AIM:- To implement longest common subsequence.

THEORY:-

A common subsequence of two or more strings is a sequence that appears in the same order in each of the strings, but not necessarily consecutively. For example, given the strings "ABCD" and "ACDF", the common subsequence "ACD" appears in both strings. The longest common subsequence (LCS) of two or more strings is the longest subsequence that is common to all the strings. For example, given the strings "ABCD" and "ACDF", the LCS is "ACD".

To find the LCS of two strings, we can use dynamic programming. We first create a table with one row and one column for each string, and fill in the cells according to the following rules:

If the two characters being compared are the same, the value in the current cell is the value in the diagonal cell plus one.Otherwise, the value in the current cell is the maximum of the value in the cell to the left and the value in the cell above.We start at the top left corner of the table and work our way down to the bottom right corner. The value in the bottom right corner of the table is the length of the LCS.

To find the LCS itself, we can backtrack through the table from the bottom right corner to the top left corner, following the same rules as above. If the value in the current cell is equal to the value in the cell to the left or the cell above, we move in that direction.If the value in the current cell is equal to the value in the diagonal cell plus one, we add the corresponding character to the LCS and move diagonally.

The algorithm for finding the LCS of two strings using dynamic programming involves creating a table to store the length of the LCS of prefixes of the two input strings. This table is initialized with zeros, and then filled in using dynamic programming. The length of the LCS can be read off from the bottom-right corner of the table. To find the actual LCS, we can backtrack through the table starting from this corner, following a path of cells with the same value until we reach the top-left corner. Along the way, we can record the characters corresponding to the cells visited. Finally, we reverse the order of these characters to obtain the final answer. The time complexity of this algorithm is O(nm), where n and m are the lengths of the two input strings, and the space complexity is also O(nm).

Example:

*Say the strings are **S1 = "AGGTAB"** and **S2 = "GXTXAYB"**.*

First step: *Initially create a 2D matrix (say dp[][]) of size 8 x 7 whose first row and first column are filled with 0.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0							
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Creating dp table and filling 0 in 0th row and column

Creating the dp table

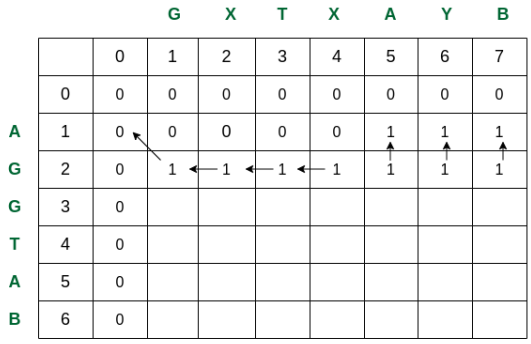
Second step: *Traverse for i = 1. When j becomes 5, S1[0] and S2[4] are equal. So the dp[][] is updated. For the other elements take the maximum of dp[i-1][j] and dp[i][j-1]. (In this case, if both values are equal, we have used arrows to the previous rows).*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0							
G	3	0							
T	4	0							
A	5	0							
B	6	0							

Updating dp table for row 1

Filling the row no 1

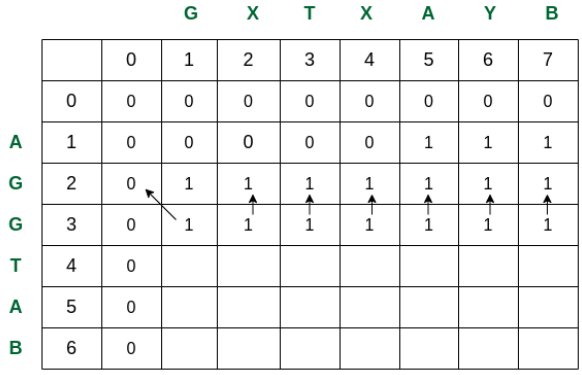
Third step: *While traversed for i = 2, S1[1] and S2[0] are the same (both are 'G'). So the dp value in that cell is updated. Rest of the elements are updated as per the conditions.*



Updating dp table for row 2

Filling the row no. 2

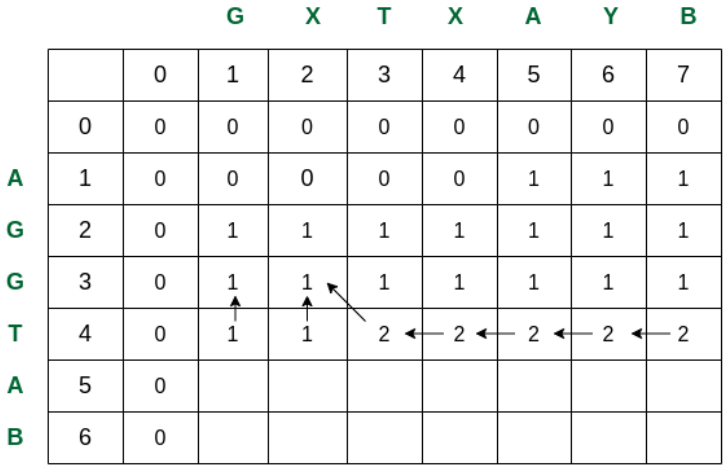
Fourth step: *For i = 3, S1[2] and S2[0] are again same. The updations are as follows.*



Updating dp table for row 3

Filling row no. 3

Fifth step: *For i = 4, we can see that S1[3] and S2[2] are same. So dp[4][3] updated as dp[3][2] + 1 = 2.*



Updating dp table for row 4

Filling row 4

Sixth step: *Here we can see that for i = 5 and j = 5 the values of S1[4] and S2[4] are same (i.e., both are 'A'). So dp[5][5] is updated accordingly and becomes 3.*

			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0							

Updating dp table for row 5

Filling row 5

Final step: *For i = 6, see the last characters of both strings are same (they are 'B'). Therefore the value of dp[6][7] becomes 4.*

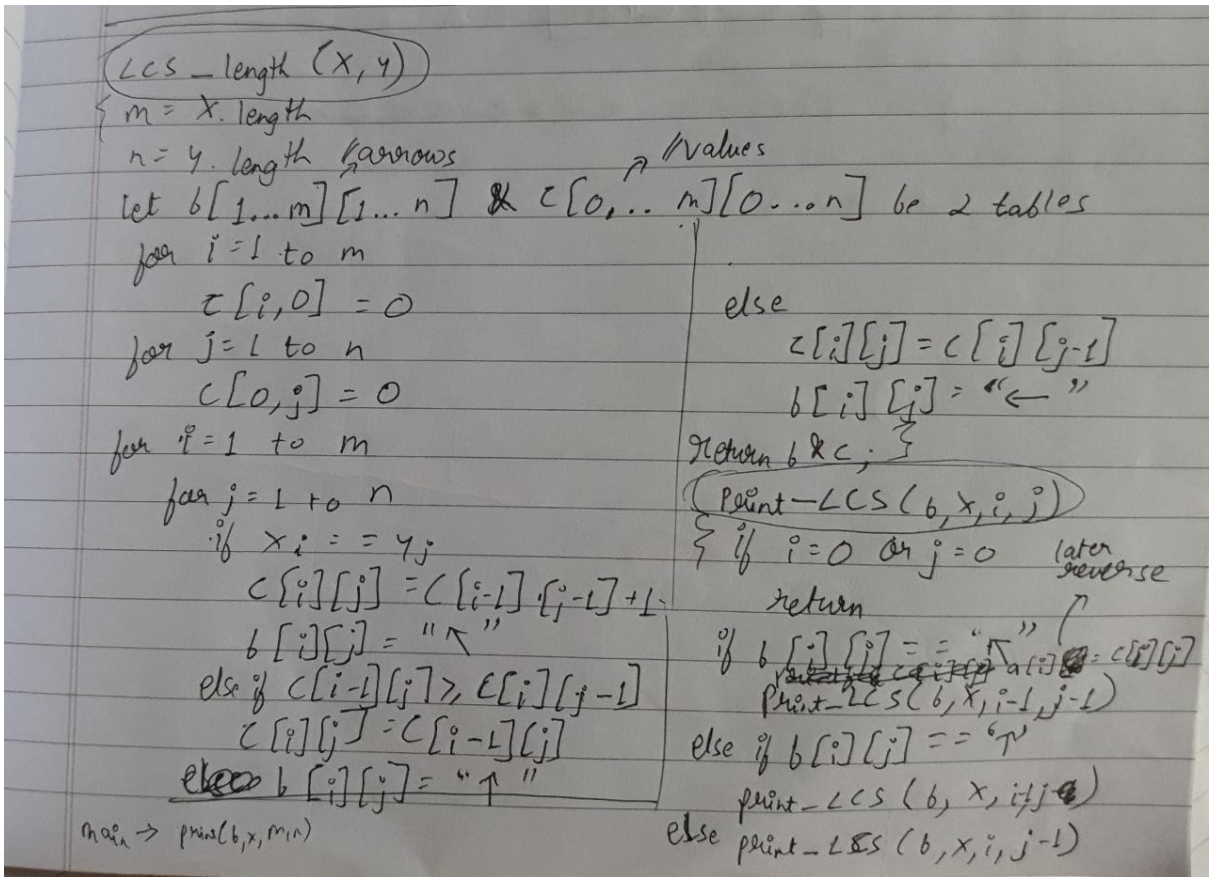
			G	X	T	X	A	Y	B
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
A	1	0	0	0	0	0	1	1	1
G	2	0	1	1	1	1	1	1	1
G	3	0	1	1	1	1	1	1	1
T	4	0	1	1	2	2	2	2	2
A	5	0	1	1	2	2	3	3	3
B	6	0	1	1	2	2	3	3	4

Updating dp table for row 6

Filling the final row

So we get the maximum length of common subsequence as ***4****.*

ALGORITHM:



CODE:-

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

int i, j, m, n, LCS_table[20][20];
char S1[20] , S2[20] , b[20][20];
void input()
{
   printf("Enter two strings");
   scanf("%s",S1);
   scanf("%s",S2);
}
void lcsAlgo() {
```

```
 m = strlen(S1);
 n = strlen(S2);

 // Filling 0's in the matrix
 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--;
  }

  // Printing the sub sequences
  printf("S1 : %s \nS2 : %s \n", S1, S2);
  printf("LCS: %s", lcsAlgo);
}

int main() {
  input();
  lcsAlgo();
  printf("\n");
}
```

OUTPUT:-

```
/tmp/dK3pxAdZmx.o
Enter two stringsABCDEFGH
ABHJK
S1 : ABCDEFGH
S2 : ABHJK
LCS: ABH
```

CONCLUSION:-

The longest common subsequence (LCS) is the longest sequence of characters that appears in the same order in both input strings. The LCS problem can be solved using dynamic programming, which involves creating a table to store the length of the LCS of prefixes of the two input strings. The algorithm has a time complexity of O(nm) and a space complexity of O(nm), where n and m are the lengths of the two input strings.The LCS problem has applications in areas such as bioinformatics, text comparison, and data compression.