

**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.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.**](http://www.math.utah.edu/~alfeld/queens/queens)



**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*)*

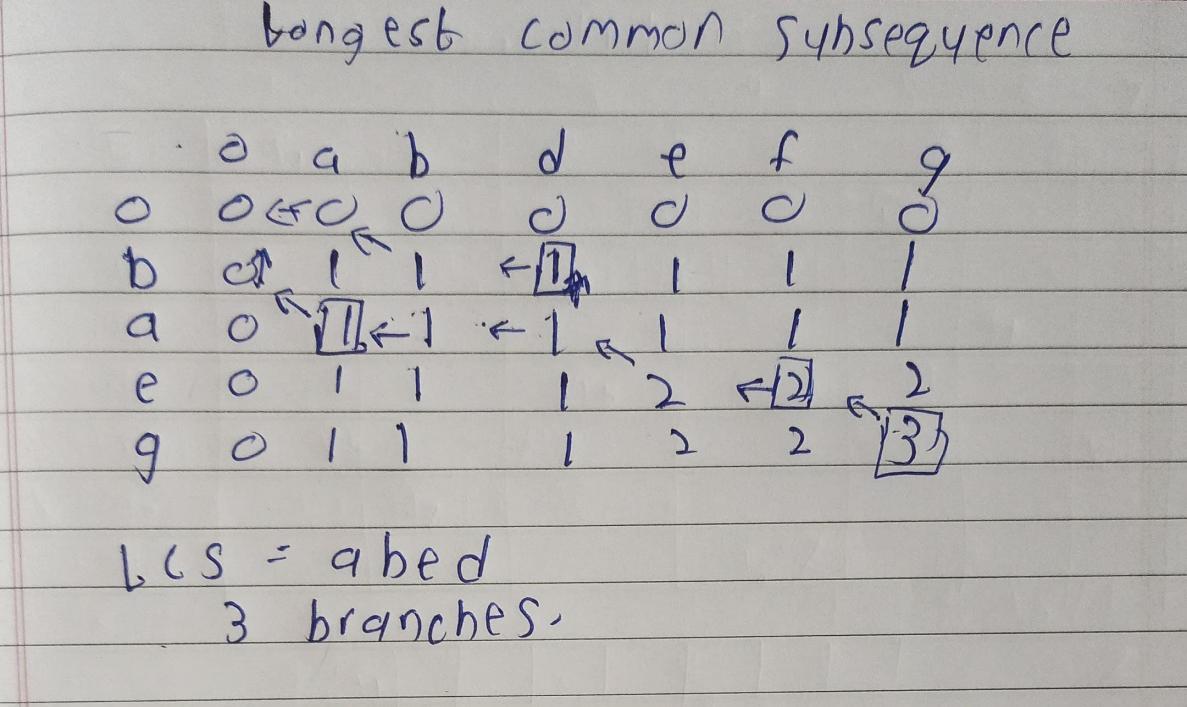


Initial call is PRINT-LCS*(b, X , m, n)*.

*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**

Dynamic Programming Approach:

The most efficient way to solve the LCS problem is through dynamic programming. This approach breaks down the problem into smaller subproblems and builds up the solution incrementally. The dynamic programming approach has a time complexity of O(mn), where m and n are the lengths of the two sequences. The space complexity can be optimized to O(min(m, n)).

Space Optimization:

The dynamic programming approach uses a table to store intermediate results, which can be optimized to reduce space complexity. Instead of using a 2D array to store the entire table, we can use two 1D arrays of size min(m, n) to store only the necessary information for the current and previous rows. This reduces the space complexity to O(min(m, n)) while maintaining the same time complexity.



**Code**

**#include <iostream>**

**#include <string>**

**#include <algorithm>**

**#include <cstring>**

**using namespace std;**

**int getLengthOfLCS(char str1[], char str2[], int p, int q) {**

**if (p == 0 || q == 0)**

**return 0;**

**if (str1[p - 1] == str2[q - 1])**

**return 1 + getLengthOfLCS(str1, str2, p - 1, q - 1);**

**else**

**return max(getLengthOfLCS(str1, str2, p, q - 1), getLengthOfLCS(str1, str2, p - 1, q));**

**}**

**int maxValue(int length1, int length2) {**

**return (length1 > length2) ? length1 : length2;**

**}**

**int main() {**

**string string1, string2;**

**cout << "Enter first sequence: ";**

**getline(cin, string1);**

**cout << "Enter second sequence: ";**

**getline(cin, string2);**

**char str1[string1.length() + 1];**

**strcpy(str1, string1.c\_str());**

**char str2[string2.length() + 1];**

**strcpy(str2, string2.c\_str());**

**int p = string1.length();**

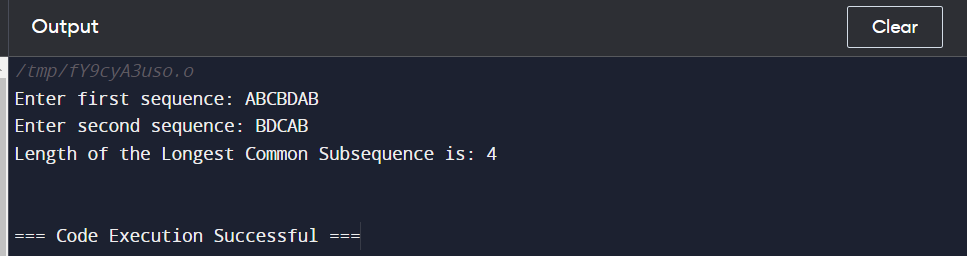
**int q = string2.length();**

**cout << "Length of the Longest Common Subsequence is: " << getLengthOfLCS(str1, str2, p, q) << endl;**

**return 0;**

**}**

**Output:**

****

**Algorithm:**

1. **First, we create a table of dimensions (p + 1)\*(q + 1) where p and q are the lengths of the given sequences. In the created table, we set 0 to the first row and the first column.**
2. **All the remaining cells of the table are filled by using the following steps:**
   1. **If the characters of the corresponding row and the column are the same and matched successfully, fill the current cell with 1 to the diagonal element and point an arrow to the diagonal cell.**
   2. **If the characters are not matched, we fill the current cell with the value of the previous column and previous row element. The arrow points to the cell having maximum value, and when the value in both the cells is equal, the arrow will point to any of them.**
   3. **We repeat step 2 until the complete table is not filled.**
   4. **The length of the longest common subsequence is the value present in the last row and the last column.**



* 1. **In order to get the LCS, we follow the direction of the arrow from the last element. The elements corresponding to the brackets() are form the longest common subsequence.**

**CONCLUSION:**

**In this experiment, we have successfully completed and implementation of Longest Common Subsequence String Matching Algorithm.**