**Practical-7**

**Aim: Backtracking and Branch & Bound**

**Software Requirement:** GNU Compiler Collection (GCC), MS Word.

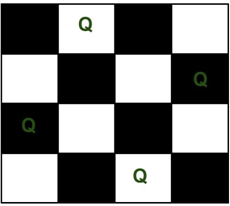
**Hardware Requirement:** Desktop Computer.

**[5.1]**

**Aim:** Write a C program to implement 4 Queen Problem. Is it possible to find the solution for 1 Queen, 2 Queen, 3 Queen?

**Theory:**

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen Problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

{0, Q, 0, 0}

{0, 0, 0, Q}

{Q, 0, 0, 0}

{0, 0, Q, 0}

Another Solution is:

{0, 0, Q, 0}

{Q, 0, 0, 0}

{0, 0, 0, Q}

{0, Q, 0, 0}

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

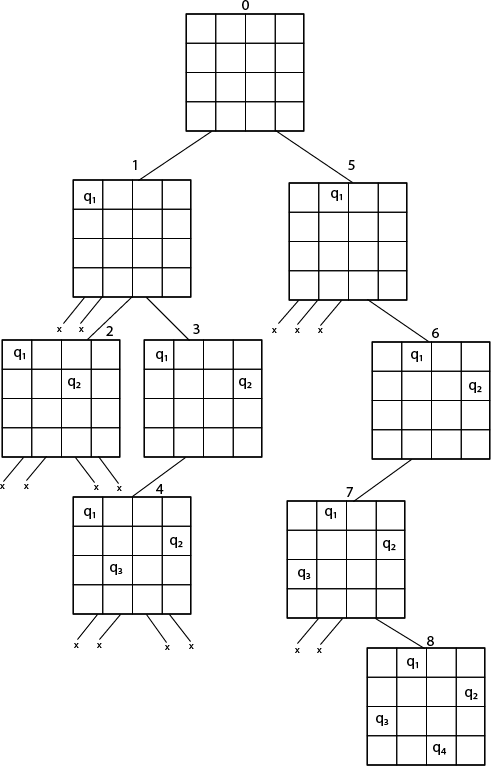
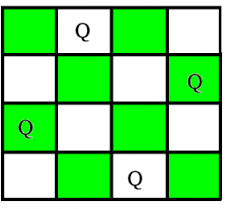


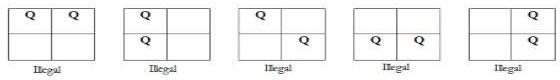
Fig shows the complete state space for 4-queens problem.

**Example:**



N-Queen Problem

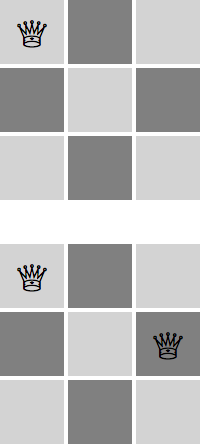
* Yes, It is possible to find **1 queen problem**, because we have only 1 queen and 1 place, so 1 queen can easily store that place.
* **2 – Queen’s problem** is not solvable because 2 – Queens can be placed on 2 x 2 chess board as shown in figure.



* So as shown in figure we can’t put a queen in any way that itfollow the

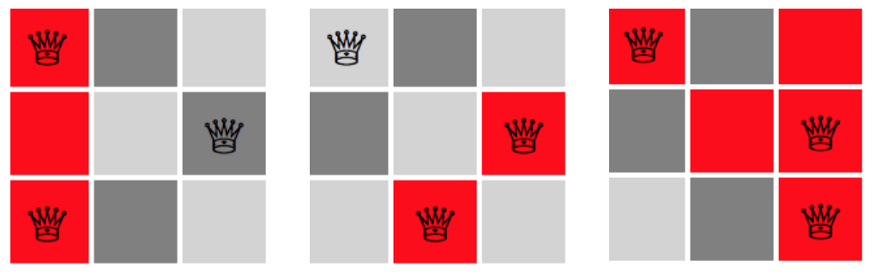
rules of N Queen Problem.

* **3 – Queen’s problem** is not solvable because 3 – Queens can be placed on 3 x 3 chess board
* let’s place the first queen to the top left.
* For the second row, we only have one spot — the far right — to place the queen without the two queens attacking each other.



Exploring permutations after placing the first piece in row 1, column 1

* But then: we are not able to fit a third queen in the third row.



Exploring Possible Placement for Row 3

Placing a Queen in any of the columns in row 3 would create a conflict with other Queens. So we do not have a valid solution for 3 queen problem.

**Algorithm:**

Step 1: Start

Step 2: Given n queens, read n from user and let us denote the queen number by k. k=1,2,..,n.

Step 3: We start a loop for checking if the queen can be placed in the respective column of the row.

Step 4: For checking that whether the queen can be placed or not, we check if the previous queens are not in diagonal or in same row with it.

Step 5: If the queen cannot be placed backtracking is done to the previous queens until a feasible solution is not found.

Step 6: Repeat the steps 3-5 until all the queens are placed.

Step 7: The column numbers of the queens are stored in an array and printed as a n-tuple solution

Step 8: Stop

**Code:**

#include <stdio.h>

int count\_DP=0, count\_R=0;// initializing counters for dynamic programming and recursive methods

//function to find binomial coefficient using dynamic programming

int binCo\_DP(int n, int k)

{

    //declaration of i,j,C

    int C[n + 1][k + 1];

    int i, j;

    count\_DP=count\_DP+2;//Increment counter

    for (i = 0; i <= n; i++) //loop from 0 to n

    {

        count\_DP=count\_DP+2;//Increment counter

        for (j = 0; j <= min(i, k); j++) //loop from 0 to min(i,k)

        {

            count\_DP=count\_DP+2;//Increment counter

            if (j == 0 || j == i) //if j=0 or j=i

            {

                C[i][j] = 1; // assign 1 to c[i][j]

                count\_DP++;//Increment counter

            }

            else//if j not equal to 0 or i

            {

                C[i][j] = C[i - 1][j - 1] + C[i - 1][j]; //assign C[i - 1][j - 1] + C[i - 1][j] to c[i][j]

                count\_DP++;//Increment counter

            }

            count\_DP=count\_DP+2;//Increment counter

        }

        count\_DP=count\_DP+2;//Increment counter

    }

    return C[n][k];//return the final value

}

//function to print the table for dynamic programming

void print\_DP(int n, int k)

{

    int i, j;

    printf("\n\t|\t");

    for(i = 0; i <= k; i++)

    {

        printf("%d\t", i);

    }

    printf("\n------------------------------------");

    for(i = 0; i <= n; i++)

    {

        printf("\n%d\t|\t", i);

        for(j = 0; j <= min(i, k); j++)

        {

            printf("%d\t", binCo\_DP(i, j));// printing the each value by calling binCo\_DP function

        }

        printf("\n");

    }

}

//function to find binomial coefficient using recursion

int binCo\_R(int n, int k)

{

    count\_R++;//Increment counter

    if (k > n)// if k greater than n then return 0

    {

        count\_R++;//Increment counter

        return 0;

    }

    count\_R=count\_R+2;//Increment counter

    if (k == 0 || k == n)// if k=0 or n then return 1

    {

        count\_R++;//Increment counter

        return 1;

    }

    return binCo\_R(n - 1, k - 1)+ binCo\_R(n - 1, k);// recursive call to function

}

//function to print the table for recursion

void print\_R(int n, int k)

{

    int i, j;

    printf("\n\t|\t");

    for(i = 0; i <= k; i++)

    {

        printf("%d\t", i);

    }

    printf("\n------------------------------------");

    for(i = 0; i <= n; i++)

    {

        printf("\n%d\t|\t", i);

        for(j = 0; j <= min(i, k); j++)

        {

            printf("%d\t", binCo\_R(i, j));// printing the each value by calling binCo\_R function

        }

        printf("\n");

    }

}

//function to calculate minimum value

int min(int a, int b)

{

    if(a < b)// if a<b then return a else return b

        return a;

    return b;

}

//driver code

int main()

{

    //Declare n and k

    int n, k;

    //Taking the values of n and k from user

    printf("Enter the value of n: ");

    scanf("%d", &n);

    printf("Enter the value of k: ");

    scanf("%d", &k);

    //Calling & Printing the binomial coefficent and total basic operations using dynamic programming

    printf("\n-----------Using Dynamic Programming-----------\n");

    print\_DP(n, k);

    printf("\nTotal Operations: %d", count\_DP);

    printf("\nValue of C(%d, %d): %d", n, k, binCo\_DP(n, k));

    //Calling & Printing the binomial coefficent and total basic operations using recursion

    printf("\n\n----------------Using Recursion----------------\n");

    print\_R(n, k);

    printf("\nTotal Operations: %d", count\_R);

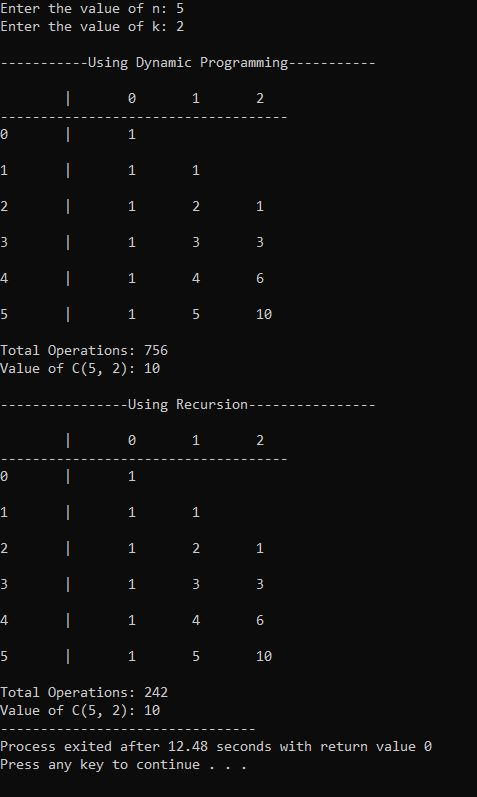
    printf("\nValue of C(%d, %d): %d", n, k, binCo\_R(n, k));

    return 0;

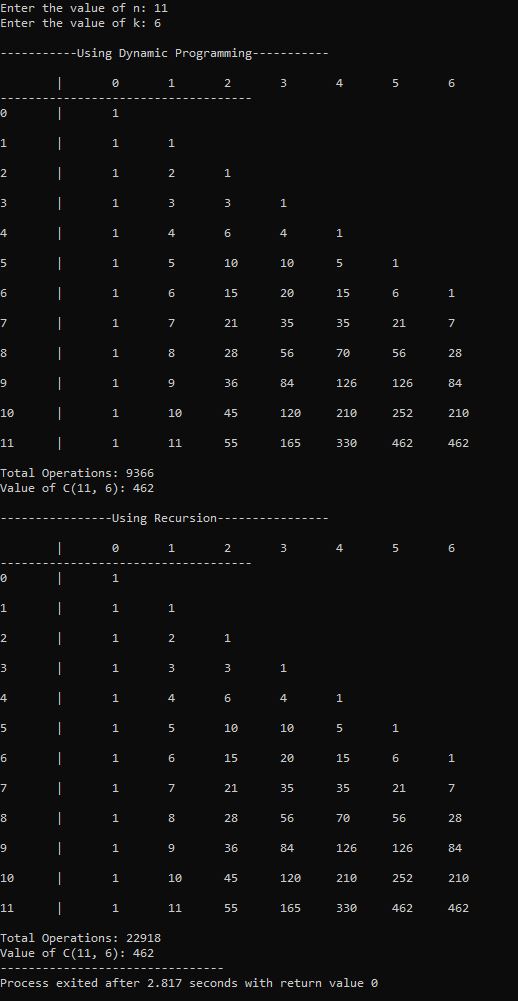
}

**Output:**

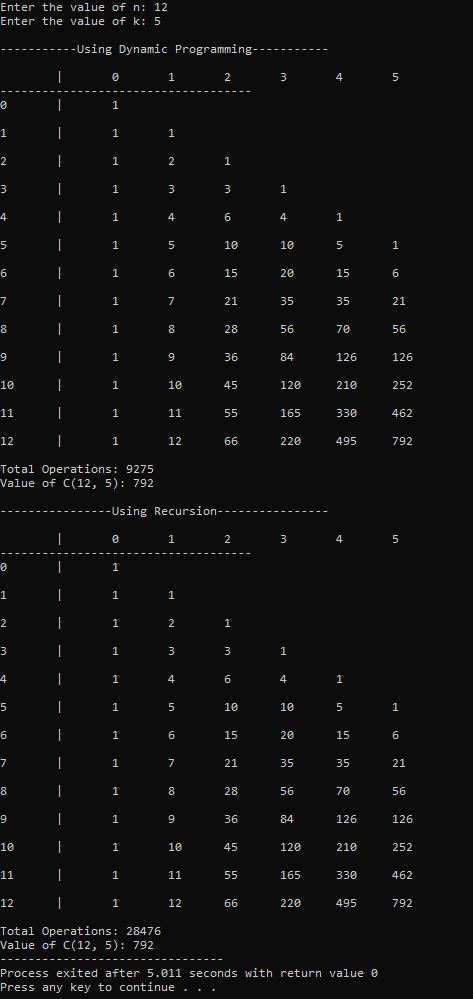
**1)**



**2)**



**3)**

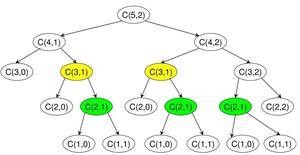
****

**Conclusion:**

In this practical we implemented Binomial Coefficient algorithm with Dynamic Programming Approach. The time complexity of Binomial Coefficient is O(n\*k). The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

C(n, k) = C(n-1, k-1) + C(n-1, k) C(n, 0) = C(n, n) = 1

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for n = 5 an k = 2. The function C(3, 1) is called two times. For large values of n, there will be many common subproblems.



Binomial Coefficients Recursion tree for C(5,2). Since the same subproblems are called again, this problem has Overlapping Subproblems property. So, the Binomial Coefficient problem has both properties (see this and this) of a dynamic programming problem.

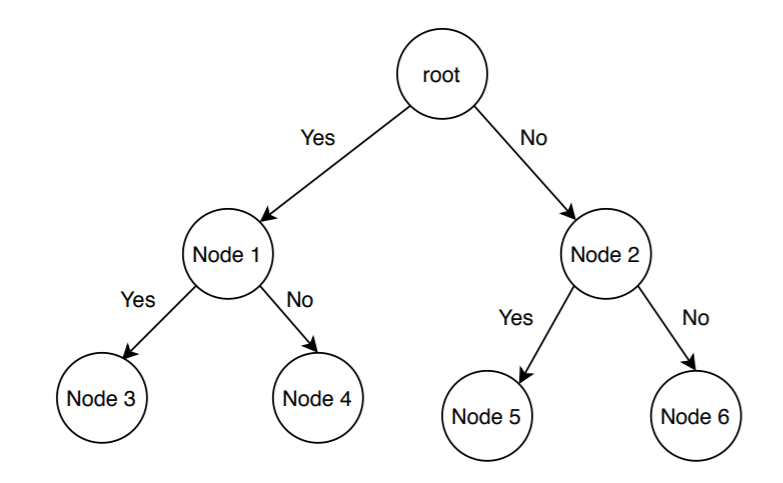
**[5.2]**

**Aim:** Implement and analyse Assembly Line Scheduling.

**Theory:**

**Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.** In general, given an NP-Hard problem, a branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.

A branch and bound algorithm consist of stepwise enumeration of possible candidate solutions by exploring the entire search space. With all the possible solutions, we first build a rooted decision tree. The root node represents the entire search space:



Here, each child node is a partial solution and part of the solution set. Before constructing the rooted decision tree, we set an [upper and lower bound](https://www.baeldung.com/cs/space-complexity) for a given problem based on the optimal solution. At each level, we need to make a decision about which node to include in the solution set. At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.

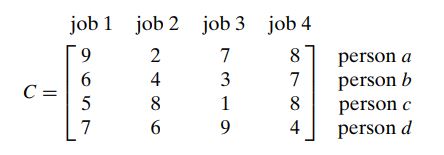
Now it is crucial to find a good upper and lower bound in such cases. We can find an upper bound by using any [local optimization method](https://en.wikipedia.org/wiki/Local_search_(optimization)) or by picking any point in the search space. On the other hand, we can obtain a lower bound from [convex relaxation](https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/2/436/files/2016/11/17-convex-relaxation.pdf) or [duality](https://en.wikipedia.org/wiki/Boolean_algebra#Duality_principle).

In general, we want to partition the solution set into smaller subsets of solution. Then we construct a rooted decision tree, and finally, we choose the best possible subset (node) at each level to find the best possible solution set.

**Example:**

Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. Recall that an instance of the assignment problem is specified by an n × n cost matrix C so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.

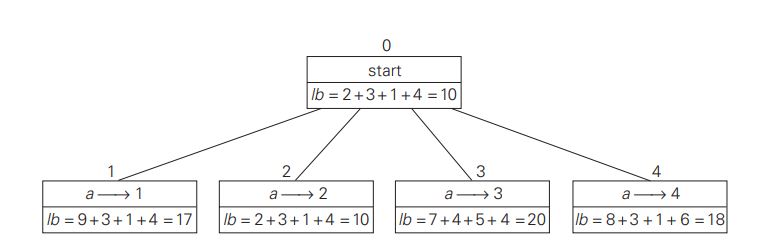
Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.



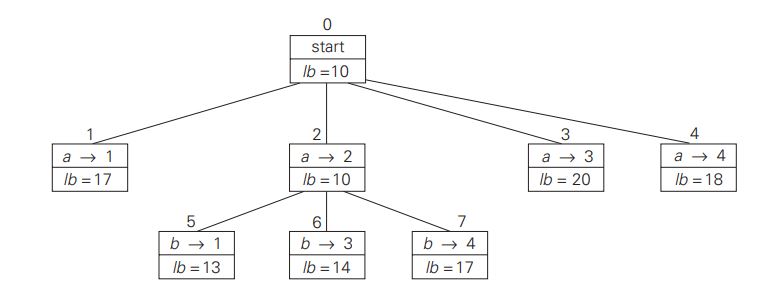
How can we find a lower bound on the cost of an optimal selection without  
actually solving the problem? We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix’s rows. For the instance here, this sum is 2 + 3 + 1 + 4 = 10. It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1 + 4 = 17.

One more comment is in order before we embark on constructing the problem’s state-space tree. It deals with the order in which the tree nodes will be generated. Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among non terminated leaves in the current tree. (Nonterminated, i.e., still promising, leaves are also called live.) How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the best-first branch-and-bound.

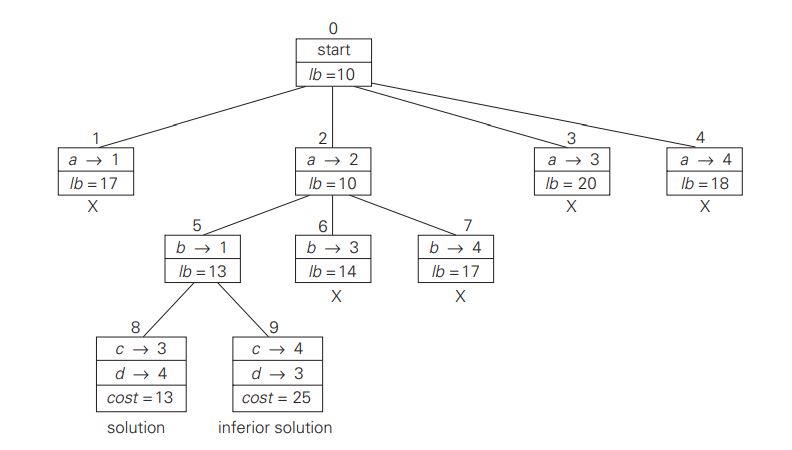
So, returning to the instance of the assignment problem given earlier, we start with the root that corresponds to no elements selected from the cost matrix. As we already discussed, the lower-bound value for the root, denoted lb, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a (Figure a).

Figure a: Levels 0 and 1 of the state-space tree for the instance of the assignment  
problem being solved with the best-first branch-and-bound algorithm. The  
number above a node shows the order in which the node was generated.  
A node’s fields indicate the job number assigned to person a and the  
lower bound value, lb, for this node.

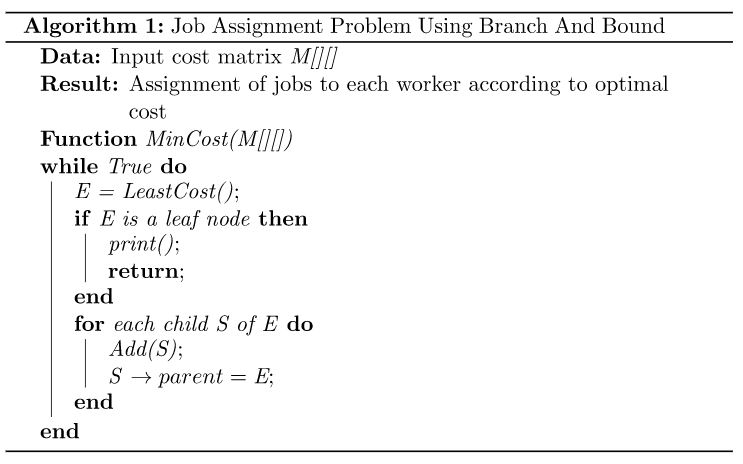
So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value. Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column the three different jobs that can be assigned to person b (Figure b).

Figure b: Levels 0, 1, and 2 of the state-space tree for the instance of the assignment  
problem being solved with the best-first branch-and-bound algorithm.

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column’s element from c’s row (i.e., assigning person c to job 3); this leaves us with no choice but to select the element from the fourth column of d’s row (assigning person d to job 4). This yields leaf 8 (Figure c), which corresponds to the feasible solution {a → 2, b → 1, c → 3, d → 4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {a → 2, b → 1, c → 4, d → 3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.) Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure c—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

Figure c: Complete state-space tree for the instance of the assignment problem  
solved with the best-first branch-and-bound algorithm.

**Algorithm:**

****

**Code:**

#include <stdio.h>

//Function to find minimum time required and to find the final path

int AssemblyLineScheduling(int lines, int stations, int a[lines][stations], int t[lines][stations], int e[lines], int x[lines])

{

    //Declare variables

    int T1[stations], T2[stations], i, L1[stations-1], L2[stations-1];

    printf("\n\nj\t|");

    for(i = 1; i <= stations; i++)

        printf("\t%d ",i);

    printf("\n-------------------------------------------------------------");

    printf("\nF1[j]\t|");//print the time taken by each station on line 1

    T1[0] = e[0] + a[0][0]; //Add start time of line 1 with 1st station of line 1

    printf("\t%d",T1[0]);// print T1[0] value

    T2[0] = e[1] + a[1][0];//Add start time of line 2 with 1st station of line 2

    for (i = 1; i < stations; ++i)//Loop to calculate the minimum time by adding transaction times with previous station values

    {

        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i]);//find minimum time and assign to T1[i]

        if(T1[i-1] + a[0][i] < T2[i-1] + t[1][i] + a[0][i])//if line 1 takes less time than line 2 then assign 1 to L1[i] else assign 2

            L1[i] = 1;

        else

            L1[i] = 2;

        printf("\t%d",T1[i]);//print print T1[i] value

        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i]);//find minimum time and assign to T2[i]

        if(T2[i-1] + a[1][i] > T1[i-1] + t[0][i] + a[1][i])//if line 1 takes less time than line 2 then assign 1 to L2[i] else assign 2

            L2[i] = 1;

        else

            L2[i] = 2;

    }

    printf("\nF2[j]\t|");

    printf("\t%d",T2[0]);

    for (i = 1; i < stations; ++i)

    {

        printf("\t%d",T2[i]);//print the time taken by each station on line 2

    }

     printf("\n\n\nj\t|");

    for(i = 2; i <= stations; i++)

        printf("\t%d ",i);

    printf("\n-------------------------------------------------------------");

    printf("\nL1[j]\t|");

    for (i = 1; i < stations; ++i)

    {

        printf("\t%d",L1[i]);// Print the line number which takes minimum time at that particular station

    }

    printf("\nL2[j]\t|");

    for (i = 1; i < stations; ++i)

    {

        printf("\t%d",L2[i]);// Print the line number which takes minimum time at that particular station

    }

    printf("\n\n\nFinal Path: \n");// print the final path

    for (i = stations-1; i >= 0; i--)

    {

        if(T1[i] < T2[i])

            printf("Station %d Line 1\n",i+1);// print the station number with line number

        else

            printf("Station %d Line 2\n",i+1);// print the station number with line number

    }

    return min(T1[stations-1] + x[0], T2[stations-1] + x[1]);//return the minimum time required

}

//function to find minimum value

int min(int a, int b)

{

    if(a < b) // if a<b then return a else return b

        return a;

    return b;

}

//driver code

int main()

{

    //Declare the variables to take input

    int i,j,lines;

    int stations;

    int e[lines],x[lines];

    //Taking the no. of lines,stations from user

    printf("Enter the number of lines: ");

    scanf("%d",&lines);

    printf("Enter the number of stations: ");

    scanf("%d",&stations);

    //Declare station time and transaction time variables

    int a[lines][stations];

    int t[lines][stations];

    //Take input of Start Time for Number of Lines from user

    printf("Enter the start times for the lines:\n");

    for (i = 0; i < lines; i++)

    {

        printf("E[%d]: ",i+1);

        scanf("%d",&e[i]);

    }

    //Take input of end Time for Number of Lines from user

    printf("Enter the end times for the lines:\n");

    for (i = 0; i < lines; i++)

    {

        printf("X[%d]: ",i+1);

        scanf("%d",&x[i]);

    }

    // Take Input for Start Time for S[i,j] from user

    printf("Enter the time taken at each station:\n");

    for(i=0;i<lines;i++)

    {

        for(j=0;j<stations;j++)

        {

            printf("S[%d][%d]: ",i+1,j+1);

            scanf("%d",&a[i][j]);

        }

    }

    //Take Input for Transaction Time for T[i,j] from user

    printf("Enter the transaction times:\n");

    for(i=0;i<lines;i++)

    {

        for(j=0;j<stations;j++)

        {

            printf("T[%d][%d]: ",i+1,j+1);

            scanf("%d",&t[i][j]);

        }

    }

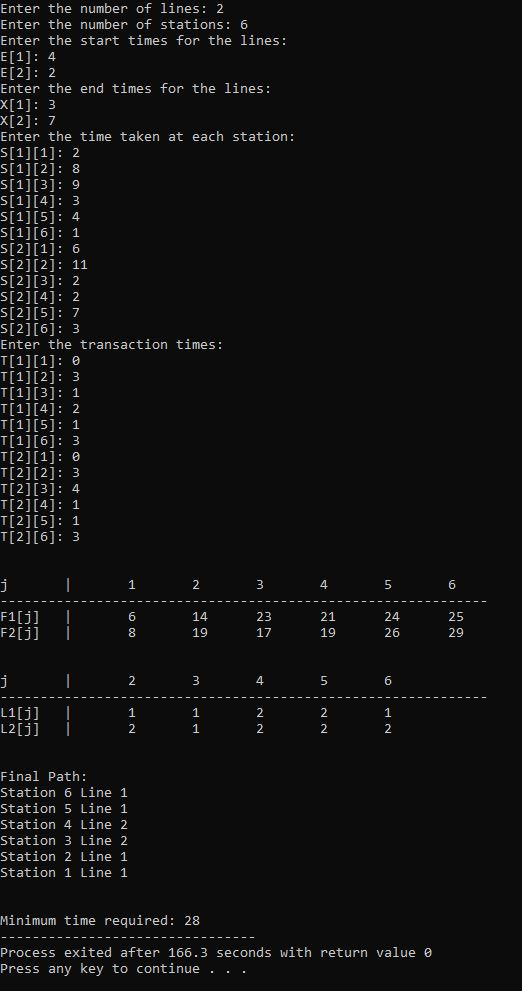
    //Passing the values to function and printing the minimum time required

    printf("\n\nMinimum time required: %d", AssemblyLineScheduling(lines, stations, a, t, e, x));

    return 0;

}

**Output:**

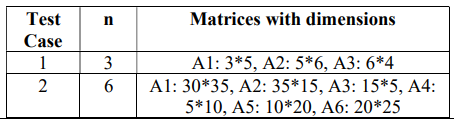
****

**Conclusion:**

In this practical we performed Assembly line scheduling algorithm using Dynamic Programming and calculated it’s time complexity. The time complexity turned out to be O(n).

**[5.3]**

**Aim:** Given a chain < A1, A2,…,An> of n matrices, where for i=1,2,…,n matrix Ai 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.



**Theory with Example:**

Let Ai,j be the result of multiplying matrices i through j. It can be seen that the dimension of Ai,j is pi-1 x pj 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

A1.....n=A1....k x Ak+1....n)

Thus we are left with two questions:

* How to split the sequence of matrices?
* How to parenthesize the subsequence A1.....k andAk+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 (n2 ) 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 Ai....j where i≤ j denotes the matrix that results from evaluating the product

Ai Ai+1....Aj.

If i < j then any parenthesization of the product Ai Ai+1 ......Aj must split that the product between Ak and Ak+1 for some integer k in the range i ≤ k ≤ j. That is for some value of k, we first compute the matrices Ai.....k & Ak+1....j and then multiply them together to produce the final product Ai....j. The cost of computing Ai....k plus the cost of computing Ak+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 matrixAi....j.

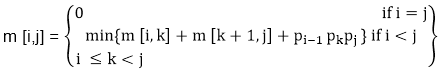
If i=j the chain consist of just one matrix Ai....i=Ai so no scalar multiplication are necessary to compute the product. Thus m [i, j] = 0 for i= 1, 2, 3....n.

If i<j we assume that to optimally parenthesize the product we split it between Ak and Ak+1 where i≤ k ≤j. Then m [i,j] equals the minimum cost for computing the subproducts Ai....k and Ak+1....j+ cost of multiplying them together. We know Ai has dimension pi-1 x pi, so computing the product Ai....k and Ak+1....jtakes pi-1 pk pj scalar multiplication, we obtain

m [i,j] = m [i, k] + m [k + 1, j] + pi-1 pk pj

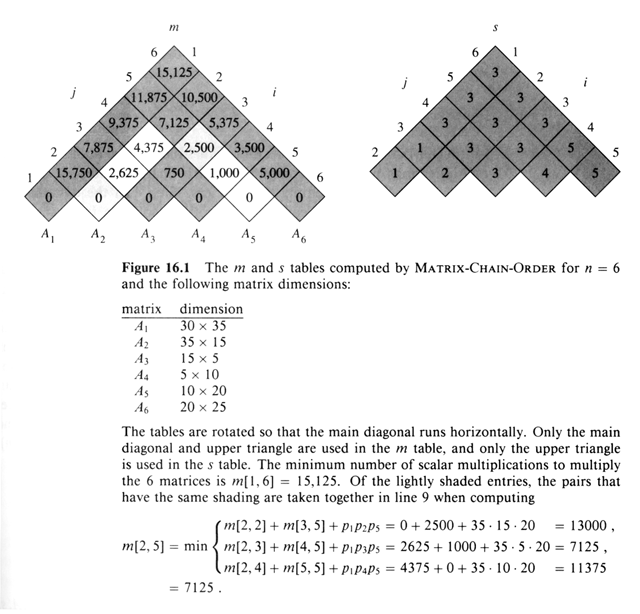
There are only (j-1) possible values for 'k' namely k = i, i+1.....j-1. Since the optimal parenthesization must use one of these values for 'k' we need only check them all to find the best.

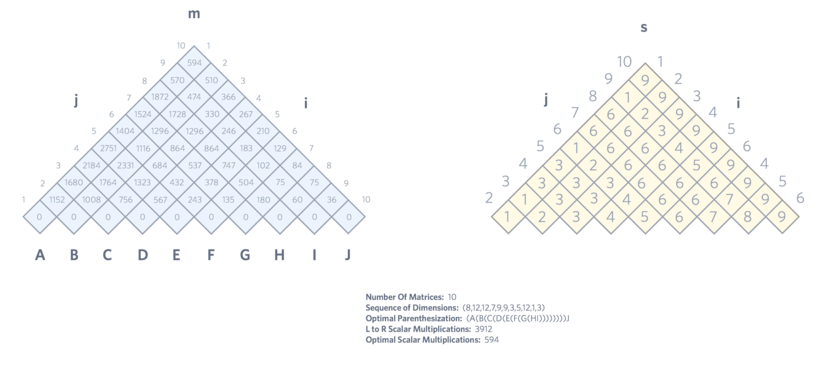
So the minimum cost of parenthesizing the product Ai Ai+1......Aj becomes



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

m [i,j] = m [i, k] + m [k + 1, j] + pi-1 pk pj





**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] + pi-1 pk pj

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

//declaration of variables globally

long int m[20][20];

int s[20][20];

int p[20],a[20],b[20],i,j,n;

//function to print optimal answer

void print\_optimal(int i,int j)

{

    if (i == j)// if i and j same then print matrix number

    {

        printf(" A%d ",i);

    }

    else// else find the optimal of below values recursively

    {

        printf("( ");

        print\_optimal(i, s[i][j]);

        print\_optimal(s[i][j] + 1, j);

        printf(" )");

    }

}

//function to find the matrix values

void matmultiply(void)

{

    //Declaration of q and k values

    long int q;

    int k;

    for(i=n;i>0;i--)

    {

    for(j=i;j<=n;j++)

    {

     if(i==j)// if i and j same then matrix value same.Ex. m[1][1]=0

       m[i][j]=0;

     else// else do find the value according to the formula with value of i as i<=k<j

       {

        for(k=i;k<j;k++)

        {

         q=m[i][k]+m[k+1][j]+p[i-1]\*p[k]\*p[j];// calculating the value using formula

         if(q<m[i][j])// checking if it is the minimum or not.If yes then replace and if not then do not replace

          {

            m[i][j]=q;

            s[i][j]=k;

          }

         }

        }

      }

    }

}

//function to find the minimum multiplications required

int MatrixChainOrder(int p[], int i, int j)

{

    if(i == j)// if i and j same then return 0

        return 0;

    //Declaration of variables and assigning INT\_MAX value to min variable

    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];//finding the minimum multiplications required for matrix chain multiplication or m[1][n]

        if (count < min)// checking if it is the minimum or not.If yes then replace and if not then do not replace

            min = count;

    }

    // Return minimum count

    return min;

}

//driver code

void main()

{

    //Declaration of k

    int k,h;

    //Taking the no. of matrices from user

    printf("Enter the no. of matrices: ");

    scanf("%d",&n);

    // Initialize with infinite values

    for(i=1;i<=n;i++)

        for(j=i+1;j<=n;j++)

        {

            m[i][i]=0;

            m[i][j]=INFY;

            s[i][j]=0;

        }

        //Taking the dimensions of each matrix from user

        printf("Enter the dimensions ( A[m x n] ): \n");

        for(k=0;k<n;k++)

        {

            printf("Matrix A%d[Dimension m]: ",k+1);

            scanf("%d",&a[k]);

            printf("Matrix A%d[Dimension n]: ",k+1);

            scanf("%d",&b[k]);

            p[k]=a[k];// assigning m dimension of each matrix to p array's kth position

        }

        p[n]=b[n-1];// assigning n dimension of last matrix to p array's nth position

        printf("\n");

        for(k=0;k<=n;k++)

        {

            printf("P%d:%d\t",k,p[k]);// printing the values of p array

        }

        printf("\n");

        matmultiply();// calling the matmultipy function

        // Printing the cost of matrix

        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;//assigning 1 to i and n to j

    //printing the multiplication sequence

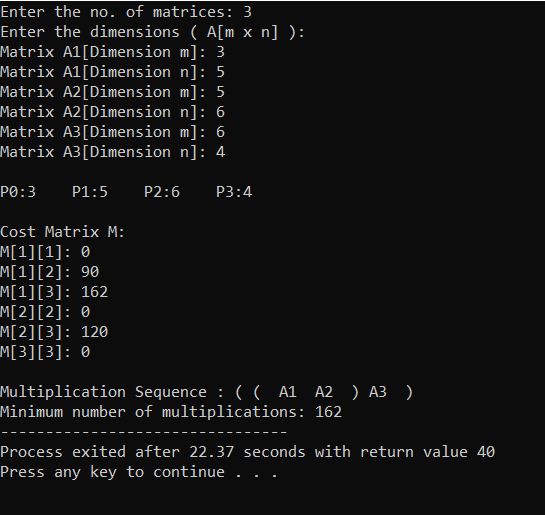
    printf("\nMultiplication Sequence : ");

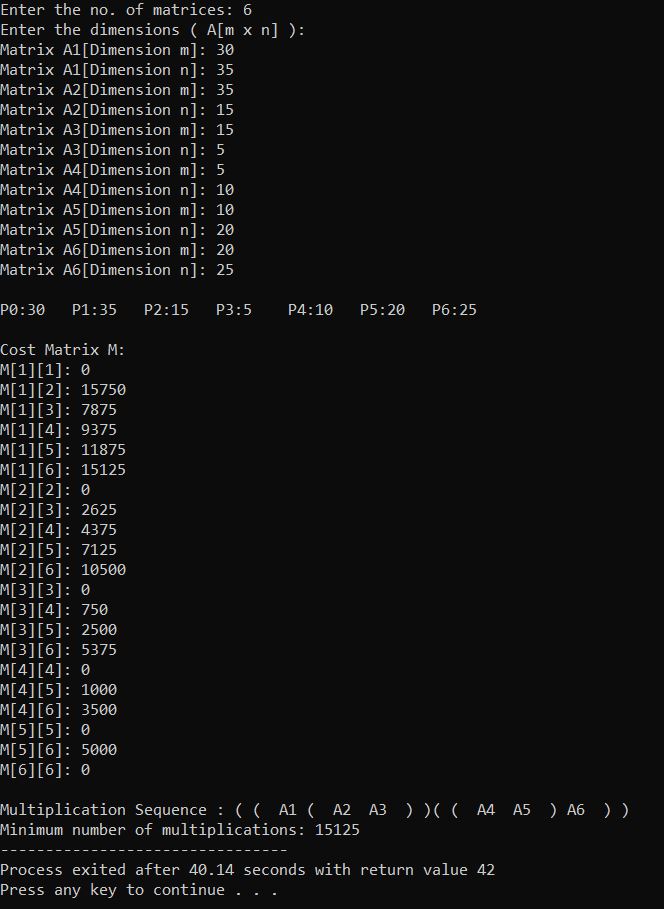
    print\_optimal(i,j);//calling the print\_optimal function to print the optimal or minimum answer

    printf("\nMinimum number of multiplications: %d ", MatrixChainOrder(p, 1, n));//calling and printing the matrixChainOrder function to find the minimum no. of multiplications required

}

**Output:**

****

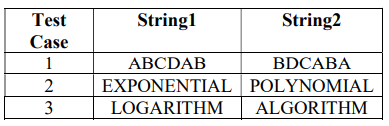
****

**Conclusion:**

In this practical we performed Matrix Chain Multiplication using Dynamic Programming Approach and calculated it’s time complexity. The time complexity turned out to be O(n3).

**[5.4]**

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



**Theory:**

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then

L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

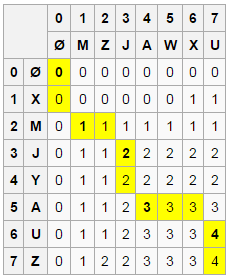
If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then

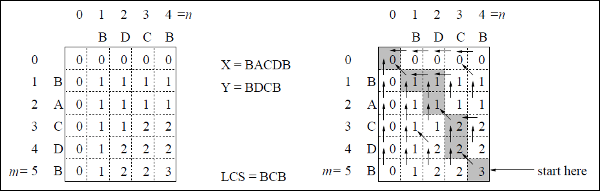
L(X[0..m-1], Y[0..n-1]) = MAX ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]) )

**Example:**

Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)





**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(xi)

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<stdio.h>

#include<string.h>

//function to find max value

int max(int x, int y)

{

    if(x > y)// if x greater than y then return x else return y

        return x;

    return y;

}

// function to find lcs

void lonComSub(char \*X, char \*Y, int m, int n)

{

   //Declare the variables

   int i, j, L[m+1][n+1];

   for (i=0; i<=m; i++)// loop from 0 to m and inner loop from 0 to n

   {

     for (j=0; j<=n; j++)

     {

       if (i == 0 || j == 0)// if i=0 or j=0 then L[i][j]=0 else if X[i-1]=Y[i-1] the L[i][j] = L[i-1][j-1] + 1 else L[i][j] = max(L[i-1][j], L[i][j-1]);

         L[i][j] = 0;

       else if (X[i-1] == Y[j-1])

         L[i][j] = L[i-1][j-1] + 1;

       else

         L[i][j] = max(L[i-1][j], L[i][j-1]);

     }

   }

  //declaring the index,lcs variables

   int index = L[m][n];

   char lcs[index+1];

   lcs[index] = '\0';

   i = m, j = n;//assigning m and n to i and j respectively

   while (i > 0 && j > 0)//while loop with condition i>0 and j>0

   {

      if (X[i-1] == Y[j-1])//if X[i-1] equals Y[i-1]

      {

          lcs[index-1] = X[i-1];//assign X[i-1] to lcs[index-1] and decrement i, j and index

          i--;

          j--;

          index--;

      }

      else if (L[i-1][j] > L[i][j-1])// if L[i-1][j] greater than L[i][j-1] then decrement i

         i--;

      else

         j--;// else decrement j in other conditions

   }

   // printing the whole table to find the length and string

   printf("\n  | ");

   for(i = 0; i <= n; i++)

   {

        if(i == 0)

        {

            printf("Y");

        }

        else

        {

            printf(" %c",X[i-1]);// printing string 1

        }

   }

   printf("\n-----------------------------------------");

   for(i = 0; i <= m; i++)

   {

        if(i == 0)

        {

            printf("\nX | ");

        }

        else

        {

            printf("%c | ",Y[i-1]);// printing string 2

        }

    for(j = 0; j <= n; j++)

    {

        printf("%d ",L[i][j]);// printing all the values

    }

    printf("\n");

    }

    //printing the lcs length and lcs string

   printf("\nLength of LCS string: %d", strlen(lcs));

   printf("\nLCS of %s and %s: %s", X, Y, lcs);

}

//driver code

int main()

{

    //declare the variables m and n

    int m, n;

    //taking the length input of both strings from user

    printf("Enter the length of first string: ");

    scanf("%d",&m);

    printf("Enter the length of second string: ");

    scanf("%d",&n);

    //declare variable X and Y for strings

    char X[m], Y[n];

    ////taking the length of both strings from user

    printf("Enter the first string: ");

    scanf("%s",&X);

    printf("Enter the second string: ");

    scanf("%s",&Y);

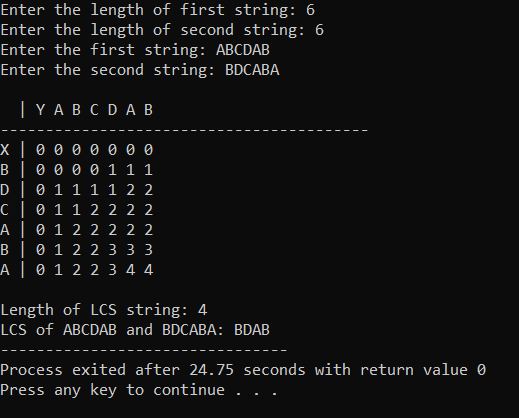
    //calling the function

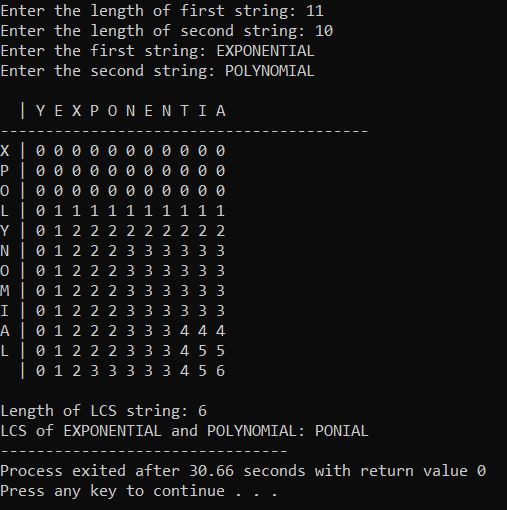
    lonComSub(X, Y, m, n);

    return 0;

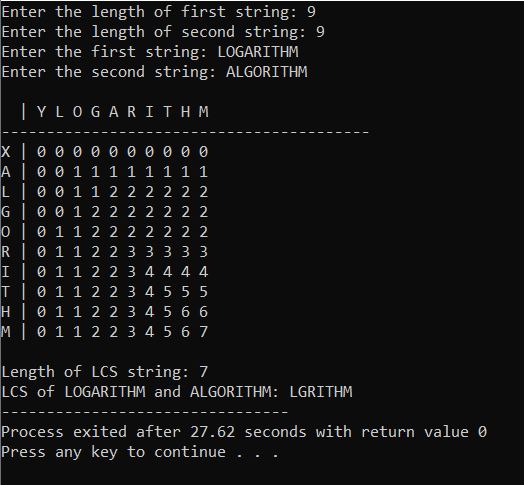
}

**Output:**

****

****

**3)**

****

**Conclusion:**

In this practical we performed Least common subsequence by Dynamic Programming Approach. And found out that time complexity of LCS with Dynamic Programming is O(m\*n). Where m and n are length of two strings. This time complexity is much better than time complexity of same problem if performed using recursion method.