PRACTICAL-5

Aim: Dynamic Programming

Software Requirement: Code Blocks, Visual Studio code

Hardware Requirement: Desktop Computer

5.1

Aim: Implement a program which has BNMCOEF() function that takes two parameters n and k and returns the value of Binomial Coefficient C(n, k). Compare the dynamic programming implementation with recursive implementation of BNMCOEF(). (In output, entire table should be displayed.)

Test Case	n	k
1	5	2
2	11	6
3	12	5

THEORY:-

- The Binomial coefficients are the positive integers that occur as coefficients in the binomial theorem.
- Computing binomial coefficients problem can be solved using dynamic programming.
- Binomial coefficients are represented by $C(n, k)$ or $\binom{n}{k}$ and can be used to represent the coefficients of a binomial:

  $(a + b)^n = C(n, 0)a^n + ... + C(n, k)a^{n-k}b^k + ... + C(n, n)b^n$

- The recursive relation is defined by the prior power

  $C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$ for $n > k > 0$
  IC $C(n, 0) = C(n, n) = 1$

Computing a Binomial Coefficient

Computing binomial coefficients is non-optimization problem but can be solved using dynamic programming.

Calculating binomial coefficients can be important for solving combinatorial problems. A formula for computing binomial coefficients is this:

$$\binom{n}{m} = \frac{n!}{(n-m)!m!}$$

Using an identity called Pascal's Formula a recursive formulation for it looks like this:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \\ 1 & \text{if } n = m \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{otherwise} \end{cases}$$

Algorithm

Step 1 : Get the two inputs, the positive value of n and the non-positive value of k which denotes the k-th binomial coefficient in the Binomial Expansion.
Step 2 : Allocate the array of size k + 1 with the value of 1 at 0-th index and rest with value 0.
Step 3 : Next, generating the sequence of pascal's triangle, with the first row containing single element valued 1 which was already created in step 2.
Step 4 : Further next consecutive rows of pascal's triangle are computed from the previous row by adding the two consecutive elements, but step 4 is to be carried out upto k-times, for enclosing n-value times.
Step 5 : Stop.

CODE:-

```
#include<bits/stdc++.h>

using namespace std;


//Utility function that returns minimum number.

int min(int a,int b){
    return (a<b)?a:b;
}


int BinomialCoeff(int n,int k){
    int c[n+1][k+1];
    int i,j;

    //dp table for overlapping subproblem
    for(i=0;i<=n;i++){
        for(j=0;j<=min(i,k);j++){
            if(j==0||j==i){
                c[i][j]=1;
            }
            else
            c[i][j]=c[i-1][j-1] + c[i-1][j];
        }
    }

    return c[n][k];
}

int main(){

    //For dynamic

    // int n,k;
    // cout<<"\nEnter value of N:";cin>>n;
    // cout<<"\nEnter value of K:";cin>>k;

    // cout<<"\nValue of C["<<n<<"]["<<k<<"] is "<<BinomialCoeff(n,k);

    //Static test cases
    const char separator = ' ';
    const int nameWidth = 15;
    const int numWidth = 15;
    int x[3] = {5,11,12};
    int y[3]={2,6,5};
   int n=sizeof(x)/sizeof(x[0]);
```

```
   cout<<"Answer is:\n\n";

    cout << left << setw(nameWidth) << setfill(separator) << "I";
     cout << left << setw(nameWidth) << setfill(separator) << "N";
     cout << left << setw(nameWidth) << setfill(separator) << "K";
     cout << left << setw(nameWidth) << setfill(separator) << "C[N][K]"<<"\n";

    for(int i=0;i<n;i++){
        cout << left << setw(nameWidth) << setfill(separator) << i;
     cout << left << setw(nameWidth) << setfill(separator) << x[i];
     cout << left << setw(nameWidth) << setfill(separator) << y[i];
     cout << left << setw(nameWidth) << setfill(separator) << BinomialCoeff(x[
i],y[i])<<"\n";
    }

    return 0;
}
```

OUTPUT:-

```
F:\Sem5\DAA\Practicals\prac5>g++ -std=c++11 5.1.cpp -o 51

F:\Sem5\DAA\Practicals\prac5>51.exe
Answer is:

I               N               K               C[N][K]
0               5               2               10
1               11              6               462
2               12              5               792
```

Conclusion: -

I have successfully performed the given problem using 'Dynamic programming Approach' and had taken out the output of the given test cases.

The Time and Space Complexities of code by using Dynamic Programming Approach are :

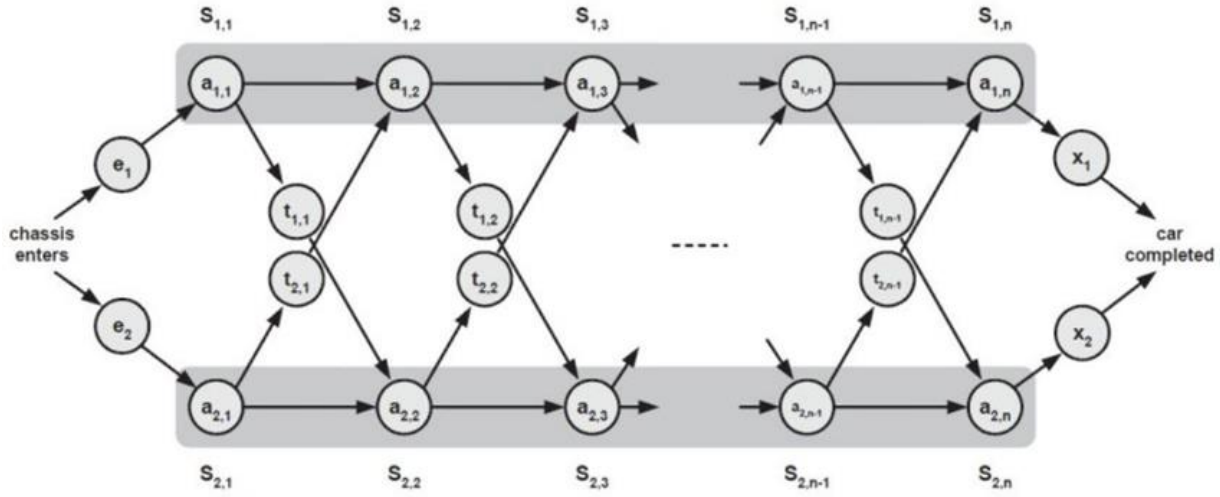
Time complexity: O(n*k)
Space complexity: O(k)

5.2

Aim: Implement and analyse Assembly Line Scheduling.

Theory:

The main goal of solving the Assembly Line Scheduling problem is to determine which stations to choose from line 1 and which to choose from line 2 in order to minimize assembly time.



In the above the diagram we have two main assembly line consider as LINE 1 and LINE 2. Parameters are:

- **S[i,j]**: The jth station on ith assembly line
- **a[i,j]**: Time required at station S[i,j], because every station has some dedicated job that needs to done.
- **e[i]**: Entry time of product on assembly line i [here i= 1,2]
- **x[i]**: Exit time from assembly line i
- **t[i,j]**: Time required to transit from station S[i,j] to the other assembly line.

Hence, the input consists of:

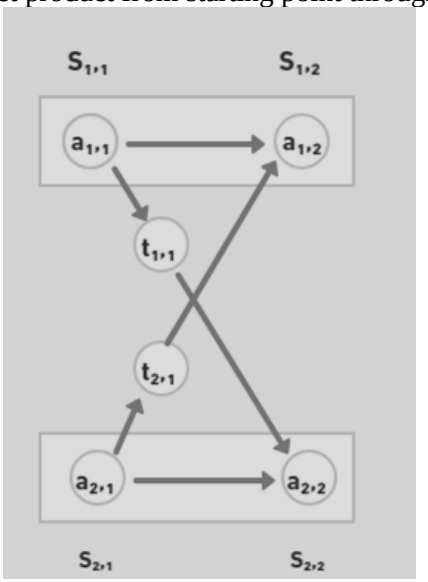
S[i,j] = j-th station on i-th assembly line
a[i,j] = Time required do task at station S[i,j]
e[i] = Entry time of product on assembly line i [here i= 1,2]
x[i] = Exit time from assembly line i
t[i,j] = Time required to transit from station S[i,j] to the other assembly line

Normally, when a raw product enters an assembly line it passes through through that line only. Therefore, the time to go from one station to another station of same assembly line is **negligible** but to pass a partially completed product from one station of a line to other station of another line a **t[i,j]** is taken.
It is a optimization problem in which we have to optimized the total assembly time. To solve the optimization problem we make use of Dynamic Programming. To solve with DP we will see how it have Overlapping Subproblem and Optimal Substructure.

Consider that **f[i,j]** denotes the fastest time taken to get the partially completed product from starting point through ith line and jth station.

The DP structure is as follows:

f[i, j] = minimum time taken to get product from starting point through i-th line and j-th station



Consider the above image, we can reach **S[1,j]** in two ways, either from station **S[1,j-1]** or from station **S[2,j-1]**. We have to find the minimum time from both the ways:-

- Time taken to reach from 1st line will be, **f[1,j-1] + a[1,j]**.
- Time taken to reach from 2nd line will be, **f[2,j-1] + t[2,j-1] + a[1,j]**.

If the partial product comes from S[2,j-1], additionally it incurs the transfer cost to change the assembly line (like t[2,j-1]).

Note that, minimum time to leave S1[j-1] and S2[j-1] have already been calculated.
We can notice from this example that the approach is using the already solved time from f[1,j-1], so the **Overlapping Subproblem** is used. And to find the fastest way through station j, we solve the sub-problems of finding the fastest way through station j-1 which is the **Optimal Substructure**.

Breaking in smaller sub-problems:

We will use the **Bottom-Up approach** to find the minimum time to complete a product and for that consider f1 & f2 be time taken from line 1 & 2 respectively and *'e'* & *'x'* are the entry & exiting time respectively. The following are the cases that we will use:

- **Base-Case:** At station 1 at partial product directly comes from entry point, therefore **f1[1]= e1 + a[1,1]** and **f2[1]= e2 + a[2,1]**.
- For calculating time to reach at jth station from line 1 we will find the optimal time as we discussed above, **f[1,j]= min{f1[j-1]+a[i,j], f2[j-1]+t[2,j-1]+a[1,j]}**
- The optimized or fastest time to exit a completed product will be **f *optimal*= min{f1[n] + x1, f2[n] + x2}**.

We need two tables to store the partial results calculated for each station in an assembly line. The table will be filled in a bottom-up fashion.

The recursion to reach the station j in assembly line i are as follows:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min\{f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\} & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min\{f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\} & \text{if } j \geq 2 \end{cases}$$

Algorithm:

where *'a'* denotes the assembly costs, *'t'* denotes the transfer costs, *'e'* denotes the entry costs, *'x'* denotes the exit costs and *'n'* denotes the number of assembly stages.

START

f1[1] = e1 + a1,1

f2[1] = e2 + a2,1

for j = 2 to n

if ((f1[j − 1] + a1,j) ≤ (f2[j − 1] + t2,j−1 + a1,j)) then

f1[j] = f1[j − 1] + a1,j

else

f1[j] = f2[j − 1] + t2,j−1 + a1,j

if ((f2[j − 1] + a2,j) ≤ (f1[j − 1] + t1,j−1 + a2,j)) then

f2[j] = f2[j − 1] + a2,j

else

f2[j] = f1[j − 1] + t1,j−1 + a2,j

end for

if (f1[n] + x1 ≤ f2[n] + x2) then

fopt = f1[n] + x1

else

fopt = f2[n] + x2

END

CODE:

```
#include<bits/stdc++.h>
using namespace std;

int assemblyLineScheduling(int n,vector<int> entry,vector<int> exit,vector<vec
tor<int> > processing,vector<vector<int> > transfer){

    vector<vector<int> > dp(2,vector<int>(n+1));
```

```
    int i;

    //Initialization and entry to first station
    dp[0][0]=entry[0]+processing[0][0];
    dp[1][0]=entry[1]+processing[1][0];

    for(i=1;i<n;i++){
        //for being on station i of assembly line 1
        dp[0][i]=min(dp[0][i-1],dp[1][i-1]+transfer[1][i-1])+processing[0][i];

        //for being on station i of assembly line 2
        dp[1][i]=min(dp[1][i-1],dp[0][i-1]+transfer[0][i-1])+processing[1][i];
    }

    //exiting from the last station
    dp[0][n]=dp[0][n-1]+exit[0];
    dp[1][n]=dp[1][n-1]+exit[1];

    return min(dp[0][n],dp[1][n]);

}

int main(){

    int i,n;
    vector<int> entry(2),exit(2);

    cout<<"Enter the number of stations:";
    cin>>n;

    vector<vector<int> > processing(2,vector<int>(n));
    vector<vector<int> >transfer(2,vector<int> (n-1));

    cout<<"Enter the entry time for assembly line 1 & 2:\n";
    cin>>entry[0]>>entry[1];

    cout<<"Enter the exit time for assembly line 1 & 2:\n";
    cin>>exit[0]>>exit[1];

    cout<<"Enter the processing time at all station of assembly line 1:\n";
    for(i=0;i<n;i++)
        cin>>processing[0][i];

    cout<<"Enter the processing time at all station of assembly line 2:\n";
    for(i=0;i<n;i++)
        cin>>processing[1][i];
```

```
    cout<<"Enter the transfer time from each station of assembly line 1 to nex
t station of assembly line 2:\n";
    for(i=0;i<n-1;i++)
        cin>>transfer[0][i];

    cout<<"Enter the transfer time from each station of assembly line 2 to nex
t station of assembly line 1:\n";
    for(i=0;i<n-1;i++)
        cin>>transfer[1][i];


    cout<<"The minimum time required to get all the jobs done is: "
        <<assemblyLineScheduling(n,entry,exit,processing,transfer)<<"\n";


    return 0;
}
```

OUTPUT:-

```
F:\Sem5\DAA\Practicals\prac5>g++ -std=c++11 5.2.cpp -o 52

F:\Sem5\DAA\Practicals\prac5>52.exe
Enter the number of stations:3
Enter the entry time for assembly line 1 & 2:
3 1
Enter the exit time for assembly line 1 & 2:
3 3
Enter the processing time at all station of assembly line 1:
8 4 6
Enter the processing time at all station of assembly line 2:
5 7 5
Enter the transfer time from each station of assembly line 1 to next station of assembly line 2:
2 2
Enter the transfer time from each station of assembly line 2 to next station of assembly line 1:
1 3
The minimum time required to get all the jobs done is: 20
```

Conclusion: -

I have successfully performed the given problem using ‘Dynamic programming Approach’ and had taken out the output of the given test cases.

I have learned how factories schedule their work using these algorithms.

5.3

Aim: Given a chain < A1, A2…, An> of n matrices, where for i=1,2,…,n matrixAi with dimensions. Implement the program to fully parenthesize the product A1,A2,…,An in a way that minimizes the number of scalar multiplications. Also calculate the number of scalar multiplications for all possible combinations of matrices.

Test Case	n	Matrices with dimensions
1	3	A1: 3*5, A2: 5*6, A3: 6*4
2	6	A1: 30*35, A2: 35*15, A3: 15*5, A4: 5*10, A5: 10*20, A6: 20*25

Theory:

Let $A_{i,j}$ be the result of multiplying matrices i through j. It can be seen that the dimension of $A_{i,j}$ is p_{i-1} x p_j matrix.

Dynamic Programming solution involves breaking up the problems into subproblems whose solution can be combined to solve the global problem.

At the greatest level of parenthesization, we multiply two matrices

$A_{1.....n}=A_{1....k}$ x $A_{k+1....n})$

Thus we are left with two questions:

- How to split the sequence of matrices?
- How to parenthesize the subsequence $A_{1.....k}$ and$A_{k+1.....n}$?

One possible answer to the first question for finding the best value of 'k' is to check all possible choices of 'k' and consider the best among them. But that it can be observed that checking all possibilities will lead to an exponential number of total possibilities. It can also be noticed that there exists only O (n^2) different sequence of matrices, in this way do not reach the exponential growth.

Step1: Structure of an optimal parenthesization: Our first step in the dynamic paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from an optimal solution to subproblems.

Let $A_{i....j}$ where i≤ j denotes the matrix that results from evaluating the product

A_i A_{i+1}....A_j.

If i < j then any parenthesization of the product A_i A_{i+1} .....A_j must split that the product between A_k and A_{k+1} for some integer k in the range i ≤ k ≤ j. That is for some value of k, we first compute the matrices $A_{i.....k}$ & $A_{k+1....j}$ and then multiply them together to produce the final product $A_{i....j}$. The cost of computing

$A_{i....k}$ plus the cost of computing $A_{k+1....j}$ plus the cost of multiplying them together is the cost of parenthesization.

Step 2: A Recursive Solution: Let m [i, j] be the minimum number of scalar multiplication needed to compute the matrix$A_{i....j}$.

If i=j the chain consist of just one matrix $A_{i....i}=A_i$ so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between A_k and A_{k+1} where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts $A_{i....k}$ and $A_{k+1....j}$+ cost of multiplying them together. We know A_i has dimension p_{i-1} x p_i, so computing the product $A_{i....k}$ and $A_{k+1....j}$takes $p_{i-1}\ p_k\ p_j$ scalar multiplication, we obtain

$$m\ [i,j] = m\ [i, k] + m\ [k + 1, j] + p_{i-1}\ \ p_k\ p_j$$

There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

So the minimum cost of parenthesizing the product $A_i\ A_{i+1}.....A_j$ becomes

$$m\ [i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\{m\ [i,k] + m\ [k+1,j] + p_{i-1}\ p_k p_j\} & \text{if } i < j \\ i\ \le k < j & \end{cases}$$

To construct an optimal solution, let us define s [i,j] to be the value of 'k' at which we can split the product $A_i\ A_{i+1}$ .....A_j To obtain an optimal parenthesization i.e. s [i, j] = k such that

$$m\ [i,j] = m\ [i, k] + m\ [k + 1, j] + p_{i-1}\ \ p_k\ p_j$$

Algorithm:

MATRIX-CHAIN-ORDER (p)

1. n length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ← ∞
8. for k ← i to j-1
9. do q ← m [i, k] + m [k + 1, j] + $p_{i-1}\ p_k\ p_j$
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.

PRINT-OPTIMAL-PARENS (s, i, j)

1. if i=j
2. then print "A"

3. else print "("
4. PRINT-OPTIMAL-PARENS (s, i, s [i, j])
5. PRINT-OPTIMAL-PARENS (s, s [i, j] + 1, j)
6. print ")"

CODE:-

```
#include <stdio.h>
#include<limits.h>
#define INFY 999999999
long int m[20][20];
int s[20][20];
int p[20],i,j,n;


void matmultiply(void)//filling the M table
{
long int q;
int k;
for(i=n;i>0;i--)
 {
   for(j=i;j<=n;j++)
    {
     if(i==j)
       m[i][j]=0;
     else
       {
        for(k=i;k<j;k++)
        {
         q=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
         if(q<m[i][j])
          {
            m[i][j]=q;
            s[i][j]=k;
          }
         }
        }
      }
 }
}

//parenthisis
void print_optimal(int i,int j)
{
    if (i == j)
        printf(" A%d ",i);
    else
       {
```

```
          printf("( ");
          print_optimal(i, s[i][j]);
          print_optimal(s[i][j] + 1, j);
          printf(" )");
       }
}

int MatrixChainOrder(int p[], int i, int j)//to filter the minimum multiplicat
ion out of diffrent values of k
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    for (k = i; k <j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

int main()
{
    int k;
    printf("Enter the no. of matrices: ");
    scanf("%d",&n);
        for(i=1;i<=n;i++)
            for(j=i+1;j<=n;j++)
                {
                 m[i][i]=0;
                 m[i][j]=INFY;//initializing elements with some fixed large va
lue
                 s[i][j]=0;
                }

    printf("\nEnter the dimensions: \n");

    for(k=0;k<=n;k++)
    {
```

```
     printf("P%d: ",k);
     scanf("%d",&p[k]);
    }

    matmultiply();

    printf("\nCost Matrix M:\n");
    for(i=1;i<=n;i++)
     for(j=i;j<=n;j++)
      printf("m[%d][%d]: %ld\n",i,j,m[i][j]);

    i=1,j=n;
    printf("\nMultiplication Sequence : ");
    print_optimal(i,j);
    printf("\nMinimum number of multiplications is : %d ",MatrixChainOrder(p,
1, n));

}
```

OUTPUT:-

```
F:\Sem5\DAA\Practicals\prac5>gcc 5.3.c -o 53

F:\Sem5\DAA\Practicals\prac5>53.exe
Enter the no. of matrices: 3

Enter the dimensions:
P0: 3
P1: 5
P2: 6
P3: 4

Cost Matrix M:
m[1][1]: 0
m[1][2]: 90
m[1][3]: 162
m[2][2]: 0
m[2][3]: 120
m[3][3]: 0

Multiplication Sequence : ( (  A1  A2  ) A3  )
Minimum number of multiplications is : 162
```

```
F:\Sem5\DAA\Practicals\prac5>53.exe
Enter the no. of matrices: 6

Enter the dimensions:
P0: 30
P1: 35
P2: 15
P3: 5
P4: 10
P5: 20
P6: 25

Cost Matrix M:
m[1][1]: 0
m[1][2]: 15750
m[1][3]: 7875
m[1][4]: 9375
m[1][5]: 11875
m[1][6]: 15125
m[2][2]: 0
m[2][3]: 2625
m[2][4]: 4375
m[2][5]: 7125
m[2][6]: 10500
m[3][3]: 0
m[3][4]: 750
m[3][5]: 2500
m[3][6]: 5375
m[4][4]: 0
m[4][5]: 1000
m[4][6]: 3500
m[5][5]: 0
m[5][6]: 5000
m[6][6]: 0

Multiplication Sequence : ( (  A1 (  A2  A3  ) )( (  A4  A5  ) A6  ) )
Minimum number of multiplications is : 15125
```

Conclusion: -

I have solved this problem using Dynamic Programming. First I have computed results for sub-chains of the original chain and store these results in a 2-D array. Then I have used these results to compute results for larger chains.

Time Complexity of this solution is O(n^3).

5.4

Aim: Implement a program to print the longest common subsequence for the following strings:

Test Case	String1	String2
1	ABCDAB	BDCABA
2	EXPONENTIAL	POLYNOMIAL
3	LOGARITHM	ALGORITHM

Theory:

Let X = < x_1, x_2, x_3,…, x_m > and Y = < y_1, y_2, y_3,…, y_n > be the sequences. To compute the length of an element the following algorithm is used.

In this procedure, table **C[m, n]** is computed in row major order and another table **B[m,n]** is computed to construct optimal solution.

Algorithm:

LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
  C[i, 0] := 0
for j = 1 to n do
  C[0, j] := 0
for i = 1 to m do
  for j = 1 to n do
    if xi = yj
      C[i, j] := C[i - 1, j - 1] + 1
      B[i, j] := 'D'
    else
      if C[i -1, j] ≥ C[i, j -1]
        C[i, j] := C[i - 1, j] + 1
        B[i, j] := 'U'
      else
      C[i, j] := C[i, j - 1]
      B[i, j] := 'L'
return C and B
```

Print-LCS (B, X, i, j)

```
if i = 0 and j = 0
  return
if B[i, j] = 'D'
```

Print-LCS(B, X, i-1, j-1)
Print(x_i)
else if B[i, j] = 'U'
Print-LCS(B, X, i-1, j)
else
Print-LCS(B, X, i, j-1)

This algorithm will print the longest common subsequence of **X** and **Y**.

CODE:-

```
#include<bits/stdc++.h>
using namespace std;

string lcsubsequence(string a, string b)
{
    int n=a.size(),m=b.size();
    int matrix[n+1][m+1];
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=m;j++)
        {
            matrix[i][j]=0;
        }
    }
    for(int i=0;i<=n;i++)
    {
        for(int j=0;j<=m;j++)
        {
            if(i==0 || j==0)
            {
                matrix[i][j]=0;
            }
            else if(a[i-1]==b[j-1])
            {
                matrix[i][j]=matrix[i-1][j-1]+1;
            }
            else
            {
                matrix[i][j]=max(matrix[i-1][j],matrix[i][j-1]);
            }
        }
    }
    //to print table
    // cout<<"   x ";
    // for(int i=0;i<m;i++)
    // {
    //  cout<<b[i]<<" ";
    // }
```

```
    // cout<<endl;
    // cout<<"y  ";
    // for(int j=0;j<=m;j++)
    // {
    //  cout<<matrix[0][j]<<" ";
    // }
    // cout<<endl;
    // for(int i=1;i<=n;i++)
    // {
    //  cout<<a[i-1]<<"  ";
    //  for(int j=0;j<=m;j++)
    //  {
    //      cout<<matrix[i][j]<<" ";
    //  }
    //  cout<<endl;
    // }
    // cout<<"\n\nlongest Common Subsequence's Length : "<<matrix[n][m]<<endl;

    int vrc=n,hrc=m;
    string ans;
    while(vrc>0|| hrc>0)
    {
        if(matrix[vrc-1][hrc]==matrix[vrc][hrc-1] && matrix[vrc][hrc-
1]==matrix[vrc][hrc])
        {
            vrc--;
        }
        else if(matrix[vrc][hrc-1]==matrix[vrc][hrc])
        {
            hrc--;
        }
        else if(matrix[vrc-1][hrc]==matrix[vrc][hrc])
        {
            vrc--;
        }
        else
        {
            ans=a[vrc-1]+ans;
            //cout<<vrc<<" ";
            vrc--;
            hrc--;
        }
    }
    // cout<<"longest Common Subsequence : "<<ans<<"\n\n";
    return ans;
}

int main(){
```

```
    // string s1="Classical",s2="Musical";
    //Dynamic
    // cout<<"Enter string 1 : ";
    // cin>>s1;
    // cout<<"Enter string 2 : ";
    // cin>>s2;
    // cout<<endl;
    // lcsubsequence(s1,s2);
    // lcsubsequence(s2,s1);

    //Static test cases
    const char separator = ' ';
    const int nameWidth = 15;
    const int numWidth = 15;
    string s1[3] = {"ABCDAB","EXPONENTIAL","LOGARITHM"};
    string s2[3]={"BDCABA","POLYNOMIAL","ALGORITHM"};
    int n=3;
   cout<<"Answer is:\n\n";

    cout << left << setw(nameWidth) << setfill(separator) << "I";
     cout << left << setw(nameWidth) << setfill(separator) << "N";
     cout << left << setw(nameWidth) << setfill(separator) << "K";
     cout << left << setw(nameWidth) << setfill(separator) << "LCS"<<"\n";

    for(int i=0;i<n;i++){
        cout << left << setw(nameWidth) << setfill(separator) << i;
     cout << left << setw(nameWidth) << setfill(separator) << s1[i];
     cout << left << setw(nameWidth) << setfill(separator) << s2[i];
     cout << left << setw(nameWidth) << setfill(separator)<< lcsubsequence(s1[
i],s2[i])<<"\n";
    }
    return 0;
}
```

OUTPUT:-

```
F:\Sem5\DAA\Practicals\prac5>g++ -std=c++11 5.4.cpp -o 54

F:\Sem5\DAA\Practicals\prac5>54.exe
Answer is:

I               N               K               LCS
0               ABCDAB          BDCABA          BCAB
1               EXPONENTIAL     POLYNOMIAL      PONIAL
2               LOGARITHM       ALGORITHM       LORITHM

F:\Sem5\DAA\Practicals\prac5>
```

Conclusion: -

Thus, I have learned that This problem can be solved using dynamic programming. First we will calculate the length of longest common subsequence for a prefix of str1 and str2. Then, we will tabulate these values in a matrix and use these values to calculate the value of LCS for longer prefixes of str1 and str2.

Time complexity – O(len1*len2)