Experiment Number: 10

Problem Statement:- Given two sequences X of length m and Y of length n as

$X = <x_1, x_2, ..., x_m>$

$Y = <y_1, y_2, ..., y_n>$

Find the *longest* common subsequence (LCS).

Aim:- Implementation of Longest Common Subsequence Algorithm.

Theory:-

Given two sequences X and Y, a sequence G is said to be a *common subsequence* of X and Y, if G is a subsequence of both X and Y. For example, if

$X = \langle A, C, B, D, E, G, C, E, D, B, G \rangle$ And

$Y = \langle B, E, G, C, F, E, U, B, K \rangle$

then a common subsequence of X and Y could be

$G = \langle B, E, E \rangle$.

The **longest common subsequence** (**LCS**) **problem** is to find the longest subsequence common to all sequences in a set of sequences (often just two).

Step 1: Characterize optimality

The brute force procedure would involve enumerating all 2^m subsequences of X (again simply consider all binary strings of length m) and check if they are also subsequences of Y keeping track of the longest one. Clearly this produces exponential run time and does not take advantage of the optimal substructure of the solution.

Define the i^{th} *prefix* of a sequence as the first i elements

$X_i = <x_1, x_2, ..., x_i>$

with X_0 representing the empty sequence.

If we assume that $Z = <z_1, z_2, ..., z_k>$ is a LCS (with length k) of X and Y then one of the following three cases must hold:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a LCS of X_{m-1}, Y_{n-1} . Basically if the last elements of both sequences are the same then it must be the last element of the LCS and the $k-1$ prefix of the LCS must be a LCS of the $m-1$ and $n-1$ prefixes of the original sequences.
2. If $x_m \neq y_n$, then if $z_k \neq x_m$ Z is a LCS of X_{m-1}, Y. Basically if the last element of the LCS is *not* the same as the last element of X then it must be a LCS of the prefix of X without the last element.
3. If $x_m \neq y_n$, then if $z_k \neq y_n$ Z is a LCS of X, Y_{n-1}. Basically if the last element of the LCS is *not* the same as the last element of Y then it must be a LCS of the prefix of Y without the last element.

In all three cases we see that the LCS of the original two sequences contains a LCS of *prefixes* of the two sequences (smaller versions of the original problem) ⇒ *optimal substructure problem.*

Step 2: Define the recursive solution (top-down)

Case 1 reduces to the *single* subproblem of finding a LCS of X_{m-1}, Y_{n-1} and adding $x_m = y_n$ to the end of Z.

Cases 2 and 3 reduces to *two* subproblems of finding a LCS of X_{m-1}, Y and X, $Y_{n-1}$and selecting the longer of the two (note both of these subproblems involve also solving the subproblem of Case 1).

Hence if we let *c[i,j]* be the length of a LCS for X_i and Y_j we can write the recursion described by the above cases as

$$c[i,j]=\begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \;\; x_i=y_j \;\; \text{(case 1)} \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \;\; x_i\neq y_j \;\; \text{(cases 2 and 3)} \end{cases}$$

Note that not all subproblems are considered depending on which recursive branch is selected.

Step 3: Compute the length of the LCS (bottom-up)

Since each step of the recursion removes at least one element from one of the sequences, there are only Θ(*mn*) subproblems to consider. Hence we can solve it by creating two tables - *C* an *m* x *n* table storing the LCS lengths and *B* an *m* x *n* table for reconstructing the LCS. When the procedure is complete, the optimal length of the LCS will be stored in *c[m,n]*. Thus since we fill in the entire table, the procedure will take O(*mn*).

Algorithm

```
LCS − Length(X, Y )
   m ← length[X]
    n ← length[Y ]
  for i ← 1 to m
       c[i, 0] ← 0
  for j ← 0 to n
      c[0, j] ← 0
  for i ← 1 to m
     for j ← 1 to n
         if xi = yj
          c[i, j] ← c[i − 1, j − 1] + 1
           b[i, j] ← "-"
         else if c[i − 1, j] ≥ c[i, j − 1]
          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
```

Step 4: Construct an optimal LCS

Start at *any* entry containing the max-length (for example *c[m,n]*) and follow the arrows through the table adding elements in reverse order whenever a ↖ occurs. At worst we move up or left at each step giving a run time of O(*m* +*n*).

Alternatively we could avoid the *B* matrix (saving some space) and reconstruct the LCS from *C* at each step in O(1) time (using only the surrounding table cells), however it does not provide any improvement in the asymptotic run time.

Example

Consider the two sequences

X = <A, B, C, B, A>

Y = <B, D, C, A, B>

We will fill in the table row-wise starting in the upper left corner using the following formulas

$$x_i = y_j \;\Rightarrow\; c[i,j] = c[i-1,j-1]+1 \qquad \nwarrow$$

$$x_i \neq y_j \;\Rightarrow\; c[i-1,j] \geqslant c[i,j-1]$$
$$c[i,j] = c[i-1,j] \qquad \uparrow$$

$$c[i-1,j] < c[i,j-1]$$
$$c[i,j] = c[i,j-1] \qquad \leftarrow$$

The completed table is given by

	j	→	1	2	3	4	5
i		y_j	**B**	**D**	**C**	**A**	**B**
↓	x_i	0	0	0	0	0	0
1	**A**	0	0 ↑	0 ↑	0 ↑	1 ↖	1 ←
2	**B**	0	1 ↖	1 ←	1 ←	1 ↑	2 ↖
3	**C**	0	1 ↑	1 ↑	2 ↖	2 ←	2 ↑
4	**B**	0	1 ↖	1 ↑	2 ↑	2 ↑	3 ↖
5	**A**	0	1 ↑	1 ↑	2 ↑	3 ↖	3 ↑

Thus the optimal LCS length is *c[m,n]* = 3.

Constructing an optimal LCS starting at *c[5,5]* we get *Z* = <B, C, B> (added at elements *c[4,5]*, *c[3,3]*, and *c[2,1]*). Alternatively we could start at *c[5,4]* which would produce *Z* = <B, C, A>. Note that the LCS *is not unique* but the optimal length of the LCS *is*.

Questionnaires

Ques.1	List different String Matching algorithms.
Ans	Different string matching algorithms are: i. The Naïve string matching algorithm ii. The Rabin Karp algorithm iii. String matching with finite automata iv. The knuth-Morris-Pratt algorithm v. Longest Common Subsequence algorithm
Ques.2	Define Longest Common Subsequence problem.
Ans	Given two sequences X and Y, the Longest Common Subsequence problem is to find the longest subsequence common to both X and Y.
Ques.3	Write the recurrence formula for solving Longest Common Subsequence problem.
Ans	The recurrence formula for solving Longest Common Subsequence problem is as follows: $$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1][j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_i \\ max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}$$

NAME: HARSHBHAI SOLANKI
ROLL.N0:62
SE-4-D

EXPERIMENT N0-10

AIM: Implementation of Longest Common Subsequence Algorithm.

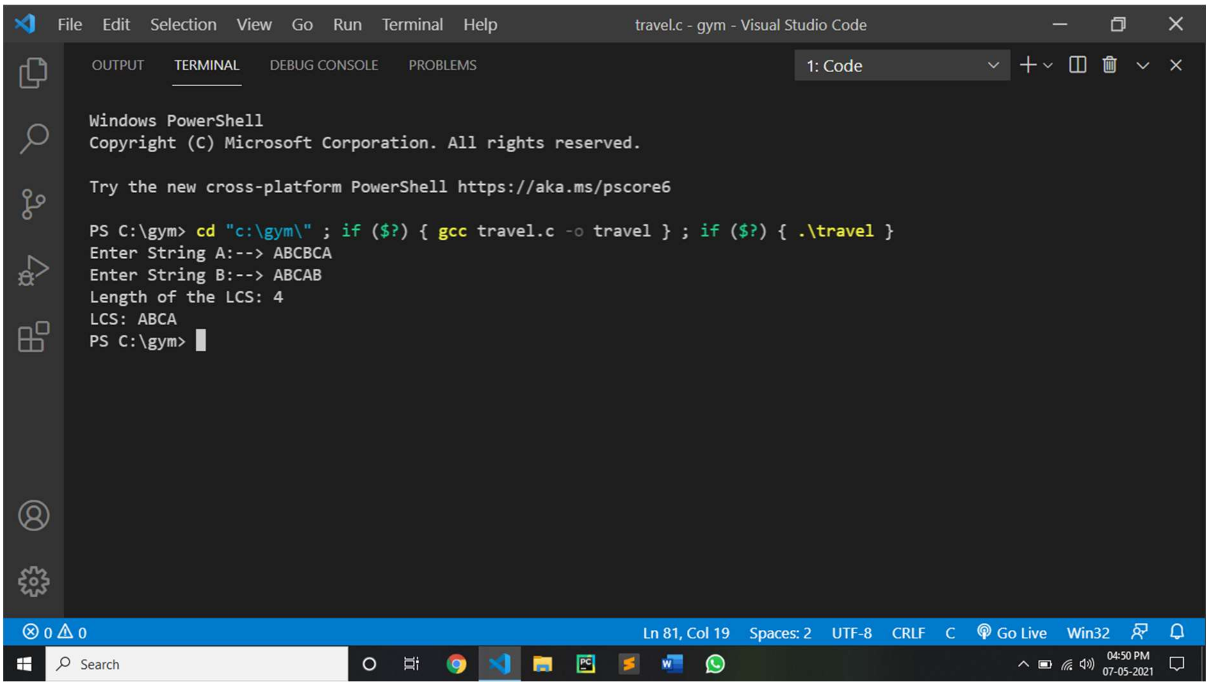
CODE:

```
#include<stdio.h>
#include<string.h>

int max(int a, int b);
void findLCS(char *X, char *Y, int XLen, int YLen);

int max(int a, int b) {
 return (a > b)? a : b;
}

void findLCS(char *X, char *Y, int XLen, int YLen) {
 int L[XLen + 1][YLen + 1];
 int r, c, i;
 for(r = 0; r <= XLen; r++) {

  for(c = 0; c <= YLen; c++) {

   if(r == 0 || c == 0) {

    L[r][c] = 0;

   } else if(X[r - 1] == Y[c - 1]) {
```

```
      L[r][c] = L[r - 1][c - 1] + 1;

    } else {

      L[r][c] = max(L[r - 1][c], L[r][c - 1]);
    }
  }
}

r = XLen;
c = YLen;
i = L[r][c];

char LCS[i+1];
LCS[i] = '\0';

while(r > 0 && c > 0) {

  if(X[r - 1] == Y[c - 1]) {

    LCS[i - 1] = X[r - 1];

    i--;
    r--;
    c--;
```

```
    } else if(L[r - 1][c] > L[r][c - 1]) {

      r--;

    } else {

      c--;

    }

  }

  printf("Length of the LCS: %d\n", L[XLen][YLen]);
  printf("LCS: %s\n", LCS);
}

int main(void) {
   char A[20],B[20];
   printf("Enter String A:--> ");
   scanf("%s",A);
   printf("Enter String B:--> ");
   scanf("%s",B);
  int XLen = strlen(A);
  int YLen = strlen(B);
  findLCS(A,B, XLen, YLen);
  return 0;
}
```

OUTPUT:



CONCLUSION:

By performing the above algorithm we can conclude that:

•Checking membership of one subsequance of P[1…m] into Q[1…n] takes O(n) time. 2^m subsequance are possible for string P of length m.

•So worst case running time of brute force approach would be O(n. 2^m)

•In dynamic programming, the only table of size m*n is filled up using two nested for loops.

•So running time of dynamic programming approach would take O(mn)

•Same thing would consider for space complexity.