Name: Vishal Shashikant Salvi.
UID: 2019230069
Batch: C
Class: SE Comps

Experiment No 7

Aim: To implement Longest common subsequence by using Dynamic Programming.

Theory:

Dynamic Programming:

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds it application in a lot of real-life situations.

In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

Dynamic Programming and Recursion:

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

Fibonacci (n) = 1; if n = 0
Fibonacci (n) = 1; if n = 1
Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: **1, 1, 2, 3, 5, 8, 13, 21...** and so on!

A code for it using pure recursion:

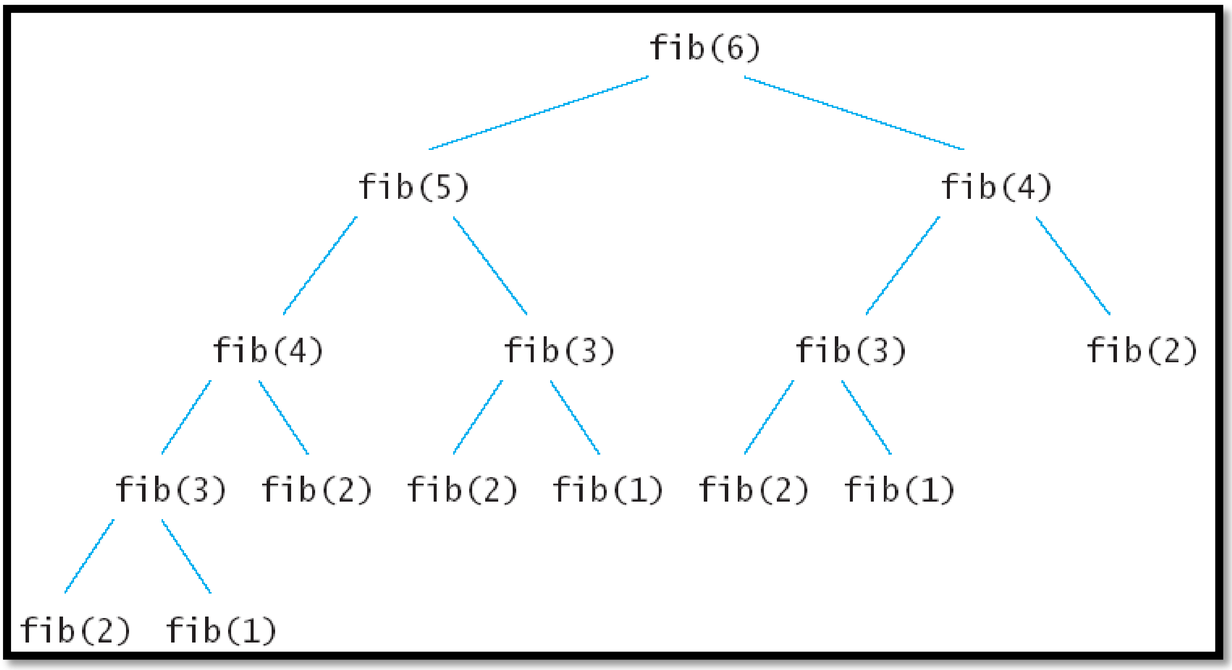
```
int fib (int n) {
    if (n < 2)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

Using Dynamic Programming approach with memoization:

```
void fib () {
      fibresult[0] = 1;
      fibresult[1] = 1;
      for (int i = 2; i<n; i++)
        fibresult[i] = fibresult[i-1] + fibresult[i-2];
  }
```

Are we using a different recurrence relation in the two codes? No. Are we doing anything different in the two codes? Yes.

In the recursive code, a lot of values are being recalculated multiple times. We could do good with calculating each unique quantity only once. Take a look at the image to understand that how certain values were being recalculated in the recursive way:



longest common subsequence (LCS)

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z.

If

S1 = {B, C, D, A, A, C, D}

Then, {A, D, B} cannot be a subsequence of S1 as the order of the elements is not the same (ie. not strictly increasing sequence).

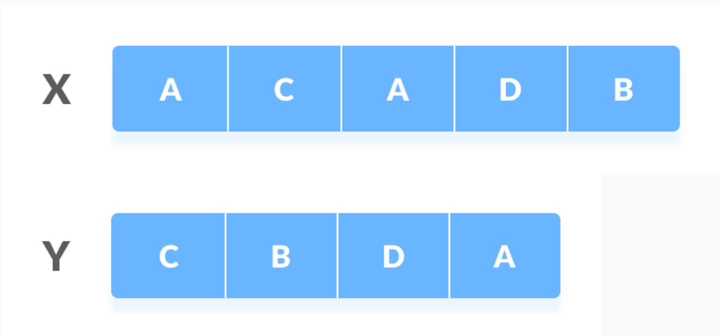
If

S1 = {B, C, D, A, A, C, D} S2 = {A, C, D, B, A, C}

Then, common subsequences are {B, C}, {C, D, A, C}, {D, A, C}, {A, A, C}, {A, C}, {C, D}, ... Among these subsequences, {C, D, A, C} is the longest common subsequence.

Using Dynamic Programming to find the LCS

Let us take two sequences:



The following steps are followed for finding the longest common subsequence.

1. Create a table of dimension n+1*m+1 where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.

		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

2. Fill each cell of the table using the following logic.

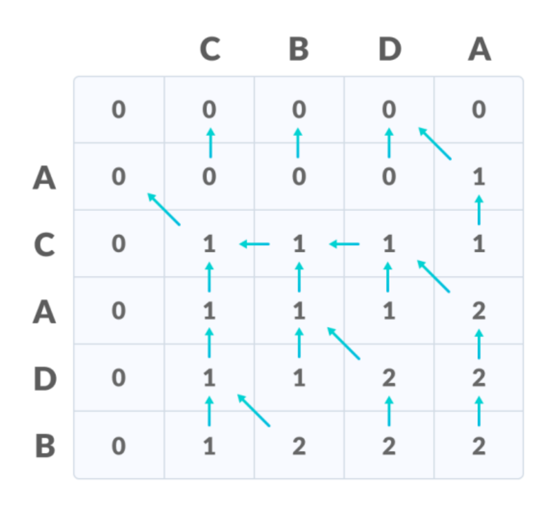
3. If the character correspoding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.

4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value.

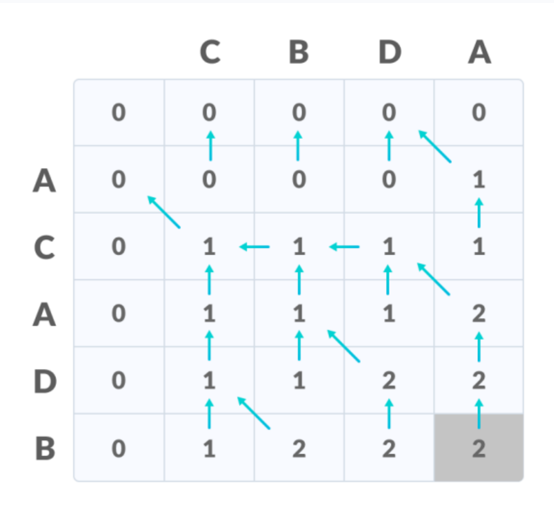
   If they are equal, point to any of them.

		C	B	D	A
	0	0	0	0	0
A	0	0	0	0	1
C	0				
A	0				
D	0				
B	0				

5. **Step 2** is repeated until the table is filled.

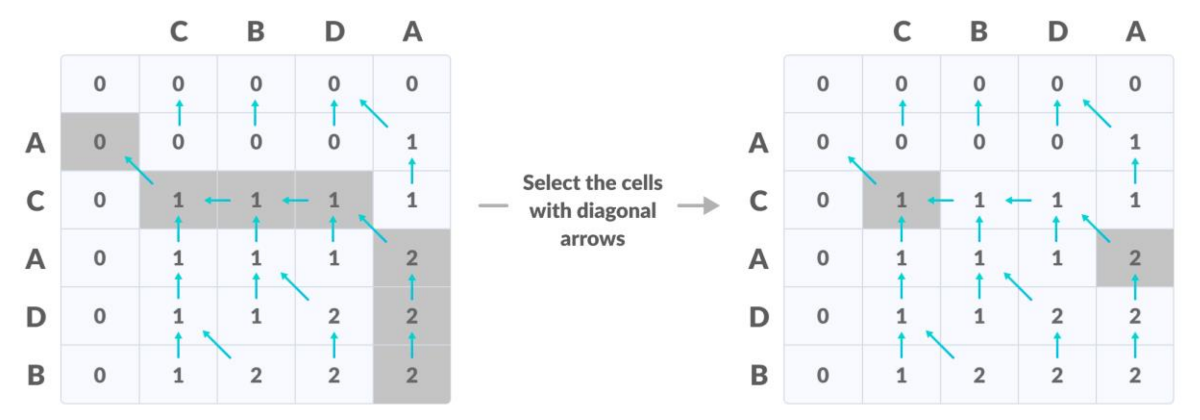


6. The value in the last row and the last column is the length of the longest common subsequence.



7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common

subsequence.



Thus, the longest common subsequence is CD.

How dynamic programming algorithm is more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations. So, the time taken by a dynamic approach is the time taken to fill the table (ie. O(mn)). Whereas, recursion algorithm has the complexity of $2^{\max(m, n)}$.

Algorithm:

Algorithm :

```
void compute_LCS ( char str1[10] , char str2[10] ){
    for ( i=0 ; i<m ; i++){
        C[0][i] = 0
    }
    for (i=0 ; i<n ; i++){
        C[i][0]=0
    }

    for (i= 1 ; i<=m ; i++){
        for (j=1 ; j<=n ; j++){

            if (str 1 [i-1] = = str2 [i-1]){
            // increment value at that
            index and set the value
            of k [i][j] to zere to
            indicate diagonal arrow
            }
        ~~else if (~~

        else {
            if ( c[i-1][j] > c[i][j-1]){
            // set the value of c[i][i]
            according to which is
            greater in previous diagonal
            element
            }
        }
    }
}
```

```
int length (char str1[10], char str2[10], int m, int n
    {
      // program to find length
    }

char string (int i, int j) {

      // recursive function to print
           the LCS from k[i] [j] array
    }
```

Let,

c[i,j] be the length of an LCS of A_i, and B_i then we get a recursive definition

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

From this definition we could build c[i,j] table and d-table in which direction are mentioned.

Analysis:

Analysis of LCS

We have two nested loops
- The outer one interates n time
- The innes one interates m times

- A constant amount of work is done inside each iteration of the inner loop

- Thus, the total running time is $O(nm)$

Space complexity is also $O(nm)$ for $n*m$ table

without using dynamic programming the time complexity would be $O(2^n \cdot n)$
but after using Dynamic programming

Two for loops

$\therefore O(n^2)$

i.e $O(nm)$

Example:

A = < X, Y, Z, Y, T, X, Y >

B = < Y, T, Z, X, Y X >

We have to obtain longest common Subsequence. Arrange elements of A and B in the arrays as -

	1	2	3	4	5	6	7
A [i]	X	Y	Z	Y	T	X	Y
B [j]	Y	T	Z	X	Y	X	

Step 1 : We will build c[i, j] and d[i, j]

Initially c[i, j] ← 0 where i represents row and c[0, j] ← 0 where j represents the column.

Note that c table will store values and d table will store directions.

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0						
2	0						
3	0						
4	0						
5	0						
6	0						
7	0						

Then

for (i ← 1 to m) and
for (j ← 1 to n)

we go on filling up c[i, j] and d[i, j] horizontally

∴ Let i = 1, j = 1

	1	2	3	4	5	6	7
A[i]	X	Y	Z	Y	T	X	Y
B[j]	Y	T	Z	X	Y	X	

j →

i ↑

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	$\boxed{0 \uparrow}$					
2	0						
3	0						
4	0						
5	0						
6	0						
7	0						

As $A[i] \neq B[j]$

If $(C[0,1] \geq C[1,0])$ is true

$\therefore \quad 0 \geq 0$

$C[1,1] \leftarrow C[i-i, j]$

$\therefore C[1,1] \leftarrow C[0,1]$

$\therefore C[1,1] \leftarrow 0$

and $d[1,1] \leftarrow \uparrow$

Step 2 :

Let $i = 1$, $j = 2$ then

	1	2	3	4	5	6	7
A [i]	$\boxed{X}$	Y	Z	Y	T	X	Y
B [j]	Y	$\boxed{T}$	Z	X	Y	X	

As $A[i] \neq B[j]$

then if $(C[i-1, j] \geq C[i, j-1]) \rightarrow$ is true

$\therefore C[0,2] \geq C[1,1]$

i.e. $0 \geq 0$

Hence $C[i,j] \leftarrow C[i-1,j]$

i.e $d[i,j] \leftarrow$ "$\uparrow$"

we get $C[1,2] \leftarrow C[0,2]$ i.e 0

and $d[i,j] \leftarrow$ "$\uparrow$"

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0↑	$\boxed{0\uparrow}$				
2	0						
3	0						
4	0						
5	0						
6	0						
7	0						

Continuing in this fashion we can fill up the table as follows-

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0↑	0↑	0↑	↖1	←1	↖1
2	0	↖1	←1	←1	1↑	↖2	←2
3	0	1↑	1↑	↖2	←2	2↑	2↑
4	0	↖1	1↑	2↑	2↑	↖3	←3
5	0	1↑	↖2	2↑	2↑	3↑	3↑
6	0	1↑	2↑	2↑	↖3	3↑	↖4
7	0	↖1	2↑	2↑	3↑	↖4	4↑

Now we can construct a longest common Subsequence using c[i, J] and d[i,j]

Step 4: Constructing LCS

To decide the LCS, we have to make use of d - table. The algorithm for this task is as show below -

```
Algorithm : display-LCS (d, A [ ], i, j)
{
    if ( i = 0 || j = 0 ) then
        return
    if (d [i, j] = "↖" then
    {
        display LCS (d, A, i-1, j-1)
        print (ai)
    }
        else if (d [i,j] = "↑" then
        {
            display - LCS (d, A, i-1, j)
        }
    else
        display - LCS (d, A, i, j-1)
}
```

Let us apply this algorithm for obtaining LCS. Initially a call is given as:
display - LCS (d, A[ ], 7, 6)

Here 7 is upper bound of i and 6 is upper bound of j.

As d [7, 6] = ↑ we get recursive call display - LCS (d, A, 6, 6.)

Then [d [6, 6] = ↖] a recursive call display. LCS (d, A, 5, 5) is encountered

Then the algorithm tells us to print value of a_i which is a_6

Thus if we follow the complete algorithm we get YZYX as LCS.

Ans: YZYX

Code:

```
#include<stdio.h>
#include<string.h>
#define SIZE 20

void compute_LCS(char A[],char B[])
{
        void display(char[][SIZE],char[],int,int);
        int m,n,i,j;
        int c[SIZE][SIZE];
        char d[SIZE][SIZE];
        int b;
        int k[10][10];

        m = strlen(A);
        n = strlen(B);


        for(i=0;i<=m;i++)
                c[i][0]=0;
        for(i=0;i<=n;i++)
                c[0][i]=0;

        for(i=1;i<=m;i++)
        {
                for(j=1;j<=n;j++)
                {
                if(B[i-1] == A[j-1])
                {
                                c[i][j] = c[i-1][j-1] + 1;
                                b = 0;
                                k[i][j] = b;
                                printf(" %d",k[i][j]); // 0
                }
                else {
                                if(c[i-1][j] > c[i][j-1]){
                                c[i][j] = c[i-1][j];
                                b = 2;
                                k[i][j] = b;
```

```
                printf(" %d",k[i][j]); // 2
        }
        else{
                c[i][j] = c[i][j-1];
                b = 1;
                k[i][j] = b;
                printf(" %d",k[i][j]); // 1
                }
            }
        }
    printf("\n");
}


    for(i=1;i<=m;i++)
        {
            for(j=1;j<=n;j++)
                {
                    if(A[i-1] == B[j-1])
                    {
                        c[i][j]=c[i-1][j-1]+1;
                        d[i][j]= '\\';
                    }
                    else if(c[i-1][j]>=c[i][j-1])
                    {
                        c[i][j]=c[i-1][j];
                        d[i][j]= '^';
                    }
                    else
                    {
                        c[i][j]=c[i][j-1];
                        d[i][j]= '<';
                    }
                }

        }

        printf("\n Length Calculation of LCS:\n");
        for(i=0;i<=m;i++)
```

```
{
        for(j=0;j<=n;j++)
                {
                        printf("%3d",c[i][j]);
                }
                printf("\n");
}
printf("\n The Longest Common Subsequence is :");
display(d,A,m,n);

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

int length(char A[10], char B[10], int m, int n )
{
        if(m == 0 || n == 0)
        {
                return 0;
        }
        if(A[m-1] == B[n-1])
        {
                return 1 + length(A, B, m-1, n-1);
        }
        else
        {
                return max(length(A, B, m, n-1), length(A, B, m-1, n));
        }
}

char string(int i,int j)
```

```
            {
                if(i==0 || j==0)
                {
                    return;
                }
                if(k[i][j] == 0)
                    {
                        printf("%c",B[i-1]);
                        string(i-1,j-1);
                    }
                else if(k[i][j] == 2)
                    {
                        string(i-1,j);
                    }
                else{
                        string(i,j-1);
                    }
        }
            printf("\nLength of LCS is %d\n", length(A, B, m, n) );




}

void display(char d[][20],char A[], int i, int j)
{
    if(i==0 || j==0)

        return;

    if(d[i][j]=='\\')
    {
        display(d,A,i-1,j-1);
        printf("%c",A[i-1]);
    }
    else if(d[i][j]=='^')
    {
```

```
			display(d,A,i-1,j);
		}
		else
		{
			display(d,A,i,j-1);
		}
}

void main()
{
		int i,m,j,n,c[SIZE][SIZE];
		char A[SIZE],B[SIZE];
		printf("Enter 1st sequence:");
		scanf("%s",A);
		printf("Enter 2nd sequence:");
		scanf("%s",B);
		printf("Matrix showing direction of arrow:\n1. 0 for diagonal\n2. 1 for left\n3. 2 for up\n");
		compute_LCS(A,B);
}
```

Output:

```
C:\Users\Vishal\Desktop\Longest.exe
Enter 1st sequence:XYZYTXY
Enter 2nd sequence:YTZXYX
Matrix showing direction of arrow:
1. 0 for diagonal
2. 1 for left
3. 2 for up
 1 0 1 0 1 1
 1 2 1 1 0 1
 1 2 0 1 1 1
 0 1 2 1 1 0
 2 0 1 0 1 1
 0 2 1 2 1 0
 2 2 1 2 1 2

 Length Calculation of LCS:
  0   0   0   0   0   0   0
  0   0   0   0   1   1   1
  0   1   1   1   1   2   2
  0   1   1   2   2   2   2
  0   1   1   2   2   3   3
  0   1   2   2   2   3   3
  0   1   2   2   3   3   4
  0   1   2   2   3   4   4

 The Longest Common Subsequence is :YZYX
Length of LCS is 4

--------------------------------
Process exited after 41.67 seconds with return value 20
Press any key to continue . . .
```

Conclusion:

The longest common subsequence (LCS) problem is the problem of finding the longest subsequence that is present in given two sequences in the same order. i.e. find a longest sequence which can be obtained from the first original sequence by deleting some items, and from the second original sequence by deleting other items.

Also, dynamic programming algorithms is more efficient than recursive algorithm while solving the LCS problem because it reduces the function calls.