Name	**Advait Ravi Sapkal**
UID No.	**2021700055**
Experiment No.	**4**

AIM:	To implement the solution to the longest common subsequence problem.
Program 1	
PROBLEM STATEMENT:	A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. For example, $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$. Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y. For example, if $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$, the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y. The sequence $\langle B, C, A \rangle$ is not a *longest* common subsequence (LCS) of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$, which is also common to both X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$, since X and Y have no common subsequence of length 5 or greater. In the ***longest-common-subsequence problem***, we are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ and wish to find a maximum-length common subsequence of X and Y.
ALGORITHM:	This solution has many applications in the areas of DNA sequencing, encoding, etc. ## Longest Common Subsequence: ### 1. Characterizing the Optimal Substructure: Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y. 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y. 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}. ***Proof*** (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a *longest* common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length-$(k - 1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction. (2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y. (3) The proof is symmetric to (2). ■

2. Recursive Solution:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

3. Computing subproblems after recognizing overlapping substructure:

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

4. Constructing solution from the computed data:

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Trace:

Input: X = < A; B; C; B; D; A; B>, Y = < B; D; C; A; B; A>

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Output: Z = < B C B A >

Analysis:

1. As there are 2^n subsequences possible for a string of length n, and similarly 2^m for the second string, a brute force method that compares each subsequence of X with Y would have taken total time complexity $O(2^{m+n})$
2. If we use the naïve recursive approach, in the worst case of the strings being completely dissimilar, we always make 2 more recursive calls per one recursive call. Hence the time complexity would be $O(2^l)$ where l is the length of the longer input string.
3. In the dynamic programming approach, we are iteratively making a table of size m x n while traveling it once. O(mn). When reconstructing it, in the worst case we would just go linearly vertical or horizontal (strings are completely dissimilar), which would take O(l) time where l is the length of one of the strings. Hence the overall time complexity is O(mn). As we maintain a table of size m x n there is also a space complexity of O(mn).
4. Hence, we get considerable improvement using dynamic programming. In fact, this space can be reduced to linear if only the length of the LCS is needed, by maintaining only the current row and previous row in the table of the computation step.

PROGRAM:

```
// bottom up approach for longest common subsequence
#include <stdio.h>
#include <string.h>

int max(int a, int b)
{
    if (a > b)
        return a;
```

```
    else
        return b;
}

void lcs(char str1[], char str2[], int sz1, int sz2)
{

    // making the table bottom up
    int table[sz1 + 1][sz2 + 1];
    for (int i = 0; i <= sz1; i++)
    {
        for (int j = 0; j <= sz2; j++)
        {
            if (i == 0 || j == 0)
            {
                table[i][j] = 0;
            }
            else if (str1[i - 1] == str2[j - 1])
            {
                table[i][j] = table[i - 1][j - 1] + 1;
            }
            else
            {
                table[i][j] = max(table[i - 1][j], table[i][j - 1]);
            }
        }
    }

    // finding the lcs
    int index = table[sz1][sz2];
    int temp = index;

    char lcs[index + 1];
    lcs[index] = '\0';

    int i = sz1;
    int j = sz2;
    while (i > 0 && j > 0)
    {
        if (str1[i - 1] == str2[j - 1])
        {
            lcs[index - 1] = str1[i - 1];
            i--;
            j--;
            index--;
        }
        else if (table[i - 1][j] < table[i][j - 1])
            j--;
        else
```

```
            i--;
    }
    printf("LCS of %s and %s is :\n", str1, str2);
    printf("[ %s ] and length is [ %d ] \n", lcs, temp);
}

int main()
{
    char str1[100];
    char str2[100];

    printf("enter string 1:\n"); scanf("%s", str1);
    printf("enter string 2:\n"); scanf("%s", str2);

    lcs(str1, str2, strlen(str1), strlen(str2));
}
```

RESULT:

Outputs:

```
D:\DAA>cd "d:\DAA\" && gcc daa_4.c -o daa_4 && "d:\DAA\"daa_4
enter string 1:
abcbdab
enter string 2:
bdcaba
LCS of abcbdab and bdcaba is :
[ bcba ] and length is [ 4 ]
```

```
d:\DAA>cd "d:\DAA\" && gcc daa_4.c -o daa_4 && "d:\DAA\"daa_4
enter string 1:
aaaaaa
enter string 2:
bbbbbb
LCS of aaaaaa and bbbbbb is :
[  ] and length is [ 0 ]
```

```
d:\DAA>cd "d:\DAA\" && gcc daa_4.c -o daa_4 && "d:\DAA\"daa_4
enter string 1:
aaaaaa
enter string 2:
aaaaaa
LCS of aaaaaa and aaaaaa is :
[ aaaaaa ] and length is [ 6 ]
```

Conclusions:

1. Solving the LCS problem using dynamic programming gives a significant improvement in performance compared to brute force or recursive approach.
2. Hence, we learned about the four steps of dynamic programming by solving the longest common subsequence problem.