Marwadi University Marwadi Chandarana Group NAAC A+	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: DAA (01CT0512)	**AIM: Implementing Longest Common Sub-sequence using Dynamic Programming Approach**	
Experiment No: 8	**Date:**	**Enrolment No: 92301733046**

Theory:

This method finds the Longest Common Subsequence (LCS) between two strings using Dynamic Programming. A 2D table is created where each cell l[i][j] represents the LCS length between the first i characters of str1 and the first j characters of str2. If the characters match, the value is taken as 1 + l[i-1][j-1]; otherwise, it is the maximum of excluding either character.

- Subproblem Overlapping & Optimal Substructure

Programming Language: Python

Code:

```
def LCS(str1,str2):
    m=len(str1)
    n=len(str2)
    L=[[0 for i in range(n+1)]for j in range( m+1)]
    for i in range(1,m+1):
        for j in range(1,n+1):
            if str1[i-1]==str2[j-1]:
                L[i][j]=1+L[i-1][j-1]
            else:
                L[i][j]=max(L[i][j-1],L[i-1][j])
    return L[m][n]


str1=str(input("Enter first string: "))
str2=str(input("Enter second string: "))
print("The longest common subsequence is: ",LCS(str1,str2))
```

Marwadi University Marwadi Chandarana Group NAAC A+	**Marwadi University** **Faculty of Engineering and Technology** **Department of Information and Communication Technology**	
Subject: DAA (01CT0512)	**AIM: Implementing Longest Common Sub-sequence using Dynamic Programming Approach**	
Experiment No: 8	**Date:**	**Enrolment No: 92301733046**

Output:

```
PS D:\DAA\Lab and Lecture Codes> python -u "d:\DAA\Lab and Lecture Codes\LOngest_common_subsequence.
py"
Enter first string: longest
Enter second string: stone
The longest common subsequence is:  3
PS D:\DAA\Lab and Lecture Codes>
```

Space Complexity

- **O(m × n)**
  Justification: The algorithm builds a DP table of size (m+1) × (n+1) to store intermediate LCS values, which requires space proportional to the product of the string lengths.

Time Complexity

- **Best Case: O(m × n)**
  Justification: Even if the two strings are identical and match perfectly, the algorithm still fills all entries of the DP table.
- **Worst Case: O(m × n)**
  Justification: If the two strings have no matching characters, the algorithm still processes all m × n cells in the DP table.