**SVKM’s NMIMS**

**Mukesh Patel School of Technology Management & Engineering**

**Computer Engineering Department**

Program: B.Tech. Sem V

**Course: Design and Analysis of Algorithms**

**List of Experiments**

w.e.f. 1st Jul 2020

**Faculty:** Abhay Kolhe.

LAB Manual

|  |  |
| --- | --- |
| Roll No. B032 | Name: Naman Garg |
| Class : B Tech CS B | Batch : B2 |
| Date of Experiment: 8-9-2020 | Date of Submission:8-9-2020 |
| Grade : | Time of Submission: |
| Date of Grading: |  |

**B.1 Software Code written by student:**

# NAMAN GARG

# B032

# Aim: Implementation of Dynamic Programming Technique Algorithm Design. Write a program to implement Longest Common Subsequence (LCS) problem.

def lcs(X, Y, m, n):

    L = [[None]\*(n+1) for i in range(m+1)]

    L2 = [['X']\*(n+1) for i in range(m+1)]

    # Following steps build L[m+1][n+1] in bottom up fashion. Note

    # that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]

    for i in range(m + 1):

        for j in range(n + 1):

            if i == 0 or j == 0:

                L[i][j] = 0

            elif X[i - 1] == Y[j - 1]:

                L[i][j] = L[i - 1][j - 1] + 1

                L2[i][j] = 'D'

            else:

                L[i][j] = max(L[i - 1][j], L[i][j - 1])

                if L[i - 1][j] >= L[i][j - 1]:

                    L2[i][j] = 'U'

                else:

                    L2[i][j] = 'L'

                # Following code is used to print LCS

    index = L[m][n]

    print('\nOutput Matrix 1:')

    for i in range(m + 1):

        for j in range(n + 1):

            print(L[i][j], end=' ')

        print()

    print('\nOutput Matrix 2:')

    for i in range(m + 1):

        for j in range(n + 1):

            print(L2[i][j], end=' ')

        print()

    print('\nThe length of the LCS is:', index)

    # Create a character array to store the lcs string

    lcs = [""] \* (index + 1)

    lcs[index] = ""

    # Start from the right-most-bottom-most corner and

    # one by one store characters in lcs[]

    i = m

    j = n

    while i > 0 and j > 0:

        # If current character in X[] and Y are same, then

        # current character is part of LCS

        if X[i - 1] == Y[j - 1]:

            lcs[index - 1] = X[i - 1]

            i -= 1

            j -= 1

            index -= 1

        # If not same, then find the larger of two and

        # go in the direction of larger value

        elif L[i - 1][j] > L[i][j - 1]:

            i -= 1

        else:

            j -= 1

    print('The LCS is:', "".join(lcs))

# Driver program

X = input('Enter the first string: ')

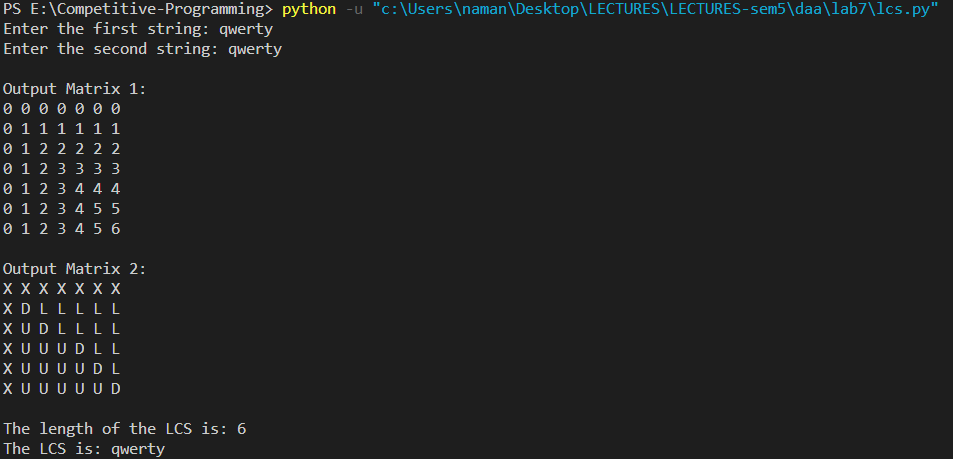
Y = input('Enter the second string: ')

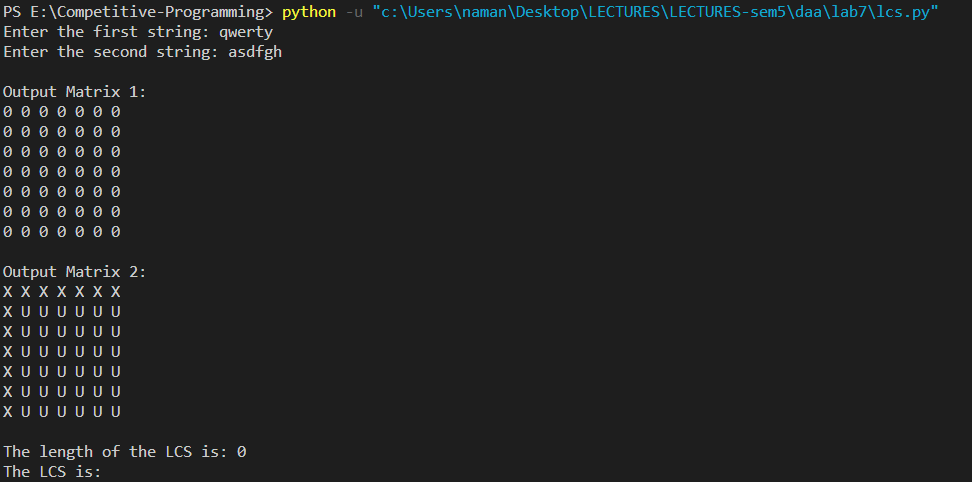
m = len(X)

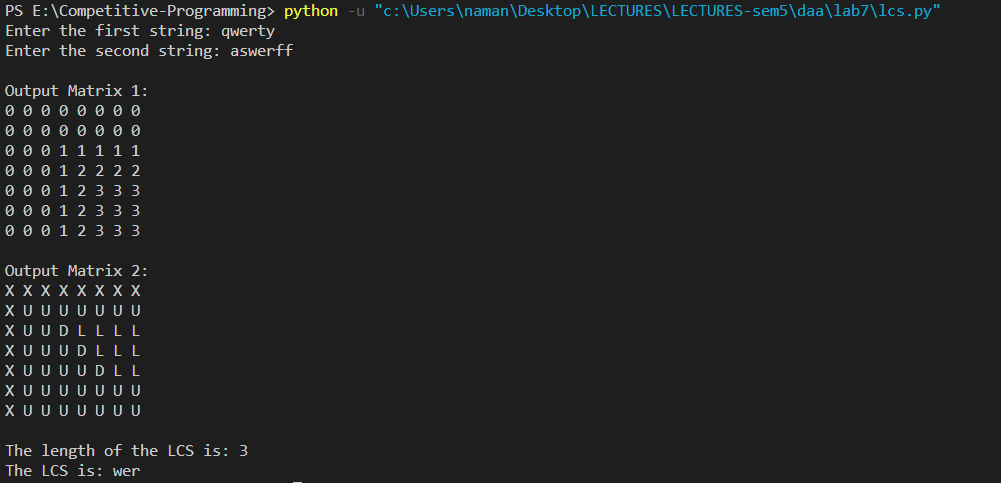
n = len(Y)

lcs(X, Y, m, n)

**B.2 Input and Output:**

****

****

****

**B.3 Observations and learning:**

implemented DP for solving the longest common subsequence problem. Dynamic programming optimizes the problem by storing the results of the subsequences as compared to the recursive approach to a problem. We maintained 2 matrices – 1 for storing the lengths at each subproblem and the other for storing the directions while traversing back the matrix. The characters of the string are compared one by one and whenever we hit a match, we basically put the value for the character in the matrix to the value of the diagonal value +1 . In case of a miss, we store the maximum of the upper and left element. The final length of the longest common subsequence is the last value of the matrix.

To get the string of lcs, we will traverse upward and then when we must go diagonal, we store the character in the lcs string. So, after finding all the letters of the string, they are stored in reverse manner as we traversed backward so we need to print it backwards. One thing to note is that, if there exit multiple subsequences of the same maximum length, we can get only 1 subsequence from the algorithm.

**B.4 Conclusion:**

I learnt and successfully implemented dynamic programming for longest common subsequence problem. I printed both the matrices for the length and the direction. I then printed the length of the lcs and the string as well. I understood how it is more optimal than using recursion as it cuts down the complexity to a polynomial.

**B.5 Question of Curiosity**

**Q.1 Identify & discuss the real life applications LCS.**

There are several motivating applications.

Molecular biology. DNA sequences (genes) can be represented as sequences of four letters ACGT, corresponding to the four sub molecules forming DNA. When biologists find a new sequence, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

File comparison. The Unix program "diff" is used to compare two different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the two files; any line in the subsequence has not been changed, so what it displays is the remaining set of lines that have changed. In this instance of the problem we should think of each line of a file as being a single complicated character in a string.

Screen redisplays. Many text editors like "emacs" display part of a file on the screen, updating the screen image as the file is changed. For slow dial-in terminals, these programs want to send the terminal as few characters as possible to cause it to update its display correctly. It is possible to view the computation of the minimum length sequence of characters needed to update the terminal as being a sort of common subsequence problem (the common subsequence tells you the parts of the display that are already correct and don't need to be changed).

**Reference: https://www.ics.uci.edu/~eppstein/161/960229.html**

**Q.2 Why do we need the matrix B/b in the above algorithm?**

The b matrix is used to extract the lcs string from the values inside the matrix. We store the directions we used to get values for the ‘a’ matrix. Example: suppose we hit a match between two characters in the 2 strings, the value we write for that in the ‘a’ matrix is the value of its upper diagonal + 1 ; so we will store a symbol denoting diagonal direction in the b matrix. After getting all directions, we will traverse backward from the last element and we will store the character for which we hit the diagonal symbol. Since we are traversing backwards, we will get the lcs string in reverse fashion. We will reverse that to get the final output.

**Q.3 Devise an algorithm to print the LCS in forward direction.**

We construct L[m+1][n+1] table and traverse the 2D array starting from L[m][n]. For current cell L[i][j] in the matrix-

a) If the last characters of X and Y are same (i.e. X[i-1] == Y[j-1]), then the character must be present in all LCS of substring X[0…i-1] and Y[0..j-1]. We simply recurse for L[i-1][j-1] in the matrix and append current character to all LCS possible of substring X[0…i-2] and Y[0..j-2].

b) If the last characters of X and Y are not same (i.e. X[i-1] != Y[j-1]), then LCS can be constructed from either top side of the matrix (i.e. L[i-1][j]) or from left side of matrix (i.e. L[i][j-1]) depending upon which value is greater. If both the values are equal(i.e. L[i-1][j] == L[i][j-1]), then it will be constructed from both sides of matrix. So based on values at L[i-1][j] and L[i][j-1], we go in direction of greater value or go in both directions if the values are equal.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*