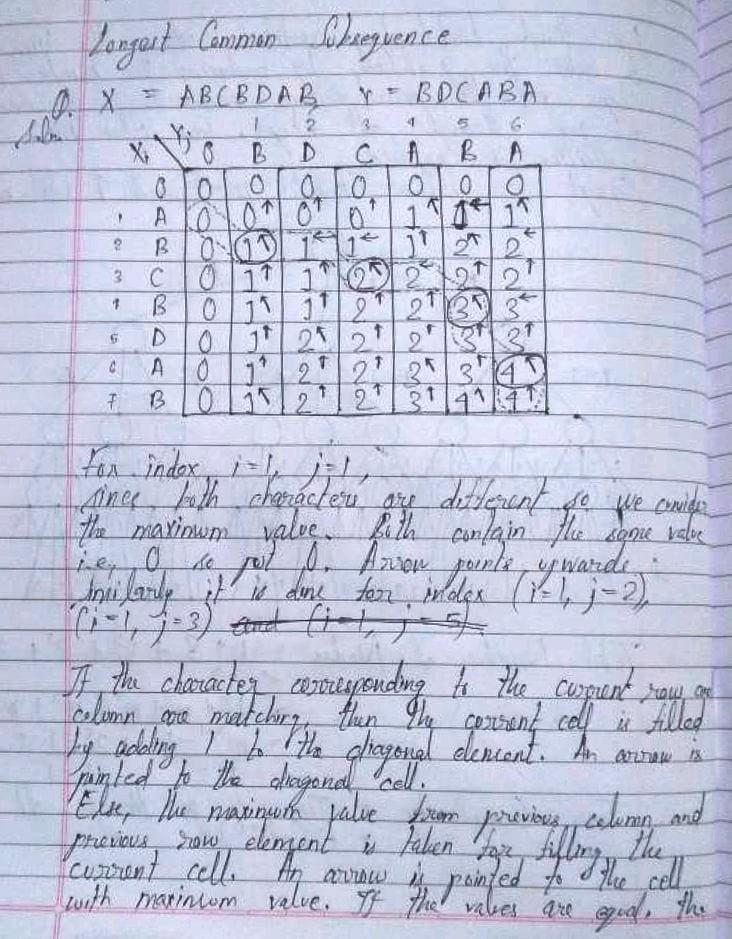
Initial call is PRINT-LCS*(b, X , m, n)*.

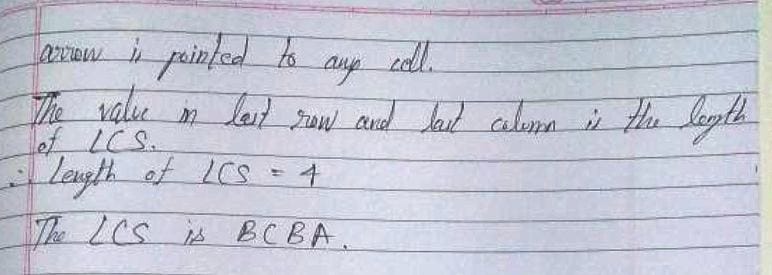
*b*[*i, j* ] points to table entry whose subproblem we used in solving LCS of *Xi*

and *Y j.*

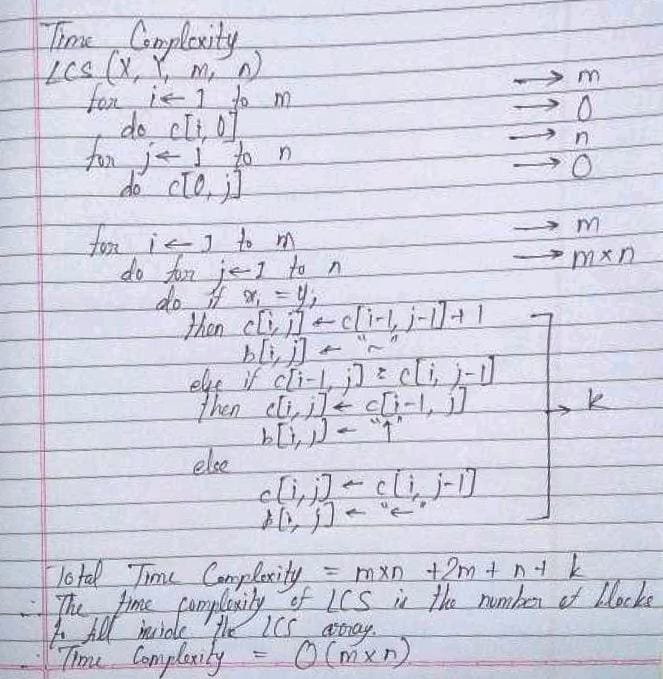
When *b*[*i, j* ] = ≈, we have extended LCS by one character. So longest com- mon subsequence = entries with ≈ in them.

**Example: LCS computation**

****

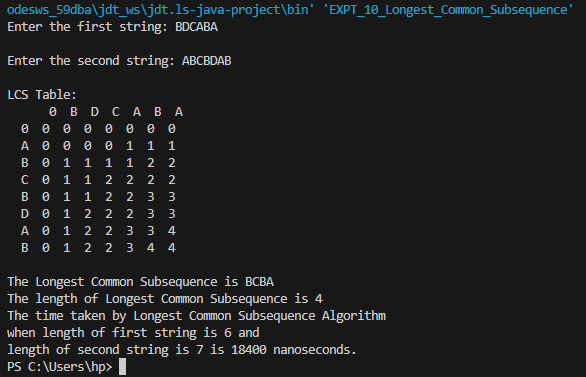
****

**Analysis of LCS computation**

****

The Space Complexity is O(n\*m) as extra space is used to store the longest common subsequence value after considering both the strings until a particular index.

**Output:**



**CONCLUSION:**

Thus, in this experiment, the concept of Dynamic Programming using Bottom-up approach was used to solve the Longest Common Subsequence problem. The time complexity of this approach is far better (O(n\*m)) as compared to that of using plain recursion (O(2N)). So in cases where speed is needed, Dynamic Programming approach should be used. However, the space complexity of this approach is not as good (O(n\*m)) as compared to that of using plain recursion (O(1)). So in cases where space is limited, Plain Recursion approach should be used.