LAB – 7

AIM OF THE EXPERIMENT:

To implement Matrix –chain multiplication algorithm using dynamic programming.

ALGORITHM:

Algorithm Matrix-Chain-Order(p)

// p is the array which has dimensions of matrices (p[0] × p[1] ….)

{

n = p. length – 1

let m [1..n, 1..n] and s [1..n-1, 2..n] be new tables

for i =1 to n

m[i, i] = 0

for l =2 to n // l is chain length

for i=1 to n-l+1

j=i+l-1

m[i, j] = ∞

for k =i to j-1

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

return m and s

}

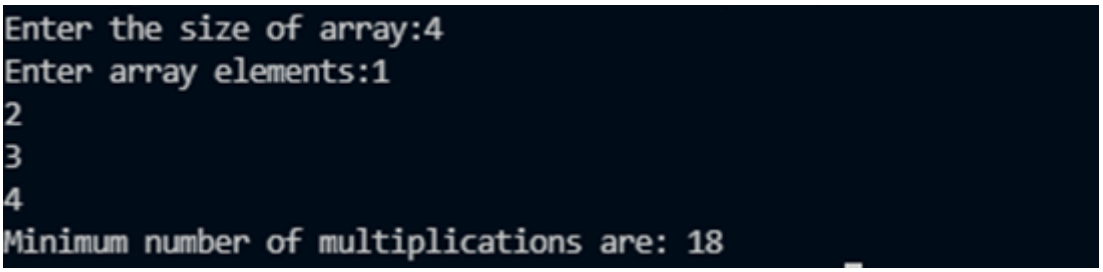
CODE:

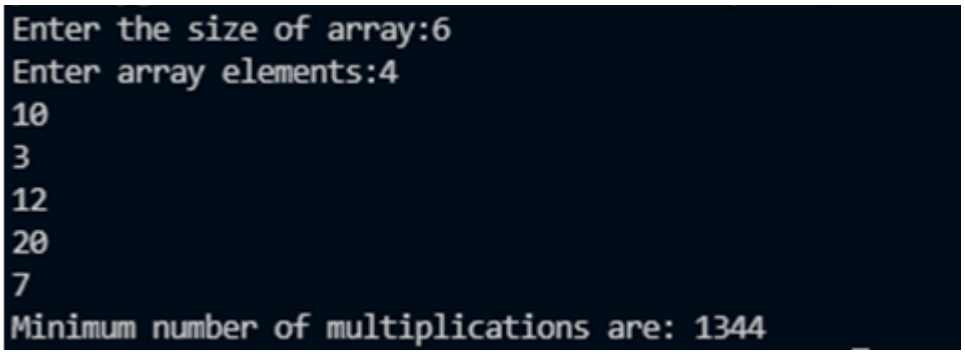
```
#include <stdio.h>
```

```
#include <limits.h>
int MatrixChainOrder(int p[], int n)
{
    int i, j, k, L, q;
    int m[n][n];
    // cost is zero when multiplying one matrix.
    for (i = 1; i< n; i++)
        m[i][i] = 0;
    // L is chain length.
    for (L = 2; L < n; L++)
    {
        for (i = 1; i< n - L + 1; i++)
        {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; 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;
            }
        }
    }
    return m[1][n - 1];
}
int main()
{
    int n;
printf("Enter the size of array:");
scanf("%d",&n);
    int a[n];
printf("Enter array elements:");
```

```
for(int i=0;i<n;i++)
   {
         scanf("%d ",&a[i]);
   }
printf("Minimum number of multiplications are: %d ",MatrixChainOrder(a,n));
   return 0;
}
```

OUTPUTS:





PERFORMANCE ANALYSIS:

Time Complexity is $O(n^3)$.

Space Complexity is $O(n^2)$ as we are using one extra 2-dimensional array of size n×n.

RESULT:

Matrix –chain multiplication algorithm using dynamic programming has been executed successfully.

PRELAB QUESTIONS:

1. Differentiate dynamic programming from greedy approach.

A. In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution whereas in Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.

2. Write a brute force procedure for parenthesizing a chain of matrices and compare it with dynamic programming.

A. Denote the number of alternative parenthesizations of a sequence of n matrices by P(n). When n =1, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When n >= 2, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and (k+1)st matrices for any k = 1,2,….n-1. Thus, we obtain the recurrence as:

$$P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)\,P(n-k) & \text{if } n>=2 \end{cases}$$

A simpler exercise is to show that the solution to the recurrence is $\Omega(2^n)$. The number of solutions is thus exponential and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain. Using dynamic programming complexity reduces to $O(n^3)$.

3. Consider the coin change problem. Assume that one has denominations of 1,5,10,25 and 50.If so how can one give change for 87 and 93?It can be recollected that the objective of the coin change problem is to use the smallest number of coin? Design dynamic programming algorithm for this problem

A.
```
void coinchange(int coin[], int n, int value)
{
   int i, j;
   int min_coin[MAX];
   int min;
   int count[MAX];
min_coin[0] = 0;

   for (i=1; i<= value; i++)
   {
        min = 999;
        for (j = 0; j<n; j++)
        {
             if (coin[j] <= i)
             {
                  if (min >min_coin[i-coin[j]]+1)
                  {
                       min = min_coin[i-coin[j]]+1;
                       count[coin[j]]++;
```

```
                    }
                }
            }
        min_coin[i] = min;
    }

    printf("Minimum coins required are: %d \n", min_coin[value]);
    }
```

4. What is the purpose of Kadane algorithm?

A. The purpose of Kadane algorithm is to find the sum of contiguous subarray within a one-dimensional array of numbers that has the largest sum.The simple idea of Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive-sum compare it with max_so_far and update max_so_far if it is greater than max_so_far.

5. Four matrices M1, M2, M3 and M4 of dimensions pxq, qxr, rxs and sxt respectively can be multiplied is several ways with different number of total scalar multiplications. For example, when multiplied as ((M1 X M2) X (M3 X M4)), the total number of multiplications is pqr + rst + prt. When multiplied as (((M1 X M2) X M3) X M4), the total number of scalar multiplications is pqr + prs + pst. If p = 10, q = 100, r = 20, s = 5 and t = 80, then how many number of scalar multiplications needed.

A. If the order of multiplication is ((M1 X M2) X (M3 X M4)) then total number of scalar multiplications are (10*100*20) + (20*5*80) + (10*20*80) = 44000. If the order of multiplication is (((M1 X M2) X M3) X M4) then total number of scalar multiplications are (10*100*20) + (10*20*5) + (10*5*80) = 25000.

LAB – 8

AIM OF THE EXPERIMENT:

To implement all pairs shortest path algorithm using dynamic programming.

ALGORITHM:

```
Algorithm Floyd (L[l.. n, 1..n])
// array [1.. n,1. n]  array D[L..n,1 ..n]
{
        D := L
        for k := 1 to n do
                for i := 1 to n do
                        for j :=1 to n do
                                D[i,j]:=min(D[i, j],D[i, k]+D[k,j])
        return D
}
```

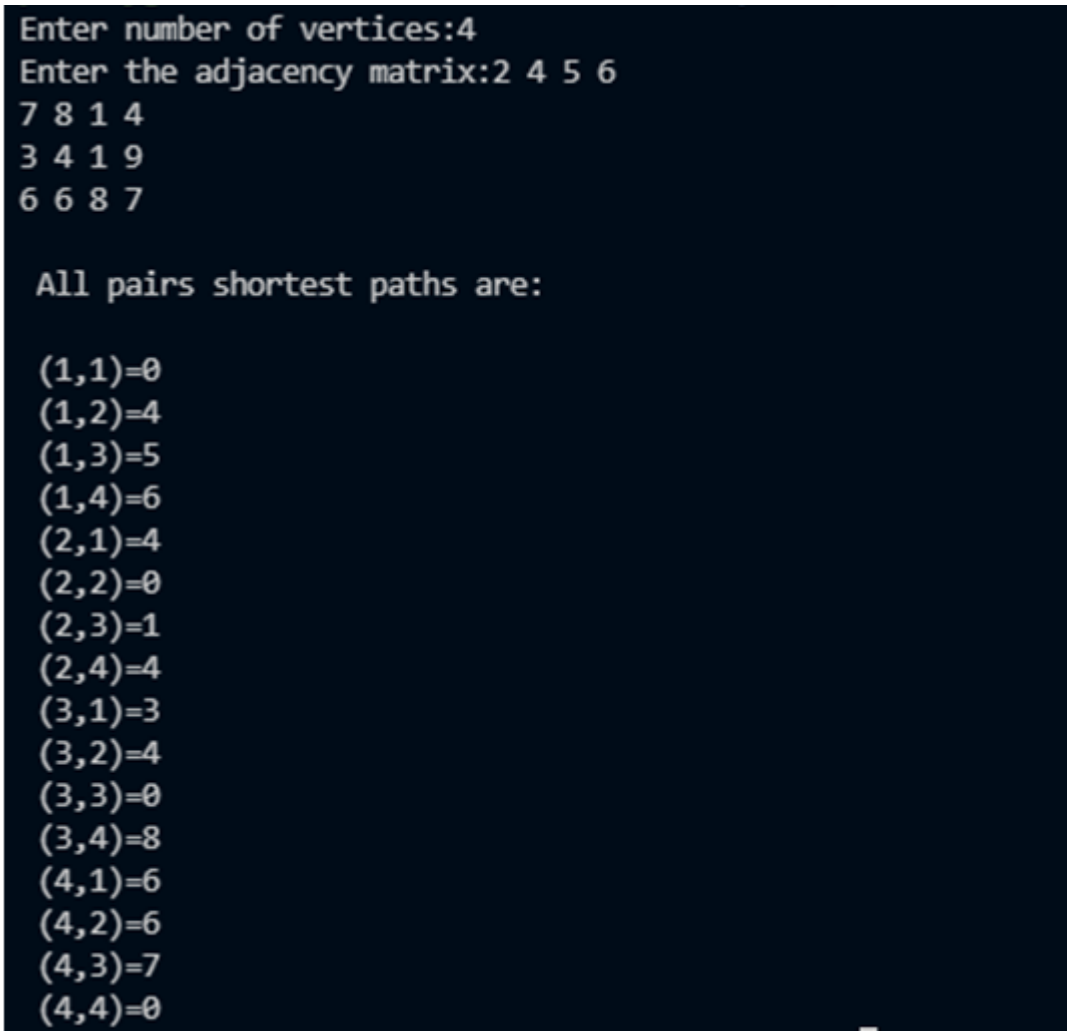
CODE:

```
#include<stdio.h>
#define INFINITY 99999999
int min (int a,int b);
void floydwarshall(int p[150][150],int n)
{
        int i,j,k;
        for (k=1;k<=n;k++)
        {
          for (i=1;i<=n;i++)
          {
           for (j=1;j<=n;j++)
           {
```

```
                    if(i==j)
                        p[i][j]=0;
                    else
                        p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
                }
            }
        }
}
int min(int a,int b)
{
    if(a<b)
        return a;
    else
        return b;
}
int main()
{
    int G[150][150];
    int n,i,j;
    printf("Enter number of vertices:");
    scanf("%d",&n);
    printf("Enter the adjacency matrix:");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&G[i][j]);
        }
    }
    floydwarshall(G,n);
```

```
        printf("\n All pairs shortest paths are:\n");
        for (i=1;i<=n;i++)
        {
          for (j=1;j<=n;j++)
          {
                printf("\n (%d,%d)=%d",i,j,G[i][j]);
          }
        }
        return 0;
}
```

OUTPUTS:



PERFORMANCE ANALYSIS:

We have to apply this algorithm ntimes, each time choosing a different as the source. So the total computation time is $n\times 0(n^2) = 0(n^3)$. This time complexity is same for best, average and worst case. Space Complexity is $O(n^2)$ as we are using extra 2 dimensional array of size $n\times n$.

RESULT: All pairs shortest path algorithm using dynamic programming has been implemented successfully.

PRELAB QUESTIONS:

1.What is the purpose of Floyd's algorithm?

A. The Floyd Warshall Algorithm is a Graph Analysis Algorithm for finding shortest paths in weighted, directed graph. It aims to compute the shortest path from each vertex to nodes. It uses a Dynamic Programming methodology to solve the all pair shortest path Problem.

2. What is the time complexity of Floyd's algorithm?

A. We have to apply this algorithm ntimes, each time choosing a different as the source. So the total computation time is $n\times 0(n^2) = 0(n^3)$. This time complexity is same for best, average and worst case.

3. Distinguish Bellman Ford and Floyd warshall's algorithm

A. Bellman Ford algorithm is used to find out shortest paths in graphs only from one fixed vertex(source) to all other vertices though negative edges are present but there should not be negative cycles. Floyd Warshall's algorithm is used to find shortest paths in graphs from every vertex to remaining other vertices. Both are implemented using dynamic programming.

4. Define optimal substructure in dynamic programming.

A. Optimal Substructure: A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its sub problems.

5. Explain how John's Algorithm on All pairs shortest path is different from Floyd's Warshall's Algorithm.

A. In Floyd warshall algorithm we initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. If we apply Dijkstra's Single Source shortest path algorithm for every vertex, considering every vertex as source, we can find all pair shortest paths in O(V*VLogV) time. So using Dijkstra's single source shortest path seems to be a better option than Floyd Warshell, but the problem with Dijkstra's algorithm is, it doesn't work for negative weight edge. The idea of Johnson's algorithm is to re-weight all edges and make them all positive, then apply Dijkstra's algorithm for every vertex. The idea of Johnson's

algorithm is to assign a weight to every vertex. Let the weight assigned to vertex u be h[u]. We reweight edges using vertex weights. For example, for an edge (u, v) of weight w(u, v), the new weight becomes w(u, v) + h[u] – h[v]. The great thing about this reweighting is, all set of paths between any two vertices are increased by same amount and all negative weights become non-negative. Consider any path between two vertices s and t, weight of every path is increased by h[s] – h[t], all h[] values of vertices on path from s to t cancel each other. The main steps in algorithm are Bellman Ford Algorithm called once and Dijkstra called V times. Time complexity of Bellman Ford is O(VE) and time complexity of Dijkstra is O(VLogV). So overall time complexity is $O(V^2 \log V + VE)$.

The time complexity of Johnson's algorithm becomes same as Floyd Warshell when the graphs is complete (For a complete graph $E = O(V^2)$). But for sparse graphs, the algorithm performs much better than Floyd Warshell.

LAB – 9

AIM OF THE EXPERIMENT:

To implement 0/1 knapsack algorithm using dynamic programming.

ALGORITHM:

Algorithm Knapsack(w[ ],V[ ],W ,n)

{

for w = 0 to W

V[0,w] = 0

for i = 1 to n

V[i,0] = 0

for i = 1 to n

for w = 0 to W

if w_i<= w // item i can be part of the solution

if b_i + V[i-1,w-w_i] > V[i-1,w]

V[i,w] = b_i + V[i-1,w- w_i]

else

V[i,w] = V[i-1,w]

else V[i,w] = V[i-1,w] // w_i> w

i=n, k=W

while i,k> 0

if $V[i,k] \neq V[i-1,k]$ then

mark the i^{th} item as in the knapsack

i=i-1;

k=k-w_i;

else

i=i-1;

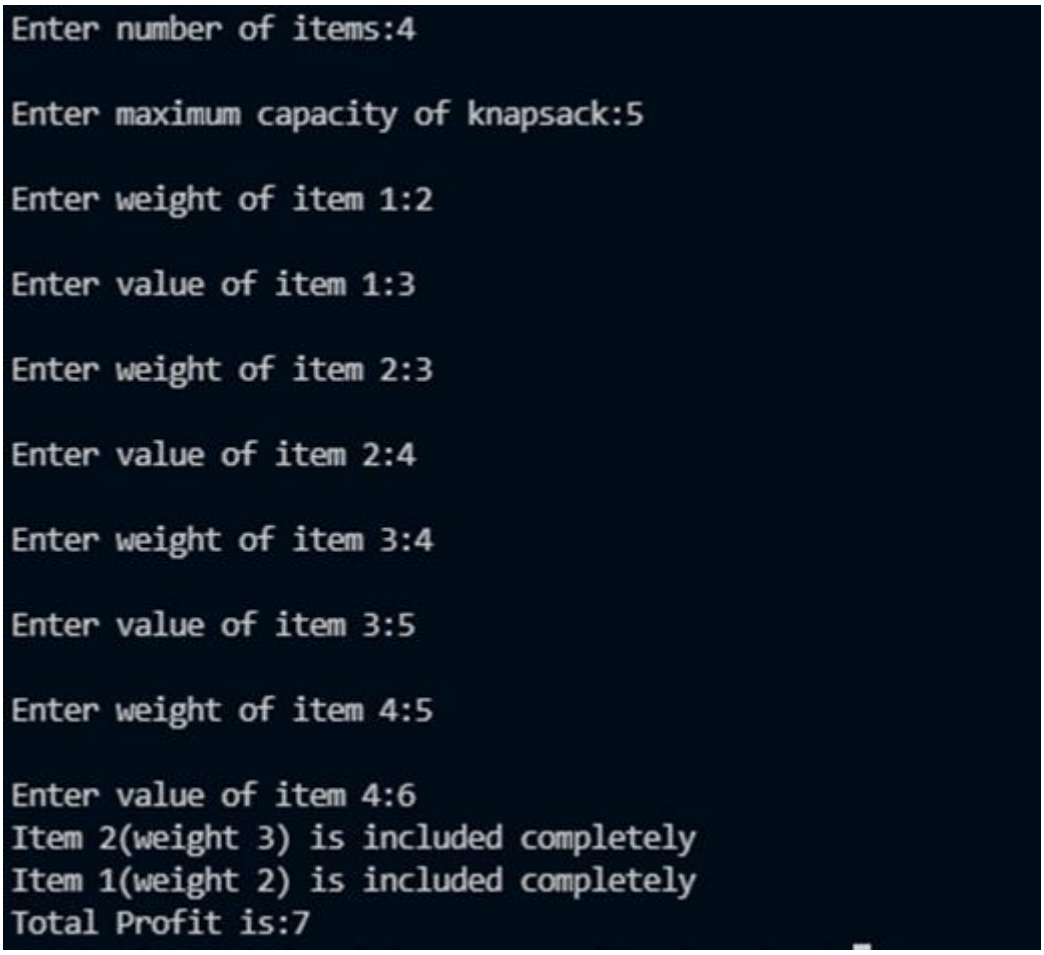
}

CODE:

```
#include <stdio.h>
int max(int a, int b)
{
   if(a>b)
         return a;
   else
         return b;
}
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n + 1][W + 1];
   for (i = 0; i<= n; i++)
   {
      for (w = 0; w <= W; w++)
      {
         if (i == 0 || w == 0)
            K[i][w] = 0;
         else if (wt[i - 1] <= w)
            K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]],K[i - 1][w]);
         else
            K[i][w] = K[i - 1][w];
      }
   }
i=n;
   int k=W;
   while(i>0 && k>0)
   {
         if(K[i][k]!=K[i-1][k])
         {
                  printf("Item %d(weight %d) is included completely\n",i,wt[i-1]);
```

```
                i=i-1;
                k=k-wt[i];
        }
        else
        {
                i=i-1;
        }
   }
printf("Total Profit is:%d",K[n][W]);
}
int main()
{
        int n,W;
        printf("Enter number of items:");
        scanf("%d",&n);
        printf("\nEnter maximum capacity of knapsack:");
        scanf("%d",&W);
        int value[n],wt[n];
        for(int i=0;i<n;i++)
        {
                printf("\nEnter weight of item %d:",i+1);
                scanf("%d",&wt[i]);
                printf("\nEnter value of item %d:",i+1);
                scanf("%d",&value[i]);
        }
        knapSack(W, wt, value, n);
      return 0;
}
```

OUTPUTS:



PERFORMANCE ANALYSIS:

Time Complexity is O(n*W). Space Complexity is O(n*W) as we are using extra 2-dimensional array of size n×W.

RESULT: 0/1 knapsack algorithm using dynamic programming has been implemented successfully.

PRELAB QUESTIONS:

1. Distinguish dynamic programming with divide-and-conquer technique.

A. Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem. Example: Quicksort, Mergesort, Binary search. Dynamic Programmingsplit a problem into subproblems, some of which are common, solve the subproblems, and combine the results for a solution to the original problem. Example: Matrix Chain Multiplication, Longest Common Subsequence.In divide and conquer, subproblems are independent. In Dynamic Programming , subproblems are not independent.

In divide and Conquer only one decision sequence is ever generated. In dynamic programming many decision sequences may be generated.

2. State principle of optimality.

A. Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions D_1, D_2, …, D_n. If this sequence is optimal, then the last k decisions, $1 < k < n$ must be optimal.

3. What is knapsack problem? How dynamic programming is useful for solving it?

A.Given a knapsack with maximum capacity *W*, and a set *S* consisting of *n* items

Each item *i* has some weight w_i and benefit value b_i(all $w_i$and *W* are integer values)

We needto pack the knapsack to achieve maximum total value of packed items.The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected. Summation of weights of all items should be less than weight of knapsack and we should try to maximize the profit. In a table say sDP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows. The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values >wi'. Now two possibilities can take place. Fill 'wi' in the given column. Do not fill 'wi' in the given column. Now we have to take a maximum of these two possibilities, formally if we do not fill 'ith' weight in 'jth' column then DP[i][j] state will be same as DP[i-1][j] but if we fill the weight, DP[i][j] will be equal to the value of 'wi'+ value of the column weighing 'j-wi' in the previous row. So we take the maximum of these two possibilities to fill the current state.

4. Find the minimum no. of deletions required to convert a string to palindrome?

A.
```
int lps(string str)
{
   int n = strlen(str);
   int L[n][n];
   for (int i = 0; i< n; i++)
      L[i][i] = 1;
   for (int cl = 2; cl <= n; cl++)
   {
      for (int i = 0;
i< n - cl + 1; i++)
      {
         int j = i + cl - 1;
         if (str[i] == str[j] && cl == 2)
            L[i][j] = 2;
         else if (str[i] == str[j])
            L[i][j] = L[i + 1][j - 1] + 2;
         else
```

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

    // length of longest palindromic subsequence
    return L[0][n - 1];
}
int minimumDeletions(string str)
{
    int n = strlen(str);;
    int len = lps(str);
    return (n - len);
}
```

5. Consider the simplest problem of finding the sum of set of elements. Design a simple dynamic programming solution for this problem.

A.

```
#include<stdio.h>

int sum(int a[],int n)

{

        int s[n];

        s[0]=a[0];

        for(int i=1;i<n;i++)

        {

                s[i]=s[i-1]+a[i];

        }

        return s[n-1];

}

int main()

{

        int arr[]={1,2,3,4,5,6,7,8};

        int n=sizeof(arr)/sizeof(arr[0]);

        int c=sum(arr,n);

        printf("Sum=%d",c);

}
```

LAB – 10

AIM OF THE EXPERIMENT:
To implement N-queens problem using backtracking.

ALGORITHM:

Algorithm place (r_2, c_2)

{

for (r_1 = 1 to r_2 -1)

{

c_1 = q[r_1]

if (c_1 == c_2 OR abs(r_1-r_2) == abs(c_1 – c_2))

return false

}

return true;

}

Algorithm nQueens (r)

{

for (c=1 to n)

{

if (place(r,c))

q[r]=c

if(r==n) displayQueens()

else nQueens(r+1)

}

}

CODE:

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

```
int board[20],count;
void displaysolution(int n)
{
	int i,j;
	printf("\nSolution %d:\n\n",++count);
	for(i=1;i<=n;++i)
		printf("\t%d",i);
	for(i=1;i<=n;++i)
	{
		printf("\n\n%d",i);
		for(j=1;j<=n;++j)
		{
			if(board[i]==j)
				printf("\tQ");
			else
				printf("\t-");
		}
	}
}
int place(int row,int column)
{
	int i;
	for(i=1;i<=row-1;++i)
	{
		if(board[i]==column)
			return 0;
		else
			if(abs(board[i]-column)==abs(i-row))
				return 0;
	}
```

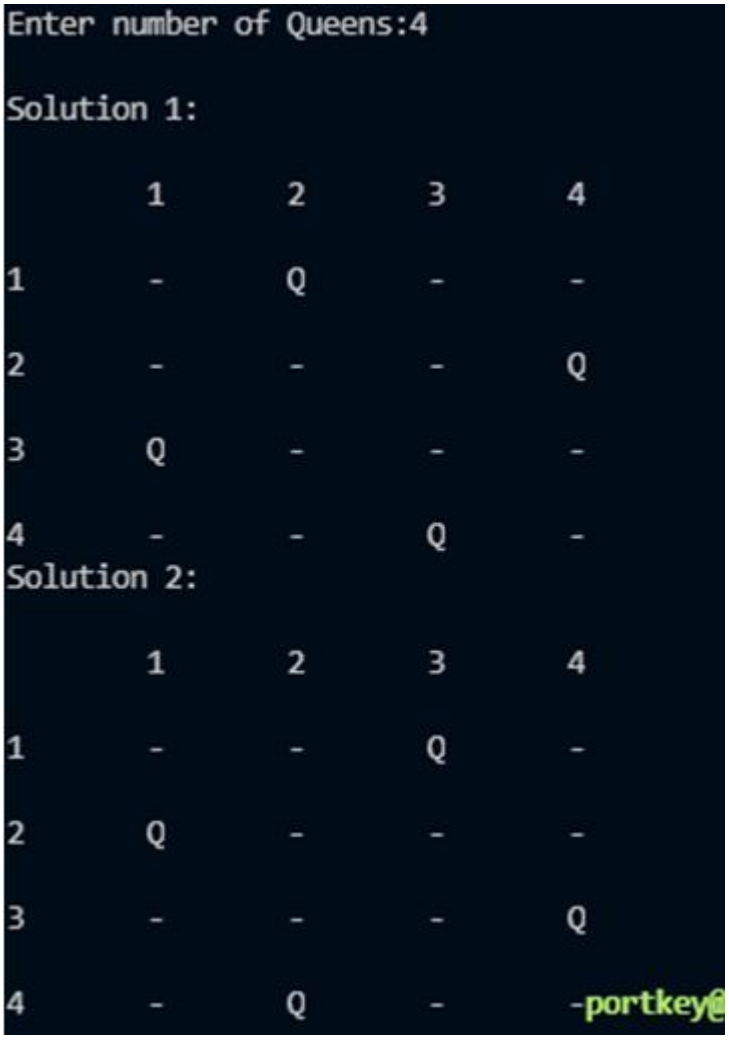
```
        return 1;
}
void queen(int row,int n)
{
        int column;
        for(column=1;column<=n;++column)
        {
                if(place(row,column))
                {
                        board[row]=column;
                        if(row==n)
                                displaysolution(n);
                        else
                                queen(row+1,n);
                }
        }
}
int main()
{
        int n,i,j;
        printf("Enter number of Queens:");
        scanf("%d",&n);
        queen(1,n);
        return 0;
}
```

OUTPUTS:

```
Enter number of Queens:5

Solution 1:

        1       2       3       4       5

1       Q       -       -       -       -

2       -       -       Q       -       -

3       -       -       -       -       Q

4       -       Q       -       -       -

5       -       -       -       Q       -
```

```
Solution 6:

        1       2       3       4       5

1       -       -       Q       -       -

2       -       -       -       -       Q

3       -       Q       -       -       -

4       -       -       -       Q       -

5       Q       -       -       -       -
```

```
Solution 10:

        1       2       3       4       5

1       -       -       -       -       Q

2       -       -       Q       -       -

3       Q       -       -       -       -

4       -       -       -       Q       -

5       -       Q       -       -       -
```



PERFORMANCE ANALYSIS:

The time complexity of n-queens problem is $O(n^n)$.

RESULT: N-queens problem using backtracking has been implemented successfully.

PRELAB QUESTIONS:

1. Define backtracking and applications of backtracking approach

A. Backtracking is nothing but the modified process of the brute force approach. where the technique systematically searches for a solution to a problem among all available options. It does so by assuming that the solutions are represented by vectors (v_1, ..., i_n) of values and by traversing through the domains of the vectors until the solutions is found.

Application of Backtracking

a. Optimization and tactical problems
b. Constraints Satisfaction Problem
c. Electrical Engineering
d. Robotics
e. Artificial Intelligence
f. Genetic and bioinformatics Algorithm
g. Materials Engineering
h. Network Communication
i. Solving puzzles and path

2. Define live node, dead node.

A. Live node is a node that has been generated but whose children have not yet been generated**.** Dead nodeis a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

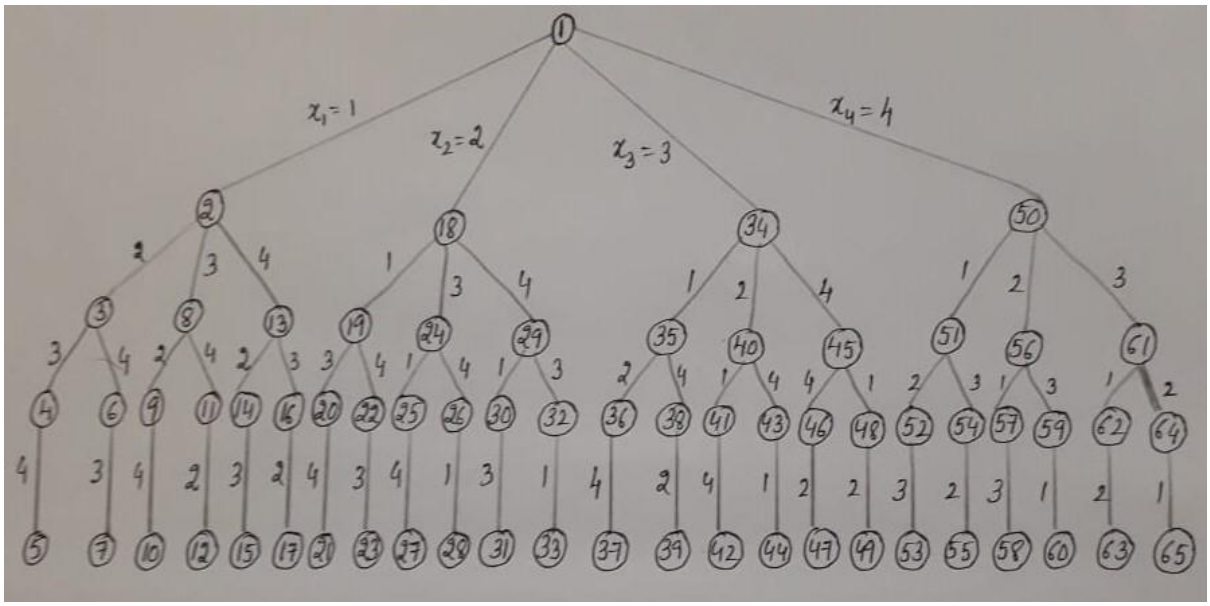
3. Define implicit and explicit constraints.

A. Explicit Constraints are rules that restrict each vector element to be chosen from the given set. Implicit constraints are rules that determine which each of the tuples in the solution space, actually satisfy the criterion function.

4. What is the time complexity of n-queens problem?

A. The time complexity of n-queens problem is $O(n^n)$.

5.Draw the State Space Tree of 4-queen's problem

A. 

LAB – 11

AIM OF THE EXPERIMENT: To implement Graph coloring problem using backtracking.

ALGORITHM:

```
Algorithm mColoring( k )
// k is the index of the next vertex to color.
{
        repeat
        {       // Generate all legal assignments for x[k]

                NextValue( k );                 // Assign to x[k] a legal color

                if ( x[k]=0 ) then return;      // No new color possible

                if ( k=n ) then     // At most m colors  have been used tocolor the n vertices
                write( x[1:n] );
                else   mColoring( k+1);
        } until ( false );
}
Algorithm NextValue( k )
// x[1],.....x[ k-1 ] have been assigned integer values in the range  [ 1 ,m ].
// A value for x[ k ] is determined in the range [ 0,m ]
{
        repeat
        {
                x[k] = ( x[k] +1) mod ( m+1 );          // Next highest color.
                if ( x[k]=0 ) then return;               // All colors have been used.
```

```
            for j = 1 to n do
            {
                if (( G[ k,j ] ≠ 0 )  and ( x[k] = x[j] ))  then
                    break;
            }
            if( j = n+1 ) then return; // Color found
        } until ( false );
}
```

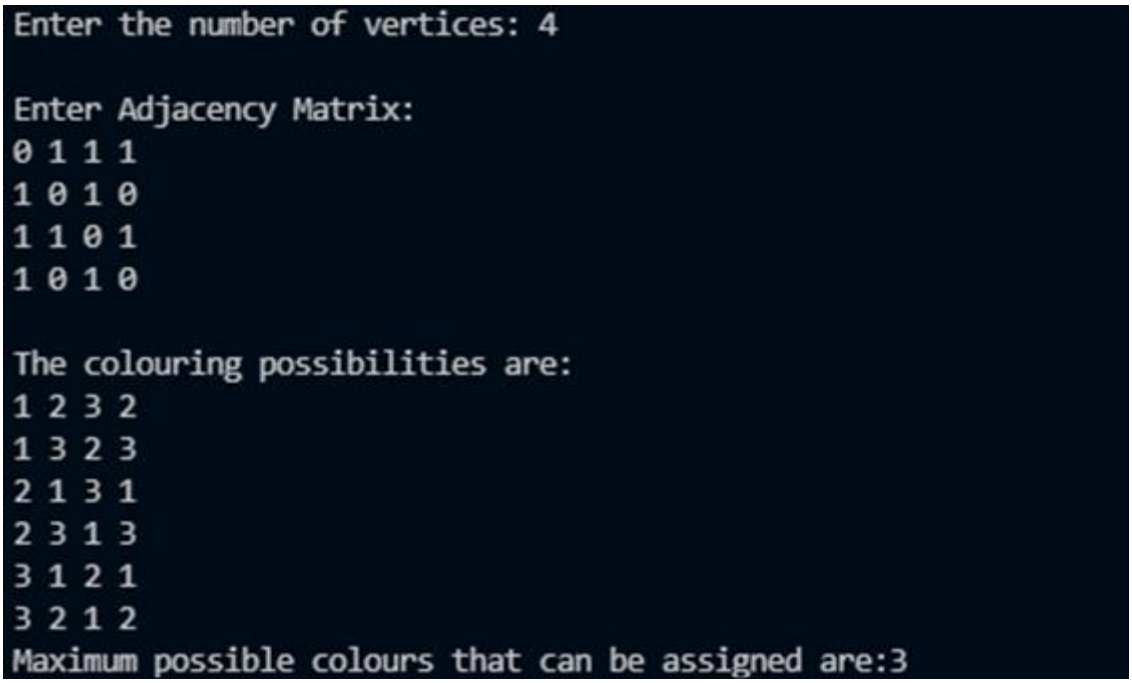
CODE:

```
#include<stdio.h>
static int m, n,c=0,count=0;
int g[50][50];
int x[50];
void nextValue(int k)
{
        int j;
        while(1)
        {
            x[k]=(x[k]+1)%(m+1);
            if(x[k]==0)
                return;
            for(j=1;j<=n;j++)
           {
                if(g[k][j]==1&&x[k]==x[j])
                    break;
            }
            if(j==(n+1))
                return;

        }
```

```
}
void GraphColoring(int k)
{
        int i;
        while(1)
        {
                nextValue(k);
                if(x[k]==0)
                        return;
                if(k==n)
                {
                        c=1;
                        for(i=1;i<=n;i++)
                                printf("%d ", x[i]);
                        count++;
                        printf("\n");

                }
                else
                GraphColoring(k+1);
        }
}
int main()
{
        int i, j;
        int temp;
        printf("Enter the number of vertices: " );
        scanf("%d", &n);
        printf("\nEnter Adjacency Matrix:\n");
```

```
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
  {
                    scanf("%d",&g[i][j]);
  }
 }
printf("\nThe colouring possibilities are:\n");
for(m=1;m<=n;m++)
{
  if(c==1)
  {
        break;
  }
GraphColoring(1);
 }
printf("Maximum possible colours that can be assigned are:%d", m-1);
printf("\nThe total number of solutions are:%d", count);
return 0;
}
```

OUTPUTS:



PERFORMANCE ANALYSIS:

Time Complexity: O(m^V).

There are total O(m^V) combination of colors. So time complexity is O(m^V). The upperbound time complexity remains the same but the average time taken will be less.

Space Complexity: O(V). Recursive Stack of graphColoring() function will require O(V) space.

RESULT: Graph coloring problem using backtracking has been implemented successfully.

PRELAB QUESTIONS:

1. Illustrate the backtracking technique to m-coloring graph.

A. The idea of backtacking approach is to assign colors one by one to different vertices, starting from the vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false.

2. Describe graph coloring problem with greedy approach

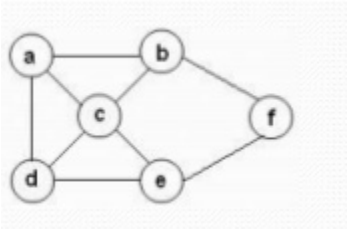
A. Color first vertex with first color.
Do following for remaining V-1 vertices.
a) Consider the currently picked vertex and color it with the

lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

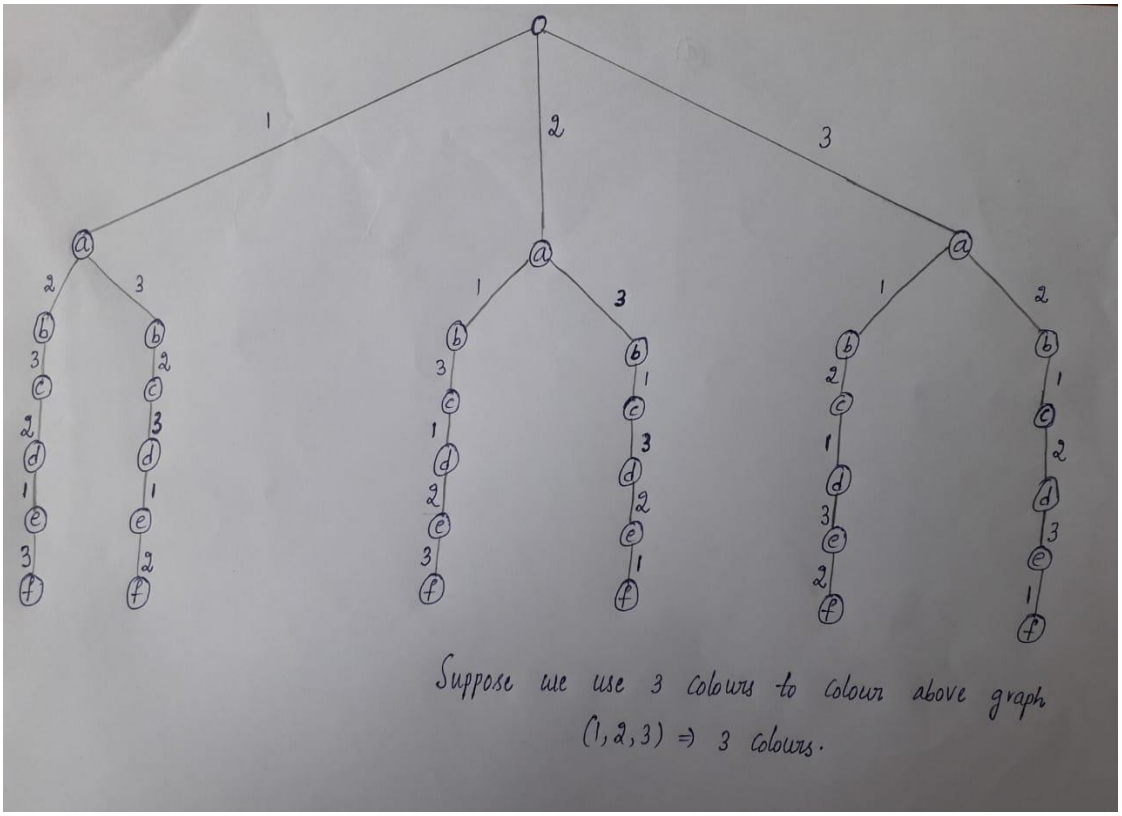
3. Define chromatic number

A. The chromatic number of a graph is the smallest number of colors needed to color the vertices of so that no two adjacent vertices share the same color

4. Draw the state space tree for the following graph



A.



LAB – 13

AIM OF THE EXPERIMENT: To implement 0/1 knapsack problem using branch and bounds.

ALGORITHM:

```
Algorithm ubound (cp,cw,k,m)
//cp is current profit
//cw is current weight
//k is index of last removed item
// m is knapsack capacity
{
        profit=cp;
        weight=cw;
        for(i=k+1;i<=n;i++)
        {
                if(weight +w[i]<=m)
                {
                        Weight+=w[i];
                        Profit-=p[i];
                }
        }
return profit;
}
Algorithm bound (cp,cw,k,m)
//cp is current profit
//cw is current weight
//k is index of last removed item
// m is knapsack capacity
```

```
{
        profit=cp;
        weight=cw;
        for(i=k+1;i<=n;i++)
        {
                weight+=w[i];
                if(weight < m)
                {
                        profit +=p[i];
                }
                else
                        Return (profit+(1-(weight – m)/w[i])*p[i]);
        }
return profit;
}
```

CODE:

```
//Program to implement 0/1 knapsack problem using branch and bound
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<math.h>
#define min(a,b) a<b?a:b;
int size;
int knapsack_weight;
bool * global_best;
int global_lb;
int global_lb_2;
typedef struct Item
{
        int profit;
```

```
        int weight;
        int idx;
}Item;
typedef struct Node
{
        int upper_bound;
        int lower_bound;
        int level;
        bool flag;
        int cp;
        int cw;
}Node;
typedef struct priority_queue
{
        Node *array;
        int queue_size;
}priority_queue;

priority_queue* create_queue()
{
        priority_queue* p = (priority_queue*)malloc(sizeof(priority_queue));
        p->array=(Node*)malloc(sizeof(Node)*1000);
        p->queue_size=0;
        return p;
}
void swap(Node *a, Node *b)
{
        Node temp=*a;
        *a=*b;
        *b=temp;
}
```

```
void heapify(priority_queue* p, int idx)
{
	int left=2*idx+1;
	int right=2*idx+2;
	int smallest=idx;
	if(left<p->queue_size&& p->array[smallest].lower_bound>=p->array[left].lower_bound)
		smallest=left;
	if(right<p->queue_size&& p->array[smallest].lower_bound>=p->array[right].lower_bound)
		smallest=right;
	if(smallest!=idx)
	{
		swap(&p->array[smallest],&p->array[idx]);
		heapify(p,smallest);
	}
}
void insert(priority_queue* p, Node element)
{
	p->queue_size++;
	int idx=p->queue_size-1;
	while(idx>0 &&element.lower_bound>=p->array[(idx-1)/2].lower_bound)
	{
		p->array[idx]=p->array[(idx-1)/2];
		idx=(idx-1)/2;
	}
	p->array[idx]=element;
}
void remove_(priority_queue* p)
{
	swap(&p->array[0],&p->array[p->queue_size-1]);
	p->queue_size--;
	heapify(p,0);
```

```
}
void sort(Item * arr)
{
	int i,j;
	for(i=0;i<size;i++)
	{
		for(j=i+1;j<size;j++)
		{
			if((double)arr[i].profit/arr[i].weight<(double)arr[j].profit/arr[j].weight)
			{
				Item temp=arr[i];
				arr[i]=arr[j];
				arr[j]=temp;
			}
		}
	}
}
int calculate_upper_bound(int cp,intcw,intidx, Item* arr)
{
	int profit=cp;
	int weight=cw,i;
	for(i=idx;i<size;i++)
	{
		if(weight+arr[i].weight<=knapsack_weight)
		{
			weight+=arr[i].weight;
			profit-=arr[i].profit;
		}
		else
		{
			profit-=(double)(knapsack_weight-weight)/arr[i].weight*arr[i].profit;
```

```
                        break;
                }
        }
        return profit;
}
int calculate_lower_bound(int cp,intcw, int idx, Item* arr)
{
        int profit=cp;
        int weight=cw;
        int i;
        for(i=idx;i<size;i++)
        {
                if(weight+arr[i].weight<=knapsack_weight)
                {
                        weight+=arr[i].weight;
                        profit-=arr[i].profit;
                }
                else
                {
                        break;
                }
        }
        return profit;
}
void knapsack(Item* arr)
{
        sort(arr);
        priority_queue* p=create_queue();
        Node current,left,right;
        current.upper_bound=current.lower_bound=0;
        current.cp=current.cw=0;
```

```
	current.level=current.flag=0;
	bool select[size];
	int i;
	global_best=(bool*)malloc(sizeof(bool)*(size+1));
	for(i=0;i<size;i++)
		select[i]=false;
	insert(p,current);

	while(p->queue_size !=0)
	{
		current=p->array[0];
		remove_(p);
		if((current.upper_bound>global_lb || current.upper_bound>global_lb_2)
&&current.level!=0)
			continue;
		if(current.level!=0)
		{
			select[current.level-1]=current.flag;
		}
		if(current.level==size)
		{
			if(current.lower_bound< global_lb_2)
			{
				for(i=0;i<size;i++)
				{
					global_best[arr[i].idx]=select[i];
				}
			}
			global_lb_2=min(global_lb_2,current.lower_bound);
			continue;
		}
```

```
        right.upper_bound=calculate_upper_bound(current.cp,current.cw,current.level+1,arr);

        right.lower_bound=calculate_lower_bound(current.cp,current.cw,current.level+1,arr);
                right.flag=false;
                right.level=current.level+1;
                right.cp=current.cp;
                right.cw=current.cw;
                global_lb=min(global_lb,right.lower_bound);
                if(current.cw+arr[current.level].weight<=knapsack_weight)
                {
                        left.upper_bound=calculate_upper_bound(current.cp-
arr[current.level].profit,current.cw+arr[current.level].weight,current.level+1,arr);
                        left.lower_bound=calculate_lower_bound(current.cp-
arr[current.level].profit,current.cw+arr[current.level].weight,current.level+1,arr);
                        left.flag=true;
                        left.level=current.level+1;
                        left.cw=current.cw+arr[current.level].weight;
                        left.cp=current.cp-arr[current.level].profit;
                        global_lb=min(global_lb,left.lower_bound);
                }
                else
                {
                        left.lower_bound=left.upper_bound=0;
                        left.level++;
                }
                if(right.upper_bound<= global_lb&&global_lb !=0) insert(p,right);
                if(left.upper_bound<=global_lb&&global_lb!=0) insert(p,left);
        }
        printf("The elements taken into the knapsack are(The resultant vector is):\n");
        for(i=0;i<size;i++)
        {
```

```
                //global_best[i]==0? printf("0 "): printf("1 ");
                if(global_best[i]==0)
                        printf("0 ");
                else
                        printf("1 ");
        printf(" ");
        }
        float profit=abs(global_lb_2);
        printf("\nMaximum profit is:%f",profit);
        free(p->array);
        free(global_best);
}
int main()
{
        int i;
        printf("Enter the number of items:\n");
        scanf("%d",&size);
        Item arr[size];
        for(i=0;i<size;i++)
        {
                printf("Enter weight and profit of Item %d:\n",i+1);
                scanf("%d %d",&arr[i].weight,&arr[i].profit);
                arr[i].idx=i;
        }
        printf("Enter the knapsack capacity:");
        scanf("%d",&knapsack_weight);
        global_lb=0;
        global_lb_2=0;
        knapsack(arr);
        return 0;
}
```

OUTPUTS:

```
Enter the number of items:
4
Enter weight and profit of Item 1:
2 3
Enter weight and profit of Item 2:
3 4
Enter weight and profit of Item 3:
4 5
Enter weight and profit of Item 4:
5 6
Enter the knapsack capacity:5
The elements taken into the knapsack are(The resultant vector is):
1  1  0  0
```

```
Enter the number of items:
4
Enter weight and profit of Item 1:
2 10
Enter weight and profit of Item 2:
3 12
Enter weight and profit of Item 3:
4 20
Enter weight and profit of Item 4:
5 25
Enter the knapsack capacity:12
The elements taken into the knapsack are(The resultant vector is):
0  1  1  1
```

PERFORMANCE ANALYSIS:

The time complexity of 0/1 knapsack using branch and bound is $O(2^n)$.

RESULT: 0/1 knapsack problem using branch and bounds has been implemented successfully.

LAB – 14

AIM OF THE EXPERIMENT: To implement Travelling Salesman problem using branch and bounds.

ALGORITHM:

1. Reduce the given cost matrix by applying row reduction (a row is reduced if it contains atleast 1 zero) .Take minimum element and subtract from all elements of row if is not reduced.
2. Apply column reduction (a column is reduced if it contains atleast 1 zero) .Take minimum element and subtract from all elements of column if is not reduced.
3. Obtain cumulative reduced sum by adding sum of row reduction elements and column reduction elements.
4. Let A is reduced cost for node R. Let S be child of R such that tree edge (R,S) corresponds to including edge <i, j> in tour. If S is not leaf node calculate reduced cost matrix as follows:
    - Change all entries in row I and column j of A to ∞.
    - Set A(j,1) to ∞
    - Reduce all columns and rows. Let r be total amount subtracted to reduce the matrix.
    - Find c(S) = c(R) + A(i,j) + r;
5. Repeat steps 4 until all nodes are visited

CODE:

```
#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

#include<string.h>

//#include<limits.h>

#define MAX 99999999

#define N 20

int final_path[11]; //final_path[] stores the path of the travelling salesman


bool visited[11];


int final_res = MAX; //Stores the final minimum cost of shortest tour
```

```
void copyToFinal(int curr_path[], int n) //Copying temporary solution to the final solution
{
	int i;
	for (i=0; i<n; i++)
		final_path[i] = curr_path[i];
	final_path[N] = curr_path[0];
}

int firstMin(int adj[N][N], int i, int n)  //Function to find the minimum edge cost
{
	int k;
	int min = MAX;
	for (k=0; k<n; k++)
		if (adj[i][k]<min &&i != k)
			min = adj[i][k];
	return min;
}

int secondMin(int adj[N][N], int i, int n) //function to find the second minimum edge cost
{
	int j;
	int first = MAX;
	int second = MAX;
	for (j=0; j<n; j++)
	{
		if (i == j)
			continue;

		if (adj[i][j] <= first)
		{
```

```
                second = first;
                first = adj[i][j];
            }
            else if (adj[i][j] <= second &&adj[i][j] != first)
                second = adj[i][j];
        }
        return second;
}

void TSPRec(int adj[N][N], int curr_bound, int curr_weight,int level, int curr_path[], int n)
{
        int i,j;
        if (level==n)
        {
            if (adj[curr_path[level-1]][curr_path[0]] != 0)
            {
                int curr_res = curr_weight + adj[curr_path[level-1]][curr_path[0]];
                if (curr_res<final_res)
                {
                    copyToFinal(curr_path, n);
                    final_res = curr_res;
                }
            }
            return;
        }
        for (i=0; i<n; i++)
        {
            if (adj[curr_path[level-1]][i] != 0 && visited[i] == false)
            {
                int temp = curr_bound;
                curr_weight += adj[curr_path[level-1]][i];
```

```
                        if (level==1)
                        curr_bound -= ((firstMin(adj, curr_path[level-1], n) + firstMin(adj, i,n))/2);
                        else
                        curr_bound -= ((secondMin(adj, curr_path[level-1], n) + firstMin(adj, i,
n))/2);
                        if (curr_bound + curr_weight<final_res)
                        {
                                curr_path[level] = i;
                                visited[i] = true;

                                TSPRec(adj, curr_bound, curr_weight, level+1, curr_path, n);
                        }
                        curr_weight -= adj[curr_path[level-1]][i];
                        curr_bound = temp;

                        memset(visited, false, sizeof(visited));
                        for (j=0; j<=level-1; j++)
                                visited[curr_path[j]] = true;
                }
        }
}

void TSP(int adj[N][N], int n)
{
        int i;
        int curr_path[n+1];
        int curr_bound = 0;
        memset(curr_path, -1, sizeof(curr_path));
        memset(visited, 0, sizeof(curr_path));

        for (i=0; i<n; i++)
```

```
            curr_bound += (firstMin(adj, i, n) + secondMin(adj, i, n));

        curr_bound = (curr_bound&1)? curr_bound/2 + 1 :curr_bound/2;

        visited[0] = true;
        curr_path[0] = 0;

        TSPRec(adj, curr_bound, 0, 1, curr_path, n);
}

int main()
{
        int n;
        int adj[N][N];
        printf("\n Enter no of vertices: ");
        scanf("%d", &n);
        int i, j;
        printf("\n");
        printf("Enter cost matrix:\n");
        for (i = 0; i<n; i++){
            //printf(" ");
            for (j = 0; j<n; j++){
            scanf("%d", &adj[i][j]);
          }
        }

        TSP(adj,n);

        printf("\n Minimum cost for TSP tour is: %d\n", final_res);
        printf("\n Path Taken is: ");
        for (i=0; i<n; i++)
```

```
            printf("%d --> ", final_path[i]);
        printf("%d",final_path[i]);


        return 0;
}
```

OUTPUTS:

```
 Enter no of vertices: 5

Enter cost matrix:
9999 5 4 5 8
5 9999 2 2 3
4 2 9999 3 5
5 2 3 9999 5
8 3 5 2 9999

 Minimum cost for TSP tour is: 16
```

```
  Enter no of vertices: 4

 Enter cost matrix:
 9999 4 1 3
 4 9999 2 1
 1 2 9999 5
 3 1 5 9999

  Minimum cost for TSP tour is: 7
```

PERFORMANCE ANALYSIS:

The time complexity of Travelling Salesman problem using branch and bounds is $O(2^n)$.

RESULT: Travelling Salesman problem using branch and bounds has been implemented successfully.