

ADITYA INSTITUTE OF TECHNOLOGY AND MANAGEMENT
(Autonomous)

(Approved by AICTE, Recognized Under 2(f) & 12(b) of UGC Permanently Affiliated to JNTU GV, Vizianagaram)

COMPUTER SCIENCE AND ENGINEERING (AI & ML)



2023-2024 A.Y.

B. Tech., III Year - I Semester (AR20)

LABORATORY MANUAL

for

Design Analysis of Algorithms Lab (20CAL303)

Prepared by

Anil Kumar Prathipati, Assistant Professor, CSM

LIST OF LAB EXERCISES

Sl. No	Name of the Program	Page Number
1	Develop a program and measure the running time for Binary Search with Divide and Conquer.	2
2	Develop a program and measure the running time for Merge Sort with Divide and Conquer.	4
3	Develop a program and measure the running time for Quick Sort with Divide and Conquer.	7
4	Develop a program and measure the running time for estimating minimum-cost spanning Trees with Greedy Method.	10
5	Develop a program and measure the running time for estimating Single Source Shortest Paths with Greedy Method.	16
6	Develop a program and measure the running time for optimal Binary search trees with Dynamic Programming.	18
7	Develop a program and measure the running time for identifying solution for traveling salesperson problem with Dynamic Programming.	20
8	Develop a program and measure the running time for identifying solution for 8-Queens problem with Backtracking.	22
9	Develop a program and measure the running time for Graph Coloring with Backtracking.	25
10	Develop a program and measure the running time to generate solution of Hamiltonian Cycle problem with Backtracking.	28
11	Develop a program and measure the running time running time to generate solution of Knapsack problem with Backtracking.	32

EXPERIMENT-1

AIM:

Develop a program and measure the running time for Binary Search with Divide and Conquer.

Logic / Algorithm:

- The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete.
- If the key element is less than the middle element, then the search continues with the first half of the list.
- If the key element is greater than the middle element, then the search continues with the second half of the list.
- This process continues until the key element is found or the search fails indicating that the key is not there in the list.

Time Complexity is $O(\log n)$

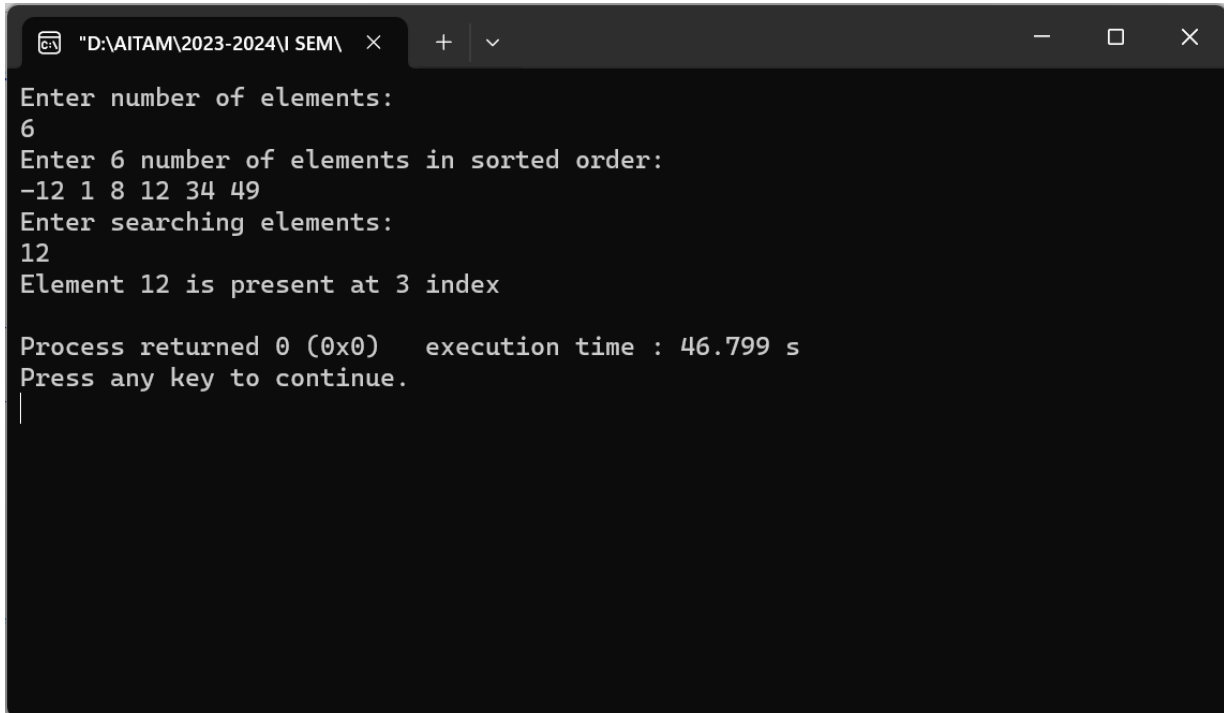
Source Code:

```
#include <stdio.h>

int binarySearch(int arr[], int l, int r, int x)
{
    if (l <= r)
    {
        int m = l + (r - l) / 2;
        if (arr[m] == x)
            return m;
        else if (arr[m] < x)
            return binarySearch(arr, m + 1, r, x);
        else
            return binarySearch(arr, l, m - 1, x);
    }
    return -1;
}

int main()
{
    int arr[20], n, x, i, result;
    printf("Enter number of elements:\n");
    scanf("%d", &n);
    printf("Enter %d number of elements in sorted order:\n", n);
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);
    printf("Enter searching elements:\n");
    scanf("%d", &x);
    result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element %d is not present in array\n", x) : printf("Element %d is present at %d index\n", x, result);
    return 0;
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  x  +  v
Enter number of elements:
6
Enter 6 number of elements in sorted order:
-12 1 8 12 34 49
Enter searching elements:
12
Element 12 is present at 3 index

Process returned 0 (0x0)   execution time : 46.799 s
Press any key to continue.
|
```

EXPERIMENT-2

AIM:

Develop a program and measure the running time for Merge Sort with Divide and Conquer.

Logic / Algorithm:

The merge sort algorithm works as follows-

Step 1: If the length of the list is 0 or 1, then it is already sorted, otherwise,

Step 2: Divide the unsorted list into two sub-lists of about half the size.

Step 3: Again sub-divide the sub-list into two parts. This process continues until each element in the list becomes a single element.

Step 4: Apply merging to each sub-list and continue this process until we get one sorted list.

Time Complexity is $O(n \log n)$

Source Code:

```
#include<stdio.h>

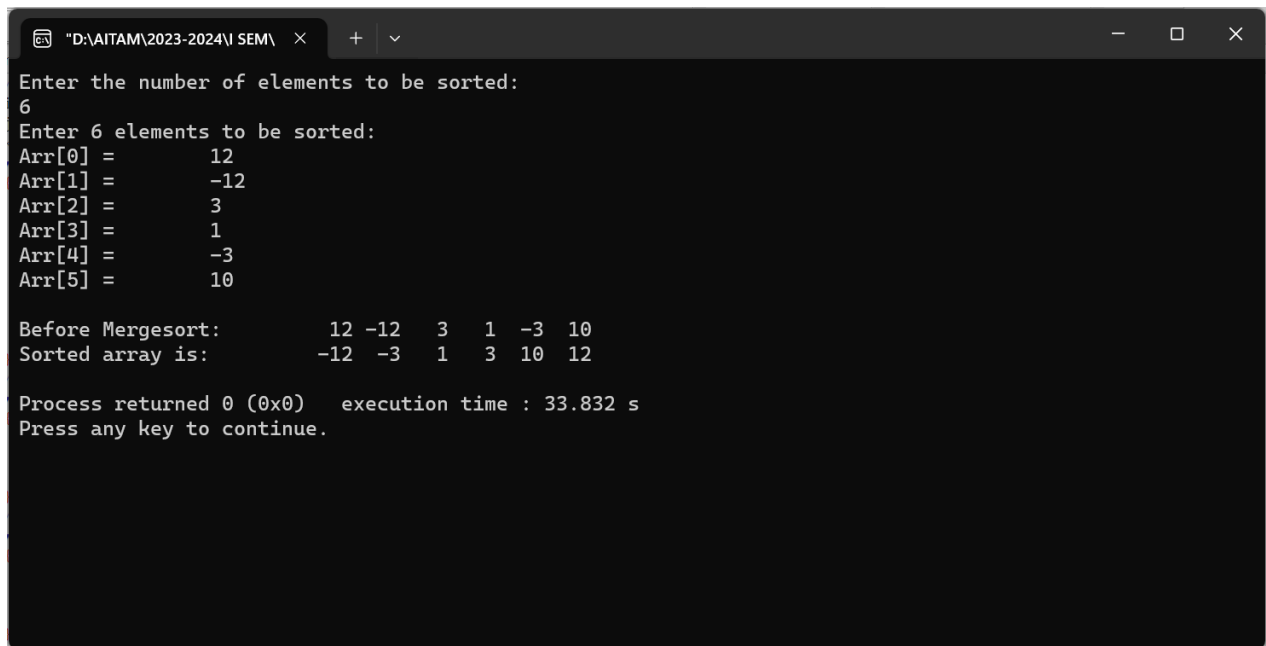
void divide(int arr[], int l, int r);
void merge(int arr[], int l, int m, int r);
int n;
int main()
{
    int arr[10], i;
    printf("Enter the number of elements to be sorted:\n");
    scanf("%d", &n);
    printf("Enter %d elements to be sorted: \n", n);
    for (i = 0; i < n; i++)
    {
        printf("Arr[%d] = \t", i);
        scanf("%d", &arr[i]);
    }
    printf("\nBefore Mergesort:\t");
    for (i = 0; i < n; i++)
    {
        printf("%4d", arr[i]);
    }
    divide(arr, 0, n - 1);
    printf("\nSorted array is:\t");
    for (i = 0; i < n; i++)
    {
        printf("%4d", arr[i]);
    }
    printf("\n");
    return 0;
}

void divide(int arr[], int l, int r)
{
    if (l < r) {
        // Finding mid element
```

```
    int m = (l + r) / 2;
    // Recursively sorting both the halves
    divide(arr, l, m);
    divide(arr, m + 1, r);
    // Merge the array
    merge(arr, l, m, r);
}
}
```

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k, n1, n2;
    n1 = m - l + 1;
    n2 = r - m;
    // Create temp arrays
    int L[n1], R[n2];
    // Copy data to temp array
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    // Merge the temp arrays
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    // Copy the remaining elements of L[]
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    // Copy the remaining elements of R[]
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  x  +  v  -  □  x
Enter the number of elements to be sorted:
6
Enter 6 elements to be sorted:
Arr[0] =      12
Arr[1] =     -12
Arr[2] =       3
Arr[3] =       1
Arr[4] =      -3
Arr[5] =      10

Before Mergesort:      12 -12  3  1  -3  10
Sorted array is:      -12 -3  1  3  10  12

Process returned 0 (0x0)   execution time : 33.832 s
Press any key to continue.
```

EXPERIMENT-3**AIM:**

Develop a program and measure the running time for Quick Sort with Divide and Conquer.

Logic / Algorithm:

- The entire list is partitioned into two sub-list one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another sub-list holds values greater than the pivot value.
- Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.

Example:

Input: 12 6 18 4 9 8 2 15

- In this example, we use the first number, say 12, which is called the pivot (rotate) element.
- Let 'i' be the position of the second element and 'j' be the position of the last element. i.e. i = 2 and j = 8, in this example.
- Assume that $a[n+1] = \infty$, where 'a' is an array of size n.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	<u>i</u>	<u>j</u>
12	6	18	4	9	8	2	15	α	2	8

- First scan the list from left to right (from i to j) can compare each and every element with the pivot. This process continues until an element found which is greater than or equal to pivot element. If such an element found, then that element position becomes the value of 'i'.
- Now scan the list from right to left (from j to i) and compare each and every element with the pivot. This process continues until an element found which is less than or equal to pivot element. If such an element finds then that element's position become 'j' value.
- Now compare 'i' and 'j'. If $i < j$, then swap $a[i]$ and $a[j]$. Otherwise swap pivot element and $a[j]$.

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	<u>i</u>	<u>j</u>
(12)	6	18	4	9	8	2	15	α	2	8
(12)	6	18	4	9	8	2	15	α	3	7
12	6	2	4	9	8	18	15	α	7	6

Since $i = 7 \nless j = 6$, then swap pivot element and 6th element (j^{th} element), we get

8	6	2	4	9	(12)	18	15
---	---	---	---	---	------	----	----

Thus pivot reaches its original position. The elements on left to the right pivot are smaller than pivot (12) and right to pivot are greater pivot (12).

8	6	2	4	9	(12)	18	15
<hr/>						<hr/>	
Sublist 1						Sublist 2	

Now take sub-list1 and sub-list2 and apply the above process recursively, at last we get sorted list.

Time Complexity is $O(n \log n)$ – Best and Average cases, and $O(n^2)$ – worst case.

Source Code:

```
#include <stdio.h>
void quicksort (int [], int, int);
int partition(int [], int, int);
int main()
{
    int a[50];
    int n, i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    if(n<=0)
    {
        printf("\nInvalid Data");
        return 0;
    }
    printf("\nEnter the %d elements to be sorted:\n",n);
    for (i = 0; i < n; i++)
    {
        printf("a[%d]=\t",i);
        scanf("%d", &a[i]);
    }
    quicksort(a, 0, n - 1);
    printf("After applying quick sort\n");
    for (i = 0; i < n; i++)
    {
        printf("%d ", a[i]);
    }
    printf("\n");

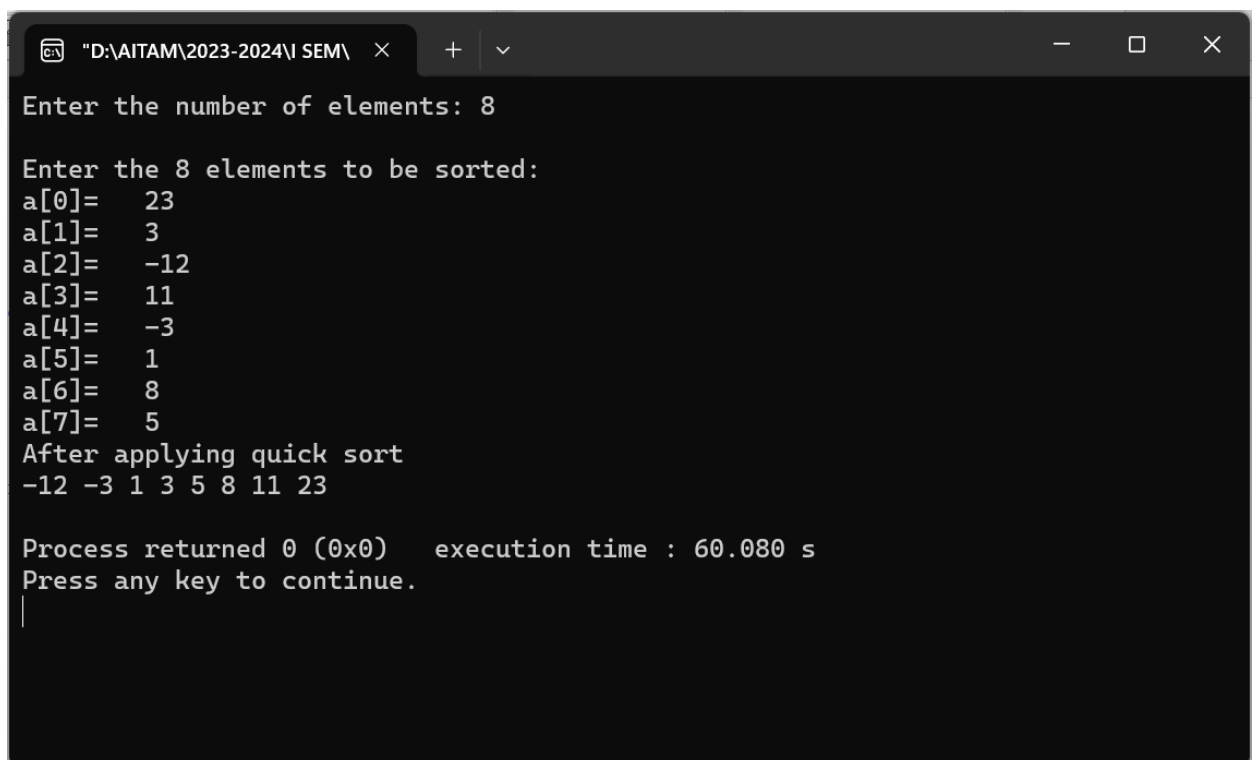
    return 0;
}
```

```
int partition(int a[], int low, int high)
{
    int pvt, i, j, temp;
    if (low < high)
    {
        pvt = low;
        i = low;
        j = high;
        while (i < j)
        {
            while (a[i] <= a[pvt] && i <= high)
            {
                i++;
            }
            while (a[j] > a[pvt] && j >= low)
            {
                j--;
            }
        }
    }
}
```

```
        if (i < j)
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    temp = a[j];
    a[j] = a[pvt];
    a[pvt] = temp;
    return j;
}

void quicksort(int a[], int low, int high)
{
    if(low<high)
    {
        int j=partition(a,low,high);
        quicksort(a, low, j - 1);
        quicksort(a, j + 1, high);
    }
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  x + v
Enter the number of elements: 8
Enter the 8 elements to be sorted:
a[0]= 23
a[1]= 3
a[2]= -12
a[3]= 11
a[4]= -3
a[5]= 1
a[6]= 8
a[7]= 5
After applying quick sort
-12 -3 1 3 5 8 11 23

Process returned 0 (0x0)   execution time : 60.080 s
Press any key to continue.
|
```

EXPERIMENT-4**AIM:**

Develop a program and measure the running time for estimating minimum-cost spanning Trees with Greedy Method.

Logic / Algorithm (Prim's):

```

1  float Prim(int E[][SIZE], float cost[][SIZE], int n, int t[][2])
11 {
12  int near[SIZE], j, k, L;
13  let (k,L) be an edge of minimum cost in E;
14  float mincost = cost[k][L];
15  t[1][1] = k; t[1][2] = L;
16  for (int i=1; i<=n; i++) // Initialize near.
17    if (cost[i][L] < cost[i][k]) near[i] = L;
18    else near[i] = k;
19  near[k] = near[L] = 0;
20  for (i=2; i <= n-1; i++) { // Find n-2 additional
21    // edges for t.
22    let j be an index such that near[j]!=0 and
23    cost[j][near[j]] is minimum;
24    t[i][1] = j; t[i][2] = near[j];
25    mincost = mincost + cost[j][near[j]];
26    near[j]=0;
27    for (k=1; k<=n; k++) // Update near[.
28      if ((near[k]!=0) &&
29          (cost[k][near[k]]>cost[k][j]))
30        near[k] = j;
31  }
32  return(mincost);
33 }
```

Time Complexity is $O(v^2)$, where v is number of vertices.

Source Code:

```

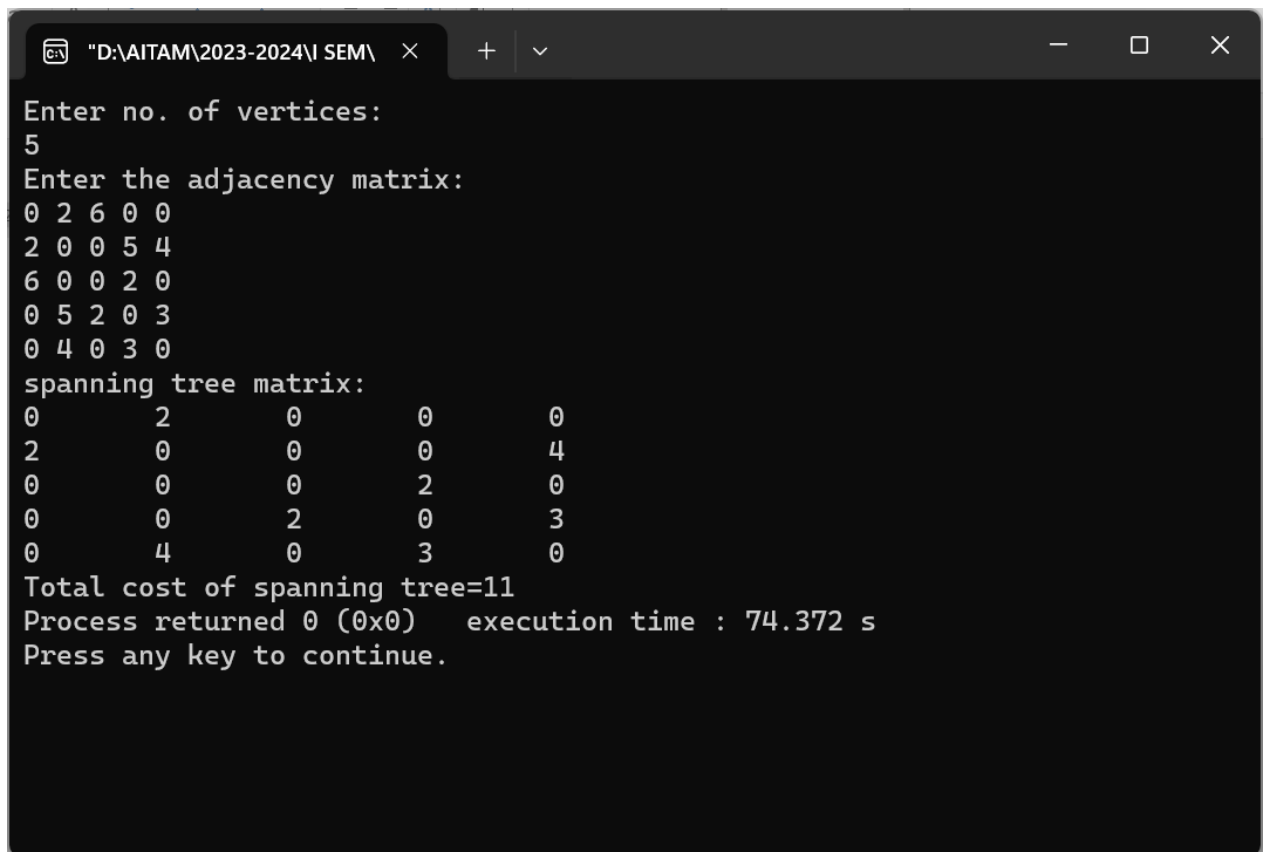
#include<stdio.h>
#include<stdlib.h>
#define infinity 999
#define MAX 20
int G[MAX][MAX],spanning[MAX][MAX],n;
int prims();
int main()
{
  int i,j,total_cost;
  printf("Enter no. of vertices:\n");
  scanf("%d",&n);
  printf("Enter the adjacency matrix:\n");
  for(i=0;i<n;i++)
    for(j=0;j<n;j++)
      scanf("%d",&G[i][j]);
  total_cost=prims();
}
```

```
printf("spanning tree matrix:\n");
for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
        printf("%d\t",spanning[i][j]);
    printf("\n");
}
printf("Total cost of spanning tree=%d",total_cost);
return 0;
}

int prims()
{
    int u,v,min_distance,distance[MAX],near[MAX];
    int visited[MAX],no_of_edges,i,j;
    int min_cost;
    //create cost[][] matrix,spanning[][]
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
            if(G[i][j]==0)
                G[i][j]=infinity;
            spanning[i][j]=0;
        }
    //Initialize visited[],distance[] and near[]
    distance[0]=0;
    visited[0]=1;
    for(i=1;i<n;i++)
    {
        distance[i]=G[0][i];
        near[i]=0;
        visited[i]=0;
    }
    min_cost=0;    //cost of spanning tree
    no_of_edges=n-1;    //no. of edges to be added
    while(no_of_edges>0)
    {
        //find the vertex at minimum distance from the tree
        min_distance=infinity;
        for(i=1;i<n;i++)
            if(visited[i]==0&&distance[i]<min_distance)
            {
                v=i;
                min_distance=distance[i];
            }
        u=near[v];
        //insert the edge in spanning tree
        spanning[u][v]=distance[v];
        spanning[v][u]=distance[v];
        no_of_edges--;
        visited[v]=1;
    }
}
```

```
//updated the distance[] array
for(i=1;i<n;i++)
    if(visited[i]==0&&distance[i]>G[v][i])
    {
        distance[i]=G[v][i];
        near[i]=v;
    }
    min_cost=min_cost+G[u][v];
}
return(min_cost);
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM" x + v
Enter no. of vertices:
5
Enter the adjacency matrix:
0 2 6 0 0
2 0 0 5 4
6 0 0 2 0
0 5 2 0 3
0 4 0 3 0
spanning tree matrix:
0 2 0 0 0
2 0 0 0 4
0 0 0 2 0
0 0 2 0 3
0 4 0 3 0
Total cost of spanning tree=11
Process returned 0 (0x0) execution time : 74.372 s
Press any key to continue.
```

Logic / Algorithm(Kruskal's):

```

float Kruskal(int E[][SIZE], float cost[][SIZE], int n, int t[][2])
{
    int parent[SIZE];
    construct a heap out of the edge costs using Heapify;
    for (int i=1; i<=n; i++) parent[i] = -1;
    // Each vertex is in a different set.
    i = 0; float mincost = 0.0;
    while ((i < n-1) && (heap not empty)) {
        delete a minimum cost edge (u,v) from the heap
        and reheapify using Adjust;
        int j = Find(u); int k = Find(v);
        if (j != k) {
            i++;
            t[i][1] = u; y[i][2] = v;
            mincost += cost[u][v];
            Union(j, k);
        }
    }
    if (i != n-1) cout << "No spanning tree" << endl;
    else return(mincost);
}

```

Time Complexity is $O(E \log E)$, where E is number of edges.

Source Code:

```

#include <stdio.h>
#include <stdlib.h>

int kruskals();
int find(int);
int funion(int , int);

int i,j,k,a,b,u,v,n,noe=1;
int min, mincost=0, cost[9][9], parent[9];

int main()
{
    printf("Enter the no. of vertices:\n");
    scanf("%d",&n);
    printf("Enter the cost adjacency matrix:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]<0)
            {

```

```
        printf("-ve edges are not allowed");
        exit(0);
    }
    if(cost[i][j]==0)
        cost[i][j]=999;

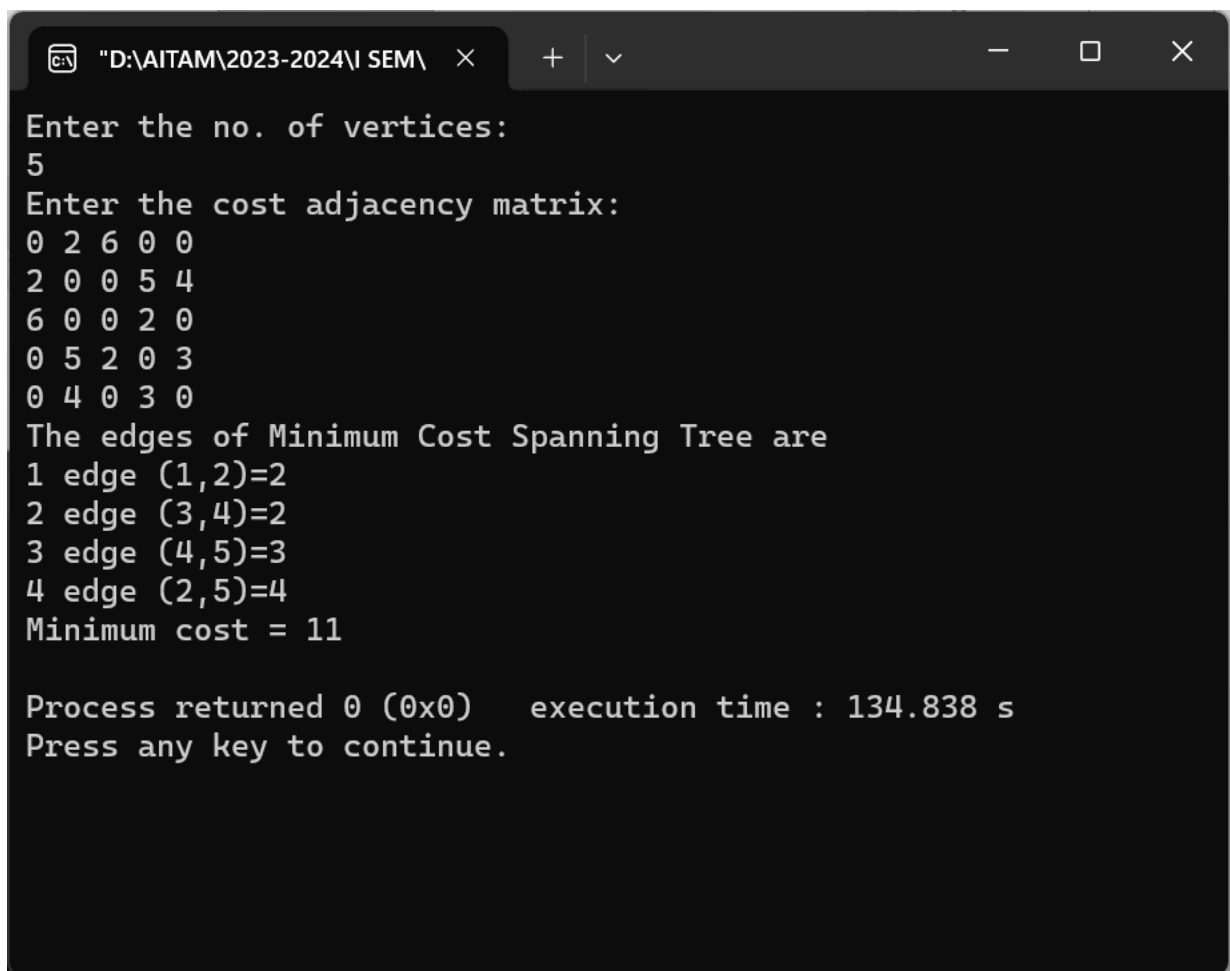
    }
    parent[i]=-1;
}
printf("The edges of Minimum Cost Spanning Tree are\n");
mincost=kruskals();
printf("Minimum cost = %d\n",mincost);
return 0;
}

int kruskals()
{
    while(noe < n)
    {
        for(i=1,min=999;i<=n;i++)
        {
            for(j=1;j <= n;j++)
            {
                if(cost[i][j] < min)
                {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            }
        }
        u=find(u);
        v=find(v);
        if(funion(u,v))
        {
            printf("%d edge (%d,%d)=%d\n",noe++,a,b,min);
            mincost +=min;
        }
        cost[a][b]=cost[b][a]=999;
    }
    return mincost;
}

int find(int i)
{
    while(parent[i] != -1)
        i=parent[i];
    return i;
}
```

```
int funion(int i,int j)
{
    if(i!=j)
    {
        parent[j]=i;
        return 1;
    }
    return 0;
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  × + ▾ − □ ×

Enter the no. of vertices:
5
Enter the cost adjacency matrix:
0 2 6 0 0
2 0 0 5 4
6 0 0 2 0
0 5 2 0 3
0 4 0 3 0
The edges of Minimum Cost Spanning Tree are
1 edge (1,2)=2
2 edge (3,4)=2
3 edge (4,5)=3
4 edge (2,5)=4
Minimum cost = 11

Process returned 0 (0x0)    execution time : 134.838 s
Press any key to continue.
```


EXPERIMENT-5**AIM:**

Develop a program and measure the running time for estimating Single Source Shortest Paths with Greedy Method.

Logic / Algorithm (Dijkstra's):

```

ShortestPaths(int v, float cost[][SIZE], float dist[], int n)
{
    int u; bool S[SIZE];
    for (int i=1; i<= n; i++) { // Initialize S.
        S[i] = false; dist[i] = cost[v][i];
    }
    S[v]=true; dist[v]=0.0; // Put v in S.
    for (int num = 2; num < n; num++) {
        // Determine n-1 paths from v.
        choose u from among those vertices not
        in S such that dist[u] is minimum;
        S[u] = true; // Put u in S.
        for (int w=1; w<=n; w++) //Update distances.
            if (( S[w]=false) && (dist[w] > dist[u] + cost[u][w]))
                dist[w] = dist[u] + cost[u][w];
    }
}

```

Time Complexity is $O(E \log V)$, where E is number of edges and v is number of vertices.

Source Code:

```

#include "stdio.h"
#define infinity 999
void dij(int n,int v,int cost[10][10],int dist[])
{
    int i,u,count,w,flag[10],min;
    for(i=1;i<=n;i++)
        flag[i]=0,dist[i]=cost[v][i];
    count=2;
    while(count<=n)
    {
        min=99;
        for(w=1;w<=n;w++)
            if(dist[w]<min && !flag[w])
                min=dist[w],u=w;
        flag[u]=1;
        count++;
        for(w=1;w<=n;w++)
            if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
                dist[w]=dist[u]+cost[u][w];
    }
}

```

```
int main()
{
    int n,v,i,j,cost[10][10],dist[10];
    printf("Enter the number of nodes:\n");
    scanf("%d",&n);
    printf("Enter the cost matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);
            if(cost[i][j]==0)
                cost[i][j]=infinity;
        }
    printf("Enter the source matrix:\n");
    scanf("%d",&v);
    dij(n,v,cost,dist);
    printf("Shortest path(s) from %d:\n",v);
    for(i=1;i<=n;i++)
    {
        if(i!=v)
            printf("%d->%d,cost=%d\n",v,i,dist[i]);
    }
    return 0;
}
```

Actual Input and Output:

```
"D:\AITAM\2023-2024\I SEM\  × + ▾ - □ ×
Enter the number of nodes:
4
Enter the cost matrix:
0 50 0 10
0 0 0 15
0 20 0 0
20 0 15 0
Enter the source matrix:
1
Shortest path(s) from 1:
1->2,cost=45
1->3,cost=25
1->4,cost=10

Process returned 0 (0x0)   execution time : 60.515 s
Press any key to continue.
```

EXPERIMENT-6**AIM:**

Develop a program and measure the running time for optimal Binary search trees with Dynamic Programming.

Logic / Algorithm:

Step 1: The structure of an optimal binary search tree

Construct an optimal solution to the problem from optimal solutions to sub-problems. Given keys k_i, \dots, k_j , one of these keys, say k_r ($i \leq r \leq j$).

Step 2: A recursive solution

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to sub problems. For the optimal binary search tree problem, We can define $W[i, j]$, $C[i, j]$, and $r[i, j]$ recursively as follows.

$$\begin{aligned} \{ W[i, j], C[i, j], r[i, j] \} &= \{ q_i, 0, 0 \} && \text{for } i = j \\ W[i, j] &= \{ W[i, j-1] + q_j + p_j \} && \text{for } i < j \\ C[i, j] &= \min_{i \leq k \leq j} \{ C[i, k-1] + C[k, j] \} + W[i, j] && \text{for } i < j \\ r[i, j] &= k && \text{for } i < j \end{aligned}$$

Step 3: Computing the expected search cost of an optimal binary search tree

Computing the optimal costs. we perform the third step of the dynamic-programming paradigm and compute the optimal cost by using a tabular, bottom-up approach.

Step 4: Constructing an optimal solution.

Now consider the last cell, and find the k value. The k th element will be the root. If T_{ij} is the tree and k th element will be the root then the tree is sub divided into left sub tree i.e $T_{i, k-1}$ and into right sub tree i.e $T_{k, j-1}$. The process will be repeated until T_{ij} is reached where $i=j$. At this condition the tree will become a leaf node.

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

Source Code:

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

```
int sum(int frequency[], int i, int j)
{
    int x, sum = 0;
    for (x = i; x <= j; x++)
        sum += frequency[x];
    return sum;
}
```

```
int optimalCost(int frequency[], int i, int j)
{
    int r, cost, frequencySum, min = 99999;
    if (j < i)
        return 0;
    if (j == i)
        return frequency[i];
    frequencySum = sum(frequency, i, j);
    for (r = i; r <= j; ++r)
```

```
{
    cost = optimalCost(frequency, i, r - 1) + optimalCost(frequency, r + 1, j);
    if (cost < min)
        min = cost;
}
return min + frequencySum;
}

int optimalSearchTree(int keys[], int frequency[], int n)
{
    return optimalCost(frequency, 0, n - 1);
}

int main()
{
    int n, i, keys[10], frequency[10], result;
    printf("enter number of nodes: \n");
    scanf("%d", &n);
    printf("enter keys and frequency of each node:(key, frequency) \n");
    for (i=0;i<n;i++)
        scanf("%d %d", &keys[i], &frequency[i]);
    result = optimalSearchTree(keys, frequency, n);
    printf("Cost of Optimal BST is %d", result);
    return 0;
}
```

Actual Input and Output:

```
"D:\AITAM\2023-2024\I SEM\
enter number of nodes:
3
enter keys and frequency of each node:(key, frequency)
10 34
12 8
20 50
Cost of Optimal BST is 142
Process returned 0 (0x0)   execution time : 27.459 s
Press any key to continue.
```

EXPERIMENT-7**AIM:**

Develop a program and measure the running time for identifying solution for traveling salesperson problem with Dynamic Programming

Logic / Algorithm:

Step 1: - Travelling sales person problem typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

Step 2:- A recursive solution

$$g(i, S) = \begin{cases} C[i,1] & \text{for } S = \emptyset \\ \min_{j \in S} \{ C[i,j] + g(j, S - \{j\}) \} & \text{for } S \neq \emptyset \end{cases}$$

Where $C[i,j]$ is the cost matrix of the given graph.

Step 3:- Computing the route until all the vertices are added to the set S.

Step 4:- Finally calculate $g(i, S)$ where set S contains all vertexes other than the starting vertex which gives the optimal cost of travelling.

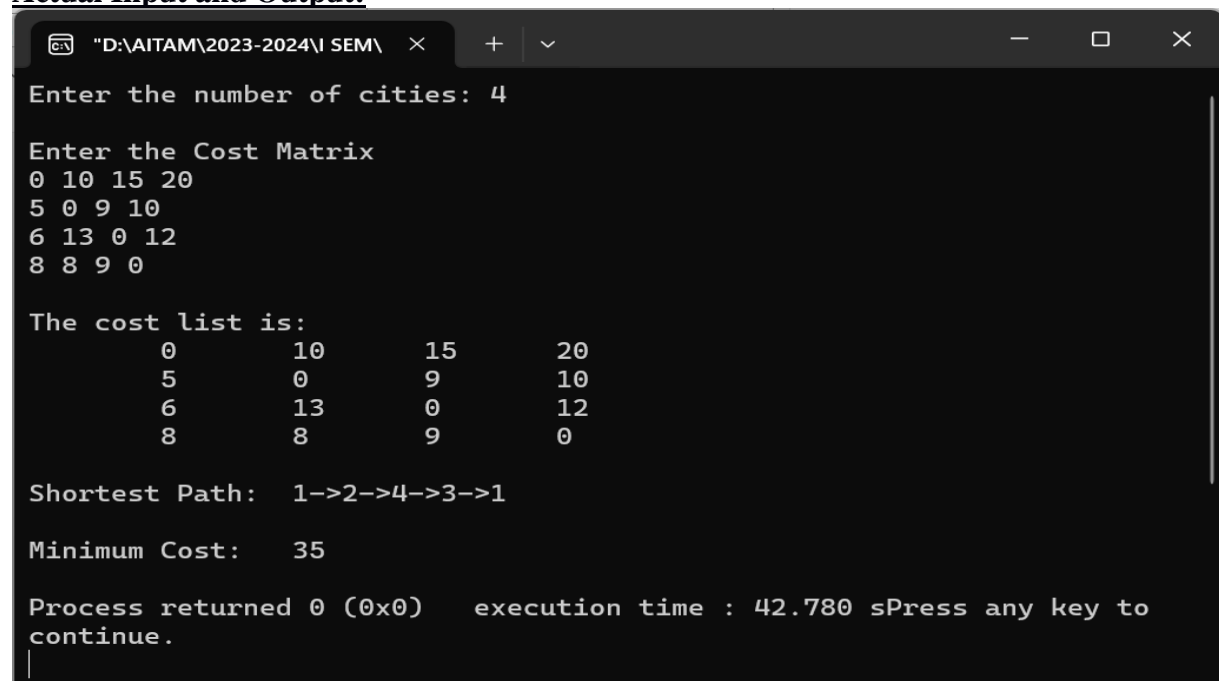
Time Complexity is $O(n^2 \cdot 2^n)$

Source Code:

```
#include <stdio.h>
int tsp_g[10][10], visited[10], n, cost = 0;
/* creating a function to generate the shortest path */
void travellingsalesman(int c)
{
    int x,k, adj_vertex = 999;
    int kmin, min = 999;
    /* marking the vertices visited in an assigned array */
    visited[c] = 1;
    /* displaying the shortest path */
    printf("%d->", c + 1);
    /* checking the minimum cost edge in the graph */
    for (k = 1; k < n; k++)
    {
        if ((tsp_g[c][k] != 0) && (visited[k] == 0))
        {
            x=(tsp_g[k][0] > tsp_g[k][c])? tsp_g[k][0] : tsp_g[k][c];
            if ((tsp_g[c][k] + x) < min)
            {
                min = tsp_g[c][k] + x;
                kmin = tsp_g[c][k];
                adj_vertex = k;
            }
        }
    }
    if (min != 999)
        cost = cost + kmin;
    if (adj_vertex == 999)
    {
        adj_vertex = 0;
    }
}
```

```
printf("%d", adj_vertex + 1);
cost = cost + tsp_g[c][adj_vertex];
return;
}
travellingsalesman(adj_vertex);
}
int main()
{
    int i, j;
    printf("Enter the number of cities: ");
    scanf("%d", & n);
    printf("\nEnter the Cost Matrix\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
            scanf("%d", & tsp_g[i][j]);
        visited[i] = 0;
    }
    printf("\nThe cost list is:");
    for (i = 0; i < n; i++)
    {
        printf("\n");
        for (j = 0; j < n; j++)
            printf("\t%d", tsp_g[i][j]);
    }
    printf("\n\nShortest Path:\t");
    travellingsalesman(0);
    printf("\n\nMinimum Cost: \t");
    printf("%d\n", cost);
    return 0;
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  x + v - □ x
Enter the number of cities: 4

Enter the Cost Matrix
0 10 15 20
5 0 9 10
6 13 0 12
8 8 9 0

The cost list is:
      0      10      15      20
      5       0       9      10
      6      13       0      12
      8       8       9       0

Shortest Path:  1->2->4->3->1

Minimum Cost:   35

Process returned 0 (0x0)   execution time : 42.780 sPress any key to
continue.
```

EXPERIMENT-8**AIM:**

Develop a program and measure the running time for identifying solution for 8-Queens problem with Backtracking.

Logic / Algorithm:

```
1  Algorithm NQueens( $k, n$ )
2  // Using backtracking, this procedure prints all
3  // possible placements of  $n$  queens on an  $n \times n$ 
4  // chessboard so that they are nonattacking.
5  {
6      for  $i := 1$  to  $n$  do
7          {
8              if Place( $k, i$ ) then
9                  {
10                      $x[k] := i$ ;
11                     if ( $k = n$ ) then write ( $x[1 : n]$ );
12                     else NQueens( $k + 1, n$ );
13                 }
14          }
15 }
```



```
1  Algorithm Place( $k, i$ )
2  // Returns true if a queen can be placed in  $k$ th row and
3  //  $i$ th column. Otherwise it returns false.  $x[ ]$  is a
4  // global array whose first ( $k - 1$ ) values have been set.
5  // Abs( $r$ ) returns the absolute value of  $r$ .
6  {
7      for  $j := 1$  to  $k - 1$  do
8          if (( $x[j] = i$ ) // Two in the same column
9              or ( $\text{Abs}(x[j] - i) = \text{Abs}(j - k)$ ))
10             // or in the same diagonal
11             then return false;
12      return true;
13 }
```

Time Complexity is $O(n!)$.

Source Code:

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

int board[20],count;
void queen(int row,int n);

int main()
{
    int n,i,j;
    printf(" - N Queens Problem Using Backtracking -");
    printf("\n\nEnter number of Queens:");
    scanf("%d",&n);
    queen(1,n);
    return 0;
}

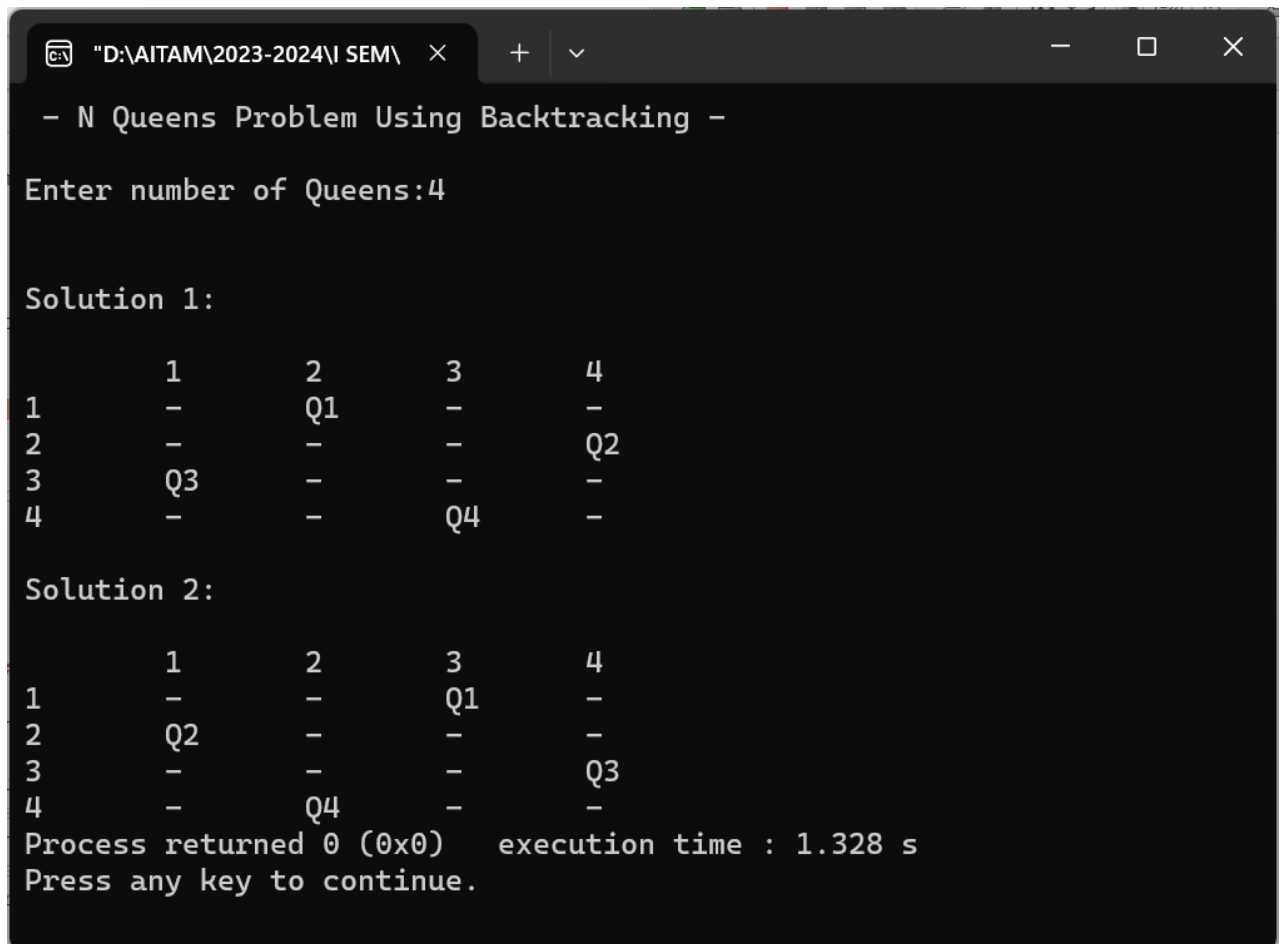
void print(int n)
{
    int i,j;
    printf("\n\nSolution %d:\n\n",++count);
    for(i=1;i<=n;++i)
        printf("\t%d",i);
    for(i=1;i<=n;++i)
    {
        printf("\n%d",i);
        for(j=1;j<=n;++j) //for nxn board
        {
            if(board[i]==j)
                printf("\tQ%d",i); //queen at i,j position
            else
                printf("\t-"); //empty slot
        }
    }
}

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; //no conflicts
}
```



```
void queen(int row,int n)
{
    int column;
    for(column=1;column<=n;++column)
    {
        if(place(row,column))
        {
            board[row]=column; //no conflicts so place queen
            if(row==n) //dead end
                print(n); //printing the board configuration
            else //try queen with next position
                queen(row+1,n);
        }
    }
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  × + ∨ - □ ×
- N Queens Problem Using Backtracking -
Enter number of Queens:4

Solution 1:

    1      2      3      4
1    -    Q1    -    -
2    -    -    -    Q2
3   Q3    -    -    -
4    -    -    Q4    -

Solution 2:

    1      2      3      4
1    -    -    Q1    -
2   Q2    -    -    -
3    -    -    -    Q3
4    -    Q4    -    -
Process returned 0 (0x0)   execution time : 1.328 s
Press any key to continue.
```

EXPERIMENT-9**AIM:**

Develop a program and measure the running time for Graph Coloring with Backtracking

Logic / Algorithm:

```

1  Algorithm mColoring( $k$ )
2  // This algorithm was formed using the recursive backtracking
3  // schema. The graph is represented by its boolean adjacency
4  // matrix  $G[1 : n, 1 : n]$ . All assignments of  $1, 2, \dots, m$  to the
5  // vertices of the graph such that adjacent vertices are
6  // assigned distinct integers are printed.  $k$  is the index
7  // of the next vertex to color.
8  {
9      repeat
10     { // Generate all legal assignments for  $x[k]$ .
11         NextValue( $k$ ); // Assign to  $x[k]$  a legal color.
12         if ( $x[k] = 0$ ) then return; // No new color possible
13         if ( $k = n$ ) then // At most  $m$  colors have been
14             // used to color the  $n$  vertices.
15             write ( $x[1 : n]$ );
16             else mColoring( $k + 1$ );
17     } until (false);
18 }

1  Algorithm NextValue( $k$ )
2  //  $x[1], \dots, x[k - 1]$  have been assigned integer values in
3  // the range  $[1, m]$  such that adjacent vertices have distinct
4  // integers. A value for  $x[k]$  is determined in the range
5  //  $[0, m]$ .  $x[k]$  is assigned the next highest numbered color
6  // while maintaining distinctness from the adjacent vertices
7  // of vertex  $k$ . If no such color exists, then  $x[k]$  is 0.
8  {
9      repeat
10     {
11          $x[k] := (x[k] + 1) \bmod (m + 1)$ ; // Next highest color.
12         if ( $x[k] = 0$ ) then return; // All colors have been used.
13         for  $j := 1$  to  $n$  do
14         { // Check if this color is
15             // distinct from adjacent colors.
16             if ( $(G[k, j] \neq 0) \text{ and } (x[k] = x[j])$ )
17                 // If  $(k, j)$  is an edge and if adj.
18                 // vertices have the same color.
19                 then break;
20         }
21         if ( $j = n + 1$ ) then return; // New color found
22     } until (false); // Otherwise try to find another color.
23 }

```

Time Complexity is $O(n.m^n)$, where n is number of vertices and m is number of colors.

Source Code:

```
#include<stdio.h>

int m, n, c=0, count=0;
int g[50][50], x[50];

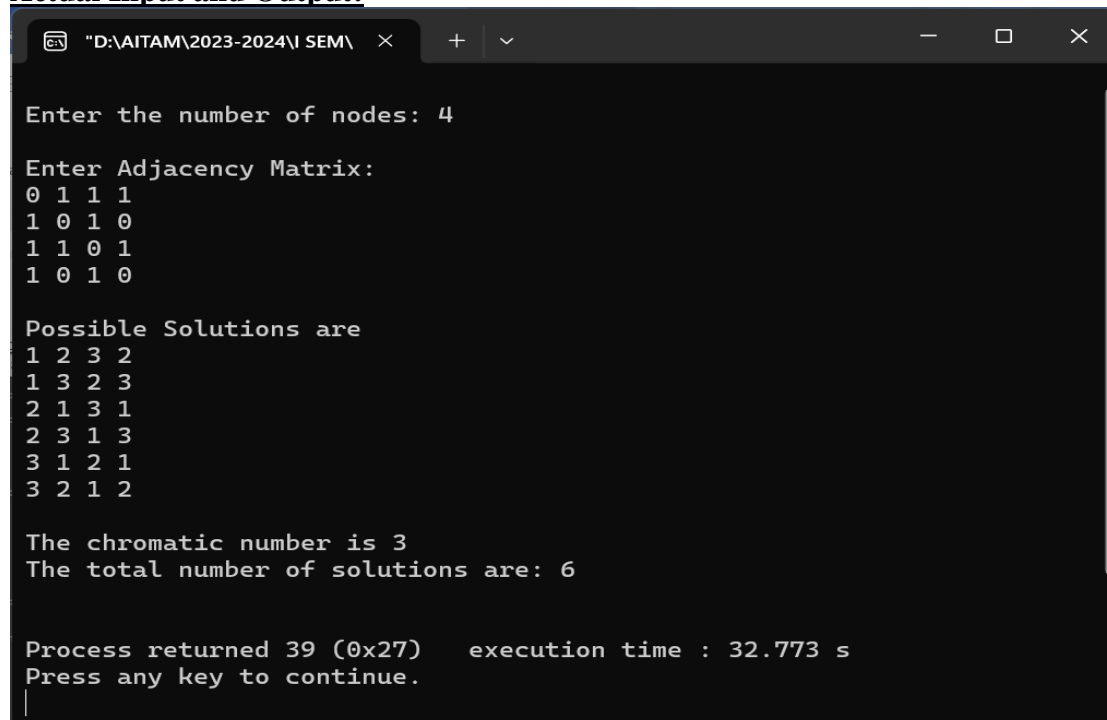
void nextValue(int k);
void GraphColoring(int k);

void main()
{
    int i, j;
    int temp;
    printf("\nEnter the number of nodes: ");
    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("\nPossible Solutions are\n");
    for(m=1;m<=n;m++)
    {
        if(c==1)
        {
            break;
        }
        GraphColoring(1);
    }
    printf("\nThe chromatic number is %d", m-1);
    printf("\nThe total number of solutions are: %d\n\n", count);
}

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);
}
}
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;
        }
    }
}
```

Actual Input and Output:



```
"D:\AITAM\2023-2024\I SEM\  × + ▾

Enter the number of nodes: 4

Enter Adjacency Matrix:
0 1 1 1
1 0 1 0
1 1 0 1
1 0 1 0

Possible Solutions are
1 2 3 2
1 3 2 3
2 1 3 1
2 3 1 3
3 1 2 1
3 2 1 2

The chromatic number is 3
The total number of solutions are: 6

Process returned 39 (0x27)   execution time : 32.773 s
Press any key to continue.
|
```

EXPERIMENT-10**AIM:**

Develop a program and measure the running time to generate solution of Hamiltonian Cycle problem with Backtracking.

Logic / Algorithm:

```
1  Algorithm Hamiltonian( $k$ )
2  // This algorithm uses the recursive formulation of
3  // backtracking to find all the Hamiltonian cycles
4  // of a graph. The graph is stored as an adjacency
5  // matrix  $G[1 : n, 1 : n]$ . All cycles begin at node 1.
6  {
7      repeat
8      { // Generate values for  $x[k]$ .
9          NextValue( $k$ ); // Assign a legal next value to  $x[k]$ .
10         if ( $x[k] = 0$ ) then return;
11         if ( $k = n$ ) then write ( $x[1 : n]$ );
12         else Hamiltonian( $k + 1$ );
13     } until (false);
14 }
```

```
1  Algorithm NextValue( $k$ )
2  //  $x[1 : k - 1]$  is a path of  $k - 1$  distinct vertices. If  $x[k] = 0$ , then
3  // no vertex has as yet been assigned to  $x[k]$ . After execution,
4  //  $x[k]$  is assigned to the next highest numbered vertex which
5  // does not already appear in  $x[1 : k - 1]$  and is connected by
6  // an edge to  $x[k - 1]$ . Otherwise  $x[k] = 0$ . If  $k = n$ , then
7  // in addition  $x[k]$  is connected to  $x[1]$ .
8  {
9      repeat
10     {
11          $x[k] := (x[k] + 1) \bmod (n + 1)$ ; // Next vertex.
12         if ( $x[k] = 0$ ) then return;
13         if ( $G[x[k - 1], x[k]] \neq 0$ ) then
14         { // Is there an edge?
15             for  $j := 1$  to  $k - 1$  do if ( $x[j] = x[k]$ ) then break;
16             // Check for distinctness.
17             if ( $j = k$ ) then // If true, then the vertex is distinct.
18                 if ( $(k < n)$  or ( $(k = n)$  and  $G[x[n], x[1]] \neq 0$ ))
19                     then return;
20         }
21     } until (false);
22 }
```

Time Complexity is $O(n!)$.

Source Code:

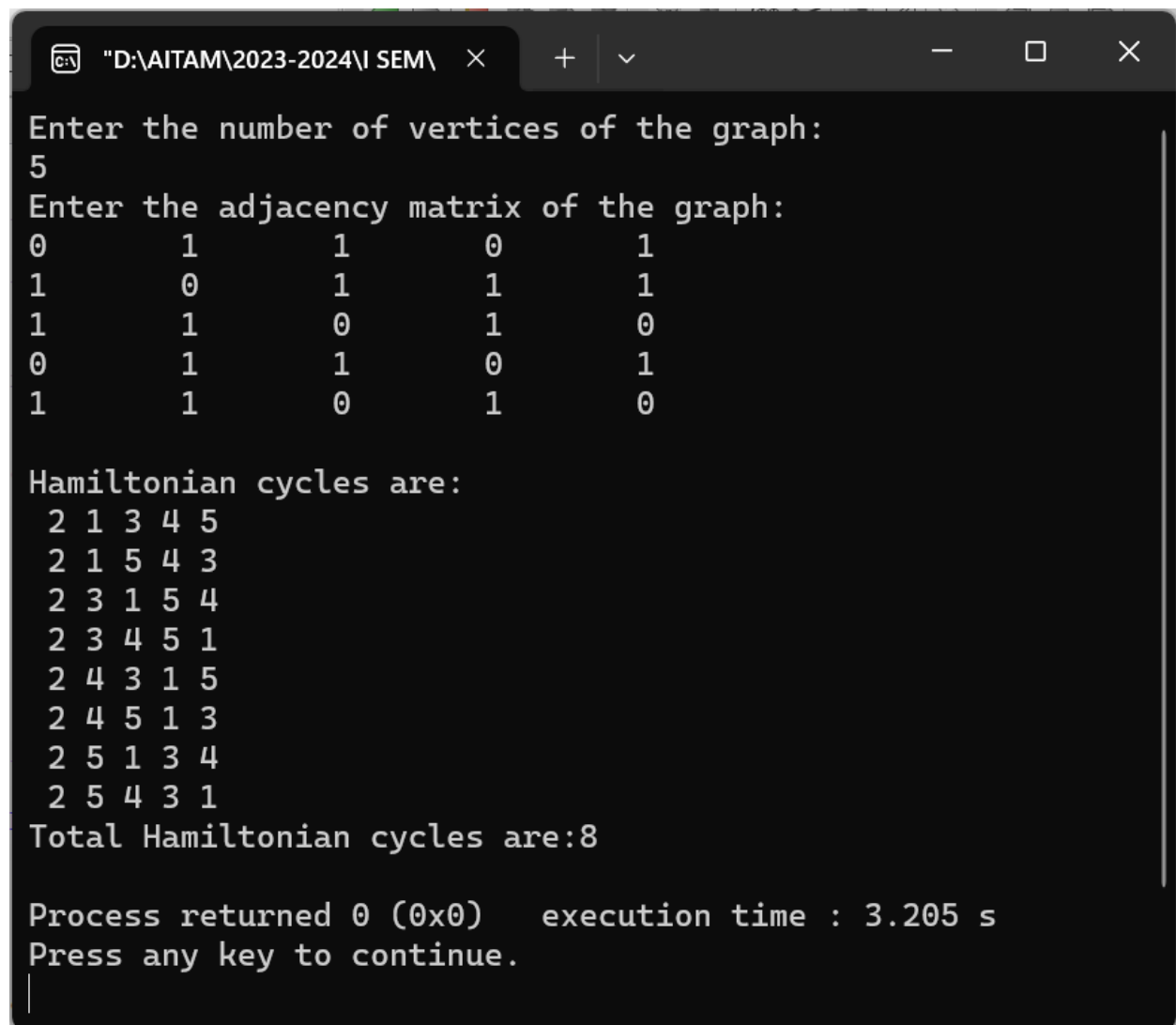
```
#include<stdio.h>
int G[10][10];
int x[10]={-1},n;
int count=0;

void HCycle(int k)
{
    int i;
    while(1)
    {
        Next(k);//generates next legal vertex for determining cycle
        if(x[k]==0)
            return;
        if(k==n)
        {
            count=count+1;
            printf("\n");
            for(i=1;i<=n;i++)
                printf(" %d",x[i]); //Hamiltonian cycle
            //printf(" %d",x[1]);
        }
        else
            HCycle(k+1);
    }
}

void Next(int k)
{
    int j;
    while(1)
    {
        x[k]=(x[k]+1)%(n+1); // next vertex
        if(x[k]==0)
            return;
        if(G[x[k-1]][x[k]]!=0) //if there is an edge between k and 'k-1'
        {
            for(j=1;j<=k-1;j++)//every adjacent vertex
            {
                if(x[j]==x[k])//not a distinct vertex
                    break;
            }
            if(j==k) //obtain a distinct vertex
            {
                if((k<n)||((k==n)&&(G[x[n]][x[1]]!=0)))
                    return;//return a distinct vertex
            }
        }
    }
}
```

```
int main()
{
    int i,j;
    printf("Enter the number of vertices of the graph:\n");
    scanf("%d",&n);
    printf("Enter the adjacency matrix of the graph:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&G[i][j]);
    printf("\nHamiltonian cycles are:");
    HCycle(1);
    printf("\nTotal Hamiltonian cycles are:%d\n",count);
    return 0;
}
```

Actual Input and Output:



```
D:\AITAM\2023-2024\I SEM\
Enter the number of vertices of the graph:
5
Enter the adjacency matrix of the graph:
0      1      1      0      1
1      0      1      1      1
1      1      0      1      0
0      1      1      0      1
1      1      0      1      0

Hamiltonian cycles are:
2 1 3 4 5
2 1 5 4 3
2 3 1 5 4
2 3 4 5 1
2 4 3 1 5
2 4 5 1 3
2 5 1 3 4
2 5 4 3 1
Total Hamiltonian cycles are:8

Process returned 0 (0x0)    execution time : 3.205 s
Press any key to continue.
```

EXPERIMENT-11**AIM:**

Develop a program and measure the running time running time to generate solution of Knapsack problem with Backtracking.

Logic / Algorithm:

```

1  Algorithm BKnap( $k, cp, cw$ )
2  //  $m$  is the size of the knapsack;  $n$  is the number of weights
3  // and profits.  $w[ ]$  and  $p[ ]$  are the weights and profits.
4  //  $p[i]/w[i] \geq p[i+1]/w[i+1]$ .  $fw$  is the final weight of
5  // knapsack;  $fp$  is the final maximum profit.  $x[k] = 0$  if  $w[k]$ 
6  // is not in the knapsack; else  $x[k] = 1$ .
7  {
8      // Generate left child.
9      if ( $cw + w[k] \leq m$ ) then
10     {
11          $y[k] := 1$ ;
12         if ( $k < n$ ) then BKnap( $k + 1, cp + p[k], cw + w[k]$ );
13         if ( $(cp + p[k] > fp)$  and ( $k = n$ )) then
14         {
15              $fp := cp + p[k]$ ;  $fw := cw + w[k]$ ;
16             for  $j := 1$  to  $k$  do  $x[j] := y[j]$ ;
17         }
18     }
19     // Generate right child.
20     if ( $\text{Bound}(cp, cw, k) \geq fp$ ) then
21     {
22          $y[k] := 0$ ; if ( $k < n$ ) then BKnap( $k + 1, cp, cw$ );
23         if ( $(cp > fp)$  and ( $k = n$ )) then
24         {
25              $fp := cp$ ;  $fw := cw$ ;
26             for  $j := 1$  to  $k$  do  $x[j] := y[j]$ ;
27         }
28     }
29 }
```

```

1  Algorithm Bound( $cp, cw, k$ )
2  //  $cp$  is the current profit total,  $cw$  is the current
3  // weight total;  $k$  is the index of the last removed
4  // item; and  $m$  is the knapsack size.
5  {
6       $b := cp$ ;  $c := cw$ ;
7      for  $i := k + 1$  to  $n$  do
8      {
9           $c := c + w[i]$ ;
9          if ( $c < m$ ) then  $b := b + p[i]$ ;
10         else return  $b + (1 - (c - m)/w[i]) * p[i]$ ;
11     }
12     return  $b$ ;
13 }
```

Time Complexity is $O(2^n)$.

Source Code:

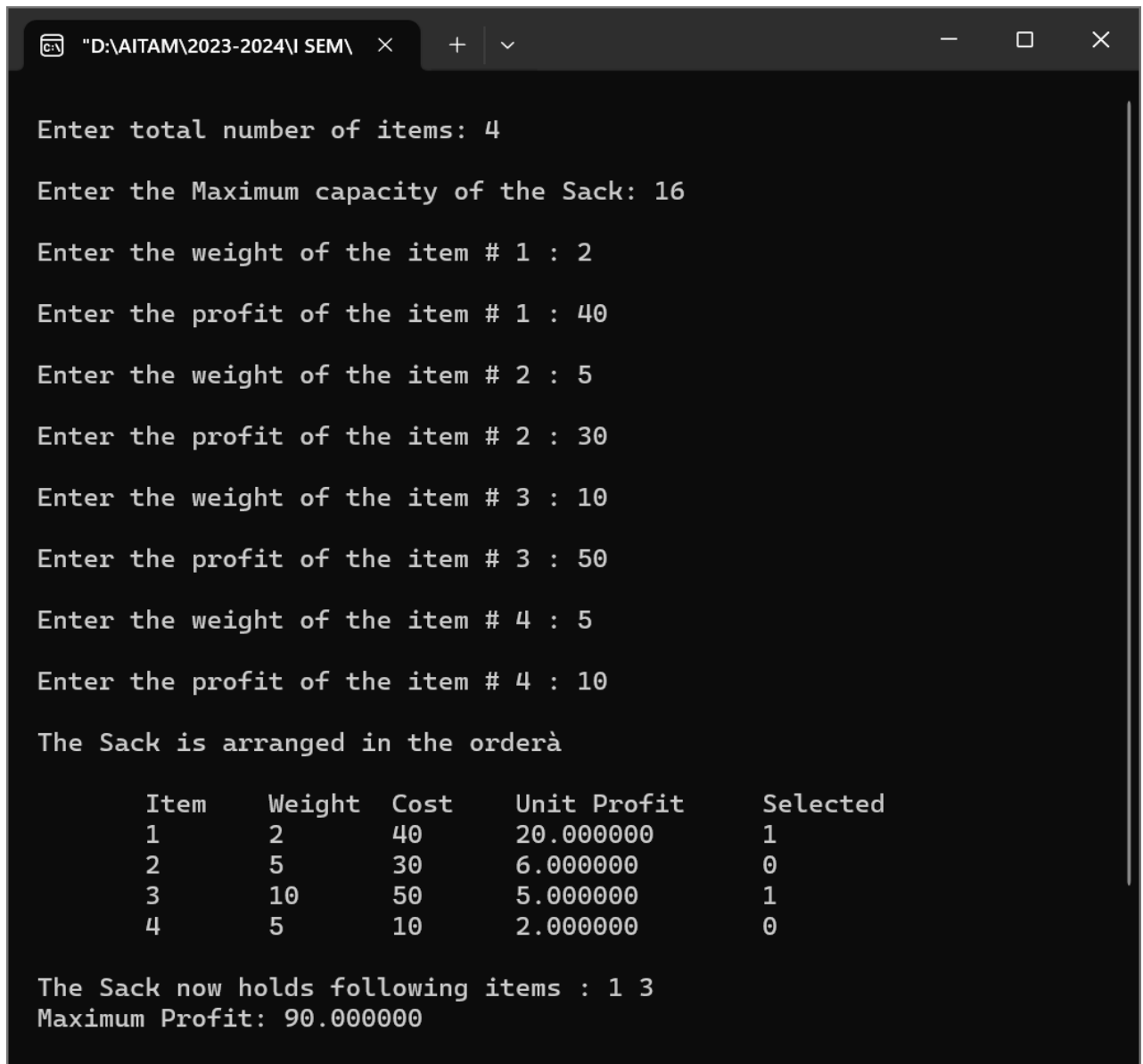
```
#include <stdio.h>
#define max 10

int w[max],i,j,p[max];
int n,m;
float unit[max];
int y[max],x[max],fp=-1,fw;
void get()
{
printf("\n Enter total number of items: ");
scanf("%d",&n);
printf("\n Enter the Maximum capacity of the Sack: ");
scanf("%d",&m);
for(i=0;i<n;i++)
{
printf("\n Enter the weight of the item # %d : ",i+1);
scanf("%d",&w[i]);
printf("\n Enter the profit of the item # %d : ", i+1);
scanf("%d", &p[i]);
}
}
void show()
{
float s=0.0;
printf("\n\tItem\tWeight\tCost\tUnit Profit\tSelected ");
for(i=0;i<n;i++)
printf("\n\t%d\t%d\t%d\t%f\t%d",i+1,w[i],p[i],unit[i],x[i]);
printf("\n\n The Sack now holds following items : ");
for(i=0;i<n;i++)
if(x[i]==1)
{
printf("%d\t",i+1);
s += (float) p[i] * (float) x[i];
}
printf("\n Maximum Profit: %f \n\n",s);
}
/*Arrange the item based on high profit per Unit*/
void sort()
{
int t,t1;
float t2;
for(i=0;i<n;i++)
unit[i] = (float) p[i] / (float) w[i];
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(unit[i] < unit[j])
{
```

```
t2 = unit[i];
unit[i] = unit[j];
unit[j] = t2;
t = p[i];
p[i] = p[j];
p[j] = t;
t1 = w[i];
w[i] = w[j];
w[j] = t1;
}
}
}
}
float bound(float cp,float cw,int k)
{
float b = cp;
float c = cw;
for(i=k;i<=n;i++)
{
c = c+w[i];
if( c < m)
b = b +p[i];
else
return (b+(1-(c-m)/ (float)w[i])*p[i]);
}
return b;
}
void knapsack(int k,float cp,float cw)
{
if(cw+w[k] <= m)
{
y[k] = 1;
if(k <= n)
knapsack(k+1,cp+p[k],cw+w[k]);
if(((cp+p[k]) > fp) && ( k == n))
{
fp = cp+p[k];
fw = cw+w[k];
for(j=0;j<=k;j++)
x[j] = y[j];
}
}
if(bound(cp,cw,k) >= fp)
{
y[k] = 0;
if( k <= n)
knapsack(k+1,cp,cw);
if((cp > fp) && (k == n))
{
fp = cp;
fw = cw;
```

```
for(j=0;j<=k;j++)
x[j] = y[j];
}
}
}
int main()
{
get();
printf("\n The Sack is arranged in the order...\n");
sort();
knapsack(0,0.0,0.0);
show();
return 0;
}
```

Actual Input and Output:



```
Enter total number of items: 4
Enter the Maximum capacity of the Sack: 16
Enter the weight of the item # 1 : 2
Enter the profit of the item # 1 : 40
Enter the weight of the item # 2 : 5
Enter the profit of the item # 2 : 30
Enter the weight of the item # 3 : 10
Enter the profit of the item # 3 : 50
Enter the weight of the item # 4 : 5
Enter the profit of the item # 4 : 10
The Sack is arranged in the orderà
    Item    Weight    Cost    Unit Profit    Selected
    1        2        40    20.000000        1
    2        5        30     6.000000        0
    3       10        50     5.000000        1
    4        5        10     2.000000        0
The Sack now holds following items : 1 3
Maximum Profit: 90.000000
```