

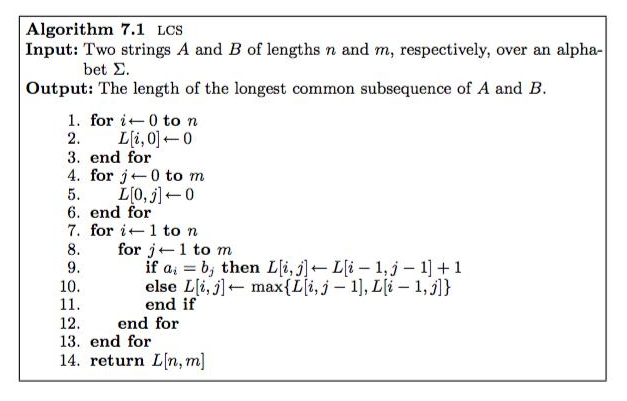
**Experiment No. : 5**

**Title:** Longest Common Subsequence using Dynamic Programming

# Batch:B2 Roll No.:1914078 Experiment No.:5

**Aim:** Implement Longest Common Subsequence using Dynamic Programming.

# Algorithm:

****

index = L[n][m]

string lcs

I = n

J = m

While i>0 and j > 0

If( a[i] == b[j])

Lcs[index] = a[i]

i--, j --, index--;

else if (L[i-1][j] > l[i][j-1])

i--;

else

j--;

print lcs

**Example of** Longest Common Subsequence (LCS) **Step by Step Execution**

Take strings ABAC and ADBC

After first instance of inner loop

A D B C

0 0 0 0 0

A 0 1 1 1 1

B 0

A 0

C 0

Second instance

A D B C

0 0 0 0 0

A 0 1 1 1 1

B 0 1 1 2 2

A 0

C 0

Third instance

A D B C

0 0 0 0 0

A 0 1 1 1 1

B 0 1 1 2 2

A 0 1 1 2 2

C 0

Fourth instance

A D B C

0 0 0 0 0

A 0 1 1 1 1

B 0 1 1 2 2

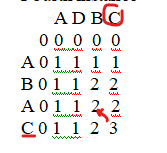
A 0 1 1 2 2

C 0 1 1 2 3

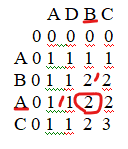
Thus length of LCS is 3

To print it, start from the end of matrix and declare string of length 3

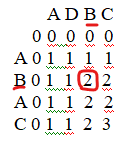
Since the characters are same



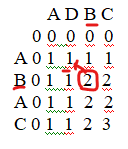
Add C to the end of lcs and go diagonally left in the matrix

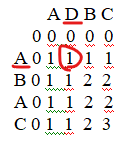


The characters are different, so compare number above and to the left and move to the greater number, in this case 2.

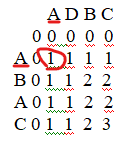


Since characters are same go diagonally left and up and add B before C.





Characters are different so compare number above and to the left. 1 is greater.



Characters are same. Add A to the first position of lcs and end the operation.

Thus we have the longest common subsequence i.e. ABC

# Derivation of Analysis:

Time complexity:

• Time complexity to find LCS is O(n²) and space required is O(n²) for 2 input sequences

• Time complexity to find only length of LCS is still O(n²) but space required is O(n) for 2 input sequences While the algorithm is polynomial with respect to a sequence length, it is exponential with respect to the number of sequences.

Time complexity to find LCS of k sequences is O(n^k).

# Program(s):

#include <stdio.h>

#include <string.h>

int i, j, m, n, LCS\_table[20][20];

char S1[20], S2[20], b[20][20];

void lcsAlgo()

{

    m = strlen(S1);

    n = strlen(S2);

    for (i = 0; i <= m; i++)

        LCS\_table[i][0] = 0;

    for (i = 0; i <= n; i++)

        LCS\_table[0][i] = 0;

    for (i = 1; i <= m; i++)

        for (j = 1; j <= n; j++)

        {

            if (S1[i - 1] == S2[j - 1])

            {

                LCS\_table[i][j] = LCS\_table[i - 1][j - 1] + 1;

            }

            else if (LCS\_table[i - 1][j] >= LCS\_table[i][j - 1])

            {

                LCS\_table[i][j] = LCS\_table[i - 1][j];

            }

            else

            {

                LCS\_table[i][j] = LCS\_table[i][j - 1];

            }

        }

    int index = LCS\_table[m][n];

    char lcsAlgo[index + 1];

    lcsAlgo[index] = '\0';

    int i = m, j = n;

    while (i > 0 && j > 0)

    {

        if (S1[i - 1] == S2[j - 1])

        {

            lcsAlgo[index - 1] = S1[i - 1];

            i--;

            j--;

            index--;

        }

        else if (LCS\_table[i - 1][j] > LCS\_table[i][j - 1])

            i--;

        else

            j--;

    }

    printf("\nString 1: : %s \nString 2: : %s \n", S1, S2);

    printf("\nLCS(Using Dynamic Programming) of the two Strings is: %s", lcsAlgo);

}

int main()

{

    printf("\nEnter String 1: ");

    scanf("%s", S1);

    printf("\nEnter String 2: ");

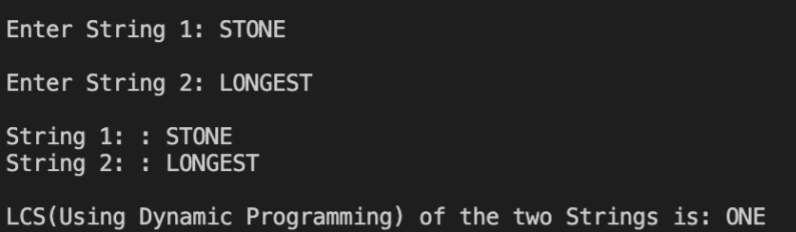
    scanf("%s", S2);

    lcsAlgo();

    printf("\n");

}

**Output(o):**



**Questions:-** Explain dynamic programming approach for LCS and write the various applications of LCS.

**Ans)**

Bottom Up Calculation Of Dynamic Programming

Rather than continue with our top-down recursive algorithm, we can employ another standard dynamic programming technique and transform the algorithm into one which works from the bottom up. Recall that, in the general case, to calculate lcs(xs, ys) we needed to solve three subproblems, lcs(xb, yb), lcs(xb, ys), lcs(xs, yb), where xb is the result of removing the end element from xs, and similarly for yb.

|  |
| --- |
| \*xb, xe = xs \*yb, ye = ys if xe == ye: return lcs(xb, yb) + [xe] else:  return max(lcs(xs, yb), lcs(xb, ys), key=len) |

These subproblems in turn depend on the LCSes of yet shorter prefix pairs of xs and ys. A bottom up approach reverses the direction and calculates the LCSes of all prefix pairs of xs and ys, from shortest to longest. We iterate through these prefix pairs filling a rectangular grid with the information needed to continue and to eventually reconstruct an LCS. Once again, the code is easier to read than an explanation.

|  |
| --- |
| from collections import defaultdict, namedtuple from itertools import product    def lcs\_grid(xs, ys):  Cell = namedtuple('Cell', 'length move') grid = defaultdict(lambda: Cell(0, 'e')) sqs = product(enumerate(ys), enumerate(xs)) for (j, y), (i, x) in sqs:  if x == y:  cell = Cell(grid[(j-1, i-1)].length + 1, '\\') else: left = grid[(j, i-1)].length over = grid[(j-1, i)].length if left < over: cell = Cell(over, '^') else: cell = Cell(left, '<') grid[(j, i)] = cell return grid |

The implementation here makes a couple of unusual choices. It models a 2 dimensional grid of values using a dict keyed by (i, j) pairs, rather than a more conventional array representation. Furthermore, it uses a default dict, which returns a value even for keys which haven’t been filled in. I’ve used this data structure primarily to avoid special case code at the edges of the grid. For example, a grid cell can be accessed at position (j-1, i-1) even if i or j is zero, and that cell automatically takes the desired (0, 'e') value.

**Applications**

1. in compressing genome resequencing data
2. to authenticate users within their mobile phone through in-air signatures

# Outcome:

Implement Greedy and Dynamic Programming algorithms

**Conclusion: (Based on the observations):**

Implemented program to find the longest common subsequence of 2 given strings using Dynamic Programming and printed the sequence

# References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication