

**DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY****(Common to CSE & ISE)****Subject Code: 10CSL47****I.A. Marks : 25**

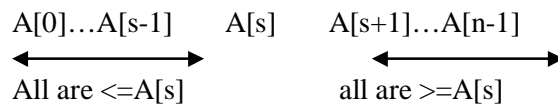
Design, develop and implement the specified algorithms for the following problems using C/C++ Language in LINUX / Windows environment.

1. Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of  $n$ , the number of elements in the list to be sorted and plot a graph of the time taken versus  $n$ . The elements can be read from a file or can be generated using the random number generator.
2. Using Open, implement a parallelized Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of  $n$ , the number of elements in the list to be sorted and plot a graph of the time taken versus  $n$ . The elements can be read from a file or can be generated using the random number generator.
3. a. Obtain the Topological ordering of vertices in a given digraph.  
b. Compute the transitive closure of a given directed graph using Warshall's algorithm.
4. Implement 0/1 Knapsack problem using Dynamic Programming.
5. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.
6. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.
7. a. Print all the nodes reachable from a given starting node in a digraph using BFS method.  
b. Check whether a given graph is connected or not using DFS method.
8. Find a subset of a given set  $S = \{s_1, s_2, \dots, s_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ . For example, if  $S = \{1, 2, 5, 6, 8\}$  and  $d = 9$  there are two solutions  $\{1, 2, 6\}$  and  $\{1, 8\}$ . A suitable message is to be displayed if the given problem instance doesn't have a solution.
9. Implement any scheme to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.
10. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.
11. Implement All-Pairs Shortest Paths Problem using Floyd's algorithm. Parallelize this algorithm, implement it using Open and determine the speed-up achieved.
12. Implement N Queen's problem using Back Tracking.

1. Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of  $n$ , the number of elements in the list to be sorted and plot a graph of the time taken versus  $n$ . The elements can be read from a file or can be generated using the random number generator.

### Quick Sort Method:

Quick Sort divides the array according to the value of elements. It rearranges elements of a given array  $A[0..n-1]$  to achieve its partition, where the elements before position  $s$  are smaller than or equal to  $A[s]$  and all the elements after position  $s$  are greater than or equal to  $A[s]$ .



#### **Algorithm : QUICKSORT(a[l..r])**

//Sorts a subarray by quicksort

//Input: A subarray  $A[l..r]$  of  $A[0..n-1]$ , defined by its left and right indices  $l$  and  $r$

//Output: Subarray  $A[l..r]$  sorted in nondecreasing order

```
{
    if l < r
    {
        s ← Partition(A[l..r]) //s is a split position
        QUICKSORT(A[l..s-1])
        QUICKSORT(A[s+1..r])
    }
}
```

#### **Algorithm : Partition(A[l..r])**

//Partition a subarray by using its first element as its pivot

//Input: A subarray  $A[l..r]$  of  $A[0..n-1]$ , defined by its left and right indices  $l$  and  $r$  ( $l < r$ )

//Output: A partition of  $A[l..r]$ , with the split position returned as this function's value

```
{
    p ← A[l]
    i ← l; j ← r+1
    repeat
    {
        repeat i ← i+1 until A[i] >= p
        repeat j ← j-1 until A[j] <= p
        swap(A[i], A[j])
    } until i >= j
    swap(A[l], A[j]) // undo last swap when i >= j
    swap(A[l], A[j])
    return j
}
```

**Complexity:**  $C_{\text{best}}(n) = 2 C_{\text{best}}(n/2) + n$  for  $n > 1$   $C_{\text{best}}(1) = 0$   
 $C_{\text{worst}}(n) \in \theta(n^2)$   
 $C_{\text{avg}}(n) \approx 1.38n \log_2 n$

**Program:**

```

/*To sort a set of elements using Quick sort algorithm.*/

#include<stdio.h>
#include<time.h>
#define max 500

void qsort(int [],int,int);
int partition(int [],int,int);

void main()
{
    int a[max],i,n;
    clock_t s,e;
    clrscr();

    printf("Enter the value of n:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
        a[i]=rand()%100;

    printf("\nThe array elements before\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);

    s=clock();
    delay(100);
    qsort(a,0,n-1);
    e=clock();

    printf("\nElements of the array after sorting are:\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    printf("\nTime taken:%f",(e-s)/CLK_TCK);
    getch();
}

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

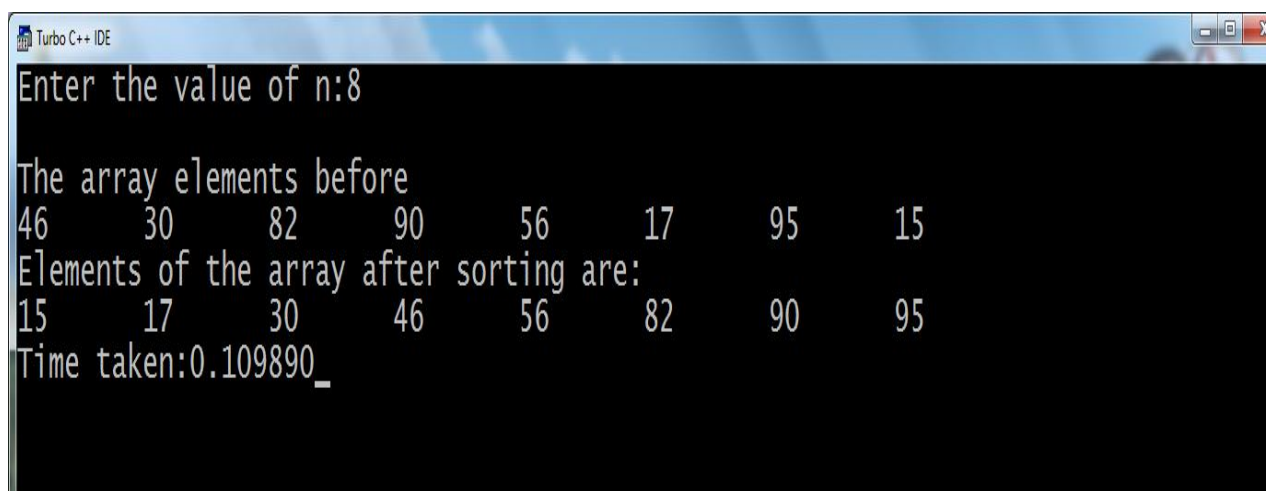
int partition(int a[], int low,int high)
{
    int pivot,i,j,temp;

    pivot=a[low];
    i=low+1;

```

```
j=high;
while(1)
{
    while(pivot>a[i] && i<=high)
        i++;
    while(pivot<a[j])
        j--;

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

**OUTPUT:**

The screenshot shows the Turbo C++ IDE window with the following output:

```
Enter the value of n:8

The array elements before
46    30    82    90    56    17    95    15
Elements of the array after sorting are:
15    17    30    46    56    82    90    95
Time taken:0.109890_
```

**Note:**

- Program to be executed for various sizes of input.
- Table to be filled with data

Size: n	Ascending		Descending		Random Order	
	Input	Time taken	Input	Time taken	Input	Time taken
4						
8						
16						
32						

2. Using OpenMP, implement a parallelized Merge Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of  $n$ , the number of elements in the list to be sorted and plot a graph of the time taken versus  $n$ . The elements can be read from a file or can be generated using the random number generator.

### Merge Sort Algorithm:

Merge sort is a perfect example of a successful application of the divide-and-conquer technique.

1. Split array  $A[1..n]$  in two and make copies of each half in arrays  $B[1.. n/2]$  and  $C[1.. n/2]$
2. Sort arrays B and C
3. Merge sorted arrays B and C into array A as follows:
  - a) Repeat the following until no elements remain in one of the arrays:
    - i. compare the first elements in the remaining unprocessed portions of the arrays
    - ii. copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - b) Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

#### **Algorithm:** MergeSort (A [0...n-1])

//This algorithm sorts array A [0...n-1] by recursive mergesort.

//Input: An array A [0...n-1] of orderable elements.

//Output: Array A [0...n-1] sorted in non-decreasing order

```
{
    if n>1
    {
        Copy A [0... ⌊n/2⌋-1] to B [0... ⌊n/2⌋-1]
        Copy A [⌊n/2⌋ ... n-1] to C [0... ⌈n/2⌉-1]
        MergeSort (B [0... ⌊n/2⌋-1])
        MergeSort (C [0... ⌈n/2⌉-1])
        Merge (B, C, A)
    }
}
```

#### **Algorithm:** Merge (B [0...p-1], C [0...q-1], A [0...p+q-1])

//Merges two sorted arrays into one sorted array.

//Input: Arrays B [0...p-1] and C [0...q-1] both sorted.

//Output: Sorted array A [0...p+q-1] of the elements of B and C.

```
{
    i ← 0; j ← 0; k ← 0
    while i<p and j<q do
    {
        if B[i] ≤ C[j]
            A[k] ← B[i]; i ← i+1
        else
            A[k] ← C[j]; j ← j+1
        k ← k+1
    }
    if i=p
        copy C [j...q-1] to A[k...p+q-1]
    else
        copy B [i...p-1] to A[k...p+q-1]
}
```

**Complexity:**

- All cases have same efficiency:  $\Theta(n \log n)$
- Number of comparisons is close to theoretical minimum for comparison-based sorting:  
 $\log n! \approx n \lg n - 1.44 n$
- Space requirement:  $\Theta(n)$  (NOT in-place)

**Program:**

/\*To sort a set of elements using Merge sort algorithm.\*/

```
#include "stdafx.h"
#include<omp.h>
#include<time.h>
#include<stdio.h>
#include<iostream>
#include<conio.h>
#define max 100

void mergesort(int[100],int,int);
void merge(int[100],int,int,int);
int a[max];

void main()
{
    int i,n;
    clock_t s,e;

    printf("Enter the no.of elements\n");
    scanf_s("%d",&n);
    printf("Elements of the array before sorting\n");

    for(i=0;i<n;i++)
    {
        a[i]=rand()%1000;
        printf("%d\t",a[i]);
    }

    s=clock();
    mergesort(a,0,n-1);
    e=clock();

    printf("\nElements of the array after sorting\n");
#pragma omp parallel for
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);

    printf("\nthe time taken=%f\n",(e-s)/CLK_TCK);
    getch();
}
```

```

void mergesort(int a[100],int low,int high)
{
    int mid;

    if(high>low)
    {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,mid,high);
    }
}

void merge(int a[100],int low,int mid,int high)
{
    int h=low,j=mid+1,i=low,b[max],k;

#pragma omp parallel for
    while((h<=mid)&&(j<=high))
    {
        if(a[h]<=a[j])
        {
            b[i]=a[h];
            h=h+1;
        }
        else
        {
            b[i]=a[j];
            j=j+1;
        }
        i=i+1;
    }

    if(h>mid)
    {
#pragma omp parallel for
        for(k=j;k<=high;k++)
        {
            b[i]=a[k];
            i++;
        }
    }
    else
    {
#pragma omp parallel for
        for(k=h;k<=mid;k++)
        {
            b[i]=a[k];
            i++;
        }
    }

#pragma omp parallel for

```



```

    for(k=low;k<=high;k++)
        a[k]=b[k];
}

```

## OUTPUT

```

c:\users\admin\documents\visual studio 2010\Projects\merge\Debug\merge.exe
Enter the no.of elements
10
Elements of the array before sorting
41    467    334    500    169    724    478    358    962    464
Elements of the array after sorting
41    169    334    358    464    467    478    500    724    962
the time taken=0.000000

```

### Note:

- Program to be executed for various sizes of input.
- Fill the given table. Obtaining a constant value in the column “time taken” would prove that the complexity of merge sort is same in all case.

Size: n	Ascending		Descending		Random Order	
	Input	Time taken	Input	Time taken	Input	Time taken
4						
8						
16						
32						

**3a. Obtain the Topological ordering of vertices in a given digraph.****Topological Ordering:**

This method is based on decrease and conquer technique a vertex with no incoming edges is selected and deleted along with the outgoing edges. If there are several vertices with no incoming edges arbitrarily a vertex is selected. The order in which the vertices are visited and deleted one by one results in topological sorting.

- 1) Find the vertices whose indegree is zero and place them on the stack
- 2) Pop a vertex u and it is the task to be done
- 3) Add the vertex u to the solution vector
- 4) Find the vertices v adjacent to vertex u. The vertices v represents the jobs which depend on job u
- 5) Decrement indegree[v] gives the number of depended jobs of v are reduced by one.

**Algorithm topological\_sort(a,n,T)**

//purpose :To obtain the sequence of jobs to be executed resut

In topological order

// Input:a-adjacency matrix of the given graph

//n-the number of vertices in the graph

//output:

// T-indicates the jobs that are to be executed in the order

```

    For j<-0 to n-1 do
        Sum<-0
        For i<- 0to n-1 do
            Sum<-sum+a[i][j]
        End for
        Top ← -1
        For i<- 0 to n-1 do
            If(indegree [i]=0)
                Top <-top+1
                S[top]<- i
            End if
        End for
        While top!= 1
            u<-s[top]
            top<-top-1
            Add u to solution vector T
            For each vertex v adjacent to u
                Decrement indegree [v] by one
                If(indegree [v]=0)
                    Top<-top+1
                    S[top]<-v
                End if
            End for
        End while
        Write T
    return

```

**Complexity:** Complexity of topological sort is given by  $O[V*V]$ .

**Program:**

```
/*To obtain the topological order in of vertices in a digraph.*/
```

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

```
void findindegree(int [10][10],int[10],int);
```

```
void topological(int,int [10][10]);
```

```
void main()
```

```
{
    int a[10][10],i,j,n;
    clrscr();

    printf("Enter the number of nodes:");
    scanf("%d",&n);

    printf("\nEnter the adjacency matrix\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);

    printf("\nThe adjacency matrix is:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("%d\t",a[i][j]);
        }
        printf("\n");
    }
    topological(n,a);
    getch();
}
```

```
void findindegree(int a[10][10],int indegree[10],int n)
```

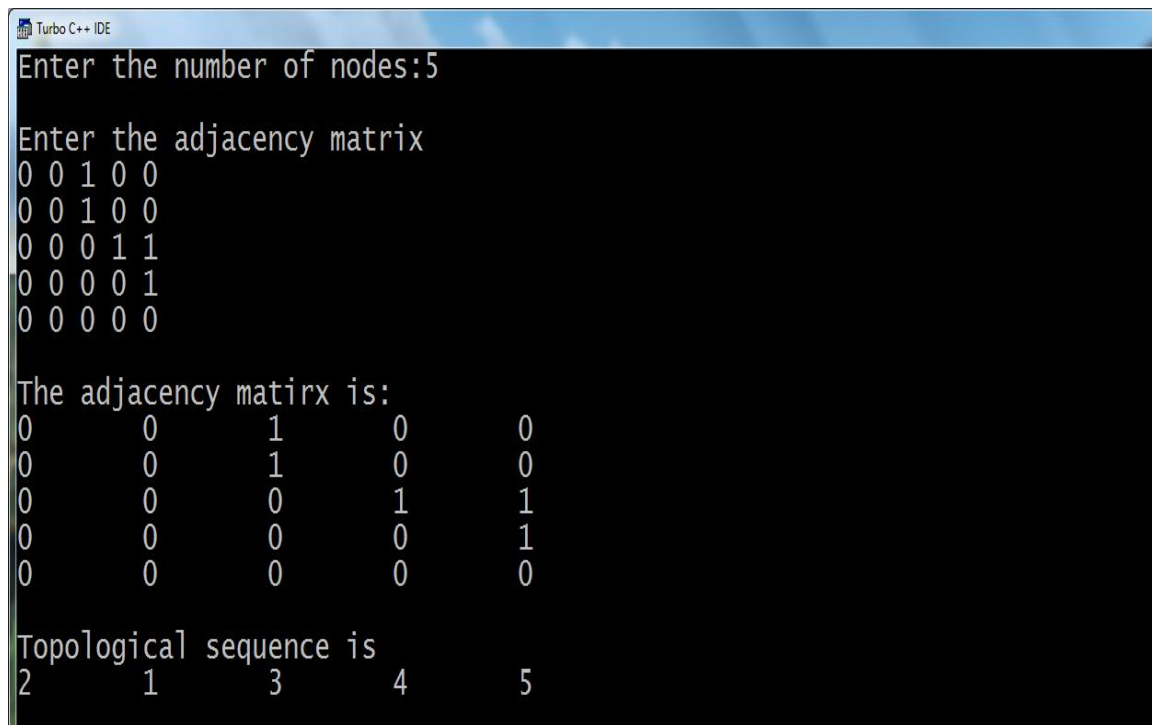
```
{
    int i,j,sum;
    for(j=1;j<=n;j++)
    {
        sum=0;
        for(i=1;i<=n;i++)
        {
            sum=sum+a[i][j];
        }
        indegree[j]=sum;
    }
}
```

```
void topological(int n,int a[10][10])
```

```
{
    int k,top,t[100],i,stack[20],u,v,indegree[20];
```

```
k=1;
top=-1;
findindegree(a,indegree,n);
for(i=1;i<=n;i++)
{
    if(indegree[i]==0)
    {
        stack[++top]=i;
    }
}
while(top!=-1)
{
    u=stack[top--];
    t[k++]=u;
    for(v=1;v<=n;v++)
    {
        if(a[u][v]==1)
        {
            indegree[v]--;
            if(indegree[v]==0)
            {
                stack[++top]=v;
            }
        }
    }
}

printf("\nTopological sequence is\n");
for(i=1;i<=n;i++)
    printf("%d\t",t[i]);
}
```

**OUTPUT:**

```
Turbo C++ IDE
Enter the number of nodes:5
Enter the adjacency matrix
0 0 1 0 0
0 0 1 0 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0

The adjacency matrix is:
0      0      1      0      0
0      0      1      0      0
0      0      0      1      1
0      0      0      0      1
0      0      0      0      0

Topological sequence is
2      1      3      4      5
```

### 3b. Compute the transitive closure of a given directed graph using Warshall's algorithm.

#### Warshall's algorithm:

The transitive closure of a directed graph with  $n$  vertices can be defined as the  $n$ -by- $n$  boolean matrix  $T=\{t_{ij}\}$ , in which the element in the  $i$ th row ( $1 \leq i \leq n$ ) and  $j$ th column ( $1 \leq j \leq n$ ) is 1 if there exists a non trivial directed path from  $i$ th vertex to  $j$ th vertex, otherwise,  $t_{ij}$  is 0.

Warshall's algorithm constructs the transitive closure of a given digraph with  $n$  vertices through a series of  $n$ -by- $n$  boolean matrices:  $R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}$  where,  $R^{(0)}$  is the adjacency matrix of digraph and  $R^{(1)}$  contains the information about paths that use the first vertex as intermediate. In general, each subsequent matrix in series has one more vertex to use as intermediate for its path than its predecessor. The last matrix in the series  $R^{(n)}$  reflects paths that can use all  $n$  vertices of the digraph as intermediate and finally transitive closure is obtained. The central point of the algorithm is that we compute all the elements of each matrix  $R^{(k)}$  from its immediate predecessor  $R^{(k-1)}$  in series.

#### **Algorithm** Warshall( $A[1..n, 1..n]$ )

//Implements Warshall's algorithm for computing the transitive closure

//Input: The Adjacency matrix  $A$  of a digraph with  $n$  vertices

//Output: The transitive closure of digraph

```
{
     $R^{(0)} \leftarrow A$ 
    for  $k \leftarrow 1$  to  $n$  do
    {
        for  $i \leftarrow 1$  to  $n$  do
        {
            for  $j \leftarrow 1$  to  $n$  do
            {
                 $R^{(k)}[i,j] \leftarrow R^{(k-1)}[i,j] \text{ or } R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j]$ 
            }
        }
    }
    return  $R^{(n)}$ 
}
```

**Complexity:** The time efficiency of Warshall's algorithm is in  $\Theta(n^3)$ .

**Program:**

```
/*Compute the transitive closure of a given directed graph using Warshall's algorithm.*/
```

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

```
void warshall(int[10][10],int);
```

```
void main()
```

```
{
    int a[10][10],i,j,n;
    clrscr();

    printf("Enter the number of nodes:");
    scanf("%d",&n);

    printf("\nEnter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);

    printf("The adjacency matrix is:\n");
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("%d\t",a[i][j]);
        }
        printf("\n");
    }

    warshall(a,n);
    getch();
}
```

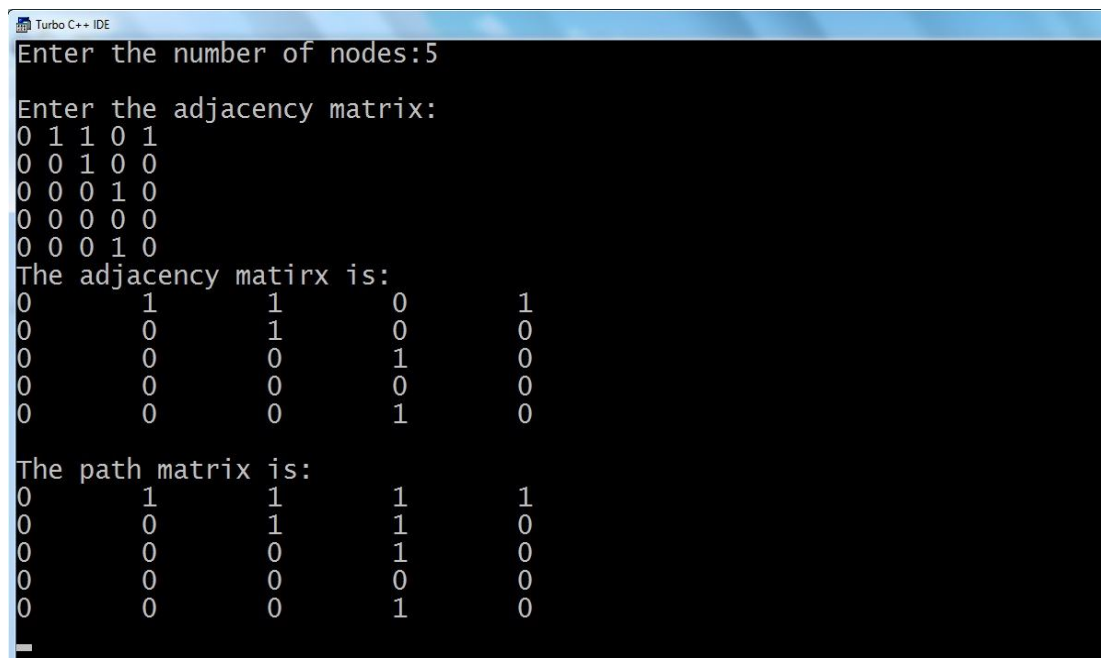
```
void warshall(int p[10][10],int n)
```

```
{
    int i,j,k;

    for(k=1;k<=n;k++)
    {
        for(j=1;j<=n;j++)
        {
            for(i=1;i<=n;i++)
            {
                if((p[i][j]==0) && (p[i][k]==1) && (p[k][j]==1))
                {
                    p[i][j]=1;
                }
            }
        }
    }

    printf("\nThe path matrix is:\n");
}
```

```
for(i=1;i<=n;i++)
{
    for(j=1;j<=n;j++)
    {
        printf("%d\t",p[i][j]);
    }
    printf("\n");
}
```

**OUTPUT:**

The screenshot shows the Turbo C++ IDE with the following output:

```
Enter the number of nodes:5
Enter the adjacency matrix:
0 1 1 0 1
0 0 1 0 0
0 0 0 1 0
0 0 0 0 0
0 0 0 1 0
The adjacency matrix is:
0      1      1      0      1
0      0      1      0      0
0      0      0      1      0
0      0      0      0      0
0      0      0      1      0
The path matrix is:
0      1      1      1      1
0      0      1      1      0
0      0      0      1      0
0      0      0      0      0
0      0      0      1      0
```



#### 4. Implement 0/1 Knapsack problem using Dynamic Programming.

##### 0/1 Knapsack problem:

Given: A set  $S$  of  $n$  items, with each item  $i$  having

- $b_i$  - a positive benefit
- $w_i$  - a positive weight

Goal: Choose items with maximum total benefit but with weight at most  $W$ . i.e.

- Objective: maximize  $\sum_{i \in T} b_i$
- Constraint:  $\sum_{i \in T} w_i \leq W$

**Algorithm:** 0/1Knapsack( $S, W$ )

//Input: set  $S$  of items with benefit  $b_i$  and weight  $w_i$ ; max. weight  $W$

//Output: benefit of best subset with weight at most  $W$

//Sk: Set of items numbered 1 to  $k$ .

//Define  $B[k, w]$  = best selection from  $S_k$  with weight exactly equal to  $w$

```
{
  for w ← 0 to n-1 do
    B[w] ← 0
  for k ← 1 to n do
    {
      for w ← W downto  $w_k$  do
        {
          if  $B[w-w_k] + b_k > B[w]$  then
             $B[w] \leftarrow B[w-w_k] + b_k$ 
        }
      }
    }
}
```

**Complexity:** The Time efficiency and Space efficiency of 0/1 Knapsack algorithm is  $\Theta(nW)$ .

**Program:**

```

#include<stdio.h>
#define MAX 50

int p[MAX],w[MAX],n;
int knapsack(int,int);
int max(int,int);

void main()
{
    int m,i,optsoln;
    clrscr();

    printf("Enter no. of objects: ");
    scanf("%d",&n);

    printf("\nEnter the weights:\n");
    for(i=1;i<=n;i++)
        scanf("%d",&w[i]);

    printf("\nEnter the profits:\n");
    for(i=1;i<=n;i++)
        scanf("%d",&p[i]);

    printf("\nEnter the knapsack capacity:");
    scanf("%d",&m);

    optsoln=knapsack(1,m);

    printf("\nThe optimal solution is:%d",optsoln);
    getch();
}

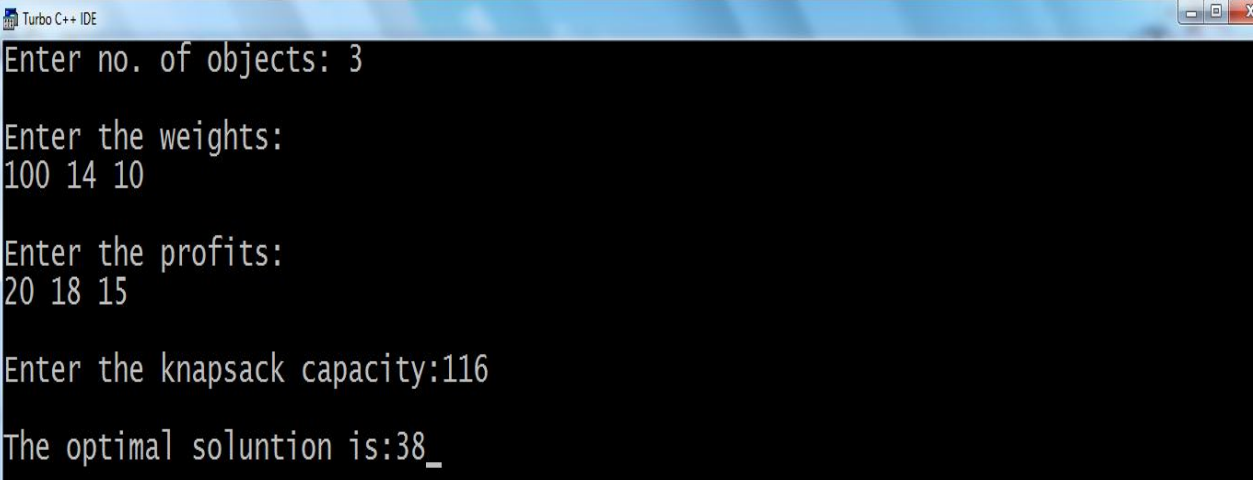
int knapsack(int i,int m)
{
    if(i==n)
        return (w[n]>m) ? 0 : p[n];

    if(w[i]>m)
        return knapsack(i+1,m);

    return max(knapsack(i+1,m),knapsack(i+1,m-w[i])+p[i]);
}

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

```

**OUTPUT:**

```
Turbo C++ IDE
Enter no. of objects: 3
Enter the weights:
100 14 10
Enter the profits:
20 18 15
Enter the knapsack capacity:116
The optimal solution is:38_
```

The screenshot shows a Turbo C++ IDE window with a black background and white text. The text displays the input and output of a program. The input consists of the number of objects (3), their weights (100, 14, 10), their profits (20, 18, 15), and the knapsack capacity (116). The output is the optimal solution value, 38, followed by an underscore.

**5. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.**

**Single Source Shortest Paths Problem:**

For a given vertex called the source in a weighted connected graph, find the shortest paths to all its other vertices. Dijkstra's algorithm is the best known algorithm for the single source shortest paths problem. This algorithm is applicable to graphs with nonnegative weights only and finds the shortest paths to a graph's vertices in order of their distance from a given source. It finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. It is applicable to both undirected and directed graphs

```

Algorithm : Dijkstra(G,s)
//Dijkstra's algorithm for single-source shortest paths
//Input :A weighted connected graph  $G=(V,E)$  with nonnegative weights and its vertex  $s$ 
//Output : The length  $dv$  of a shortest path from  $s$  to  $v$  and its penultimate vertex  $pv$  for
//every  $v$  in  $V$ .
{
    Initialise(Q)    // Initialise vertex priority queue to empty
    for every vertex  $v$  in  $V$  do
    {
         $dv \leftarrow \infty$ ;  $pv \leftarrow \text{null}$ 
        Insert(Q,v,dv) //Initialise vertex priority queue in the priority queue
    }
     $ds \leftarrow 0$ ; Decrease(Q,s ds)    //Update priority of  $s$  with  $ds$ 
     $V_t \leftarrow \emptyset$ 
    for  $i \leftarrow 0$  to  $|V|-1$  do
    {
         $u^* \leftarrow \text{DeleteMin}(Q)$     //delete the minimum priority element
         $V_t \leftarrow V_t \cup \{u^*\}$ 
        for every vertex  $u$  in  $V - V_t$  that is adjacent to  $u^*$  do
        {
            if  $du^* + w(u^*,u) < du$ 
            {
                 $du \leftarrow du^* + w(u^*, u)$ ;  $pu \leftarrow u^*$ 
                Decrease(Q,u,du)
            }
        }
    }
}

```

**Complexity:** The Time efficiency for graphs represented by their weight matrix and the priority queue implemented as an unordered array and for graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is  $O(|E| \log |V|)$ .

**Program:**

```

/*To find shortest paths to other vertices using Dijkstra's algorithm.*/

#include<stdio.h>

void dij(int,int [20][20],int [20],int [20],int);

void main()
{
    int i,j,n,visited[20],source,cost[20][20],d[20];
    clrscr();

    printf("Enter no. of vertices: ");
    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]);
        }
    }
    printf("\nEnter the source node: ");
    scanf("%d",&source);
    dij(source,cost,visited,d,n);
    for(i=1;i<=n;i++)
        if(i!=source)
            printf("\nShortest path from %d to %d is %d",source,i,d[i]);
    getch();
}

void dij(int source,int cost[20][20],int visited[20],int d[20],int n)
{
    int i,j,min,u,w;

    for(i=1;i<=n;i++)
    {
        visited[i]=0;
        d[i]=cost[source][i];
    }

    visited[source]=1;
    d[source]=0;

    for(j=2;j<=n;j++)
    {
        min=999;

        for(i=1;i<=n;i++)
        {
            if(!visited[i])

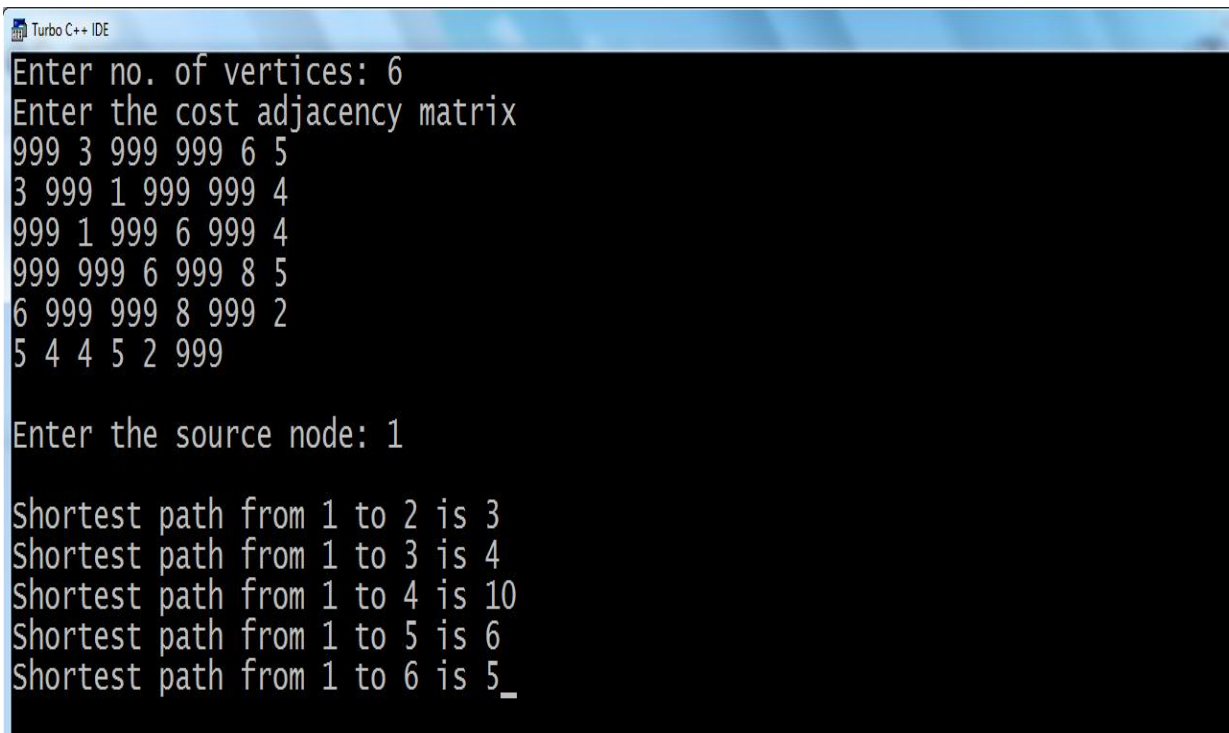
```

```

        {
            if(d[i]<min)
            {
                min=d[i];
                u=i;
            }
        }
    } //for i
    visited[u]=1;

    for(w=1;w<=n;w++)
    {
        if(cost[u][w]!=999 && visited[w]==0)
        {
            if(d[w]>cost[u][w]+d[u])
                d[w]=cost[u][w]+d[u];
        }
    } //for w
} // for j
}

```

**OUTPUT:**


```

Turbo C++ IDE
Enter no. of vertices: 6
Enter the cost adjacency matrix
999 3 999 999 6 5
3 999 1 999 999 4
999 1 999 6 999 4
999 999 6 999 8 5
6 999 999 8 999 2
5 4 4 5 2 999

Enter the source node: 1

Shortest path from 1 to 2 is 3
Shortest path from 1 to 3 is 4
Shortest path from 1 to 4 is 10
Shortest path from 1 to 5 is 6
Shortest path from 1 to 6 is 5_

```

## 6. Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.

### Kruskal's algorithm:

Kruskal's algorithm finds the minimum spanning tree for a weighted connected graph  $G=(V,E)$  to get an acyclic subgraph with  $|V|-1$  edges for which the sum of edge weights is the smallest. Consequently the algorithm constructs the minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of algorithm. The algorithm begins by sorting the graph's edges in non decreasing order of their weights. Then starting with the empty subgraph, it scans the sorted list adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

#### **Algorithm :** Kruskal( $G$ )

```
// Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G=(V,E)$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
{
    Sort  $E$  in non decreasing order of the edge weights  $w(e_{i1}) \leq \dots \leq w(e_{i|E|})$ 
     $E_T \leftarrow \emptyset$  ; ecounter  $\leftarrow 0$  //Initialize the set of tree edges and its size
     $k \leftarrow 0$  //initialize the number of processed edges
    while ecounter  $< |V|-1$  do
    {
         $k \leftarrow k+1$ 
        if  $E_T \cup \{e_{ik}\}$  is acyclic
             $E_T \leftarrow E_T \cup \{e_{ik}\}$ ; ecounter  $\leftarrow$  ecounter+1
    }
    return  $E_T$ 
}
```

**Complexity:** With an efficient sorting algorithm, the time efficiency of kruskal's algorithm will be in  $O(|E| \log |E|)$ .

### **Program:**

```
#include<stdio.h>
int ne=1,min_cost=0;

void main()
{
    int n,i,j,min,a,u,b,v,cost[20][20],parent[20];
    clrscr();

    printf("Enter the no. of vertices:");
    scanf("%d",&n);

    printf("\nEnter the cost matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
```

```

scanf("%d",&cost[i][j]);

for(i=1;i<=n;i++)
    parent[i]=0;

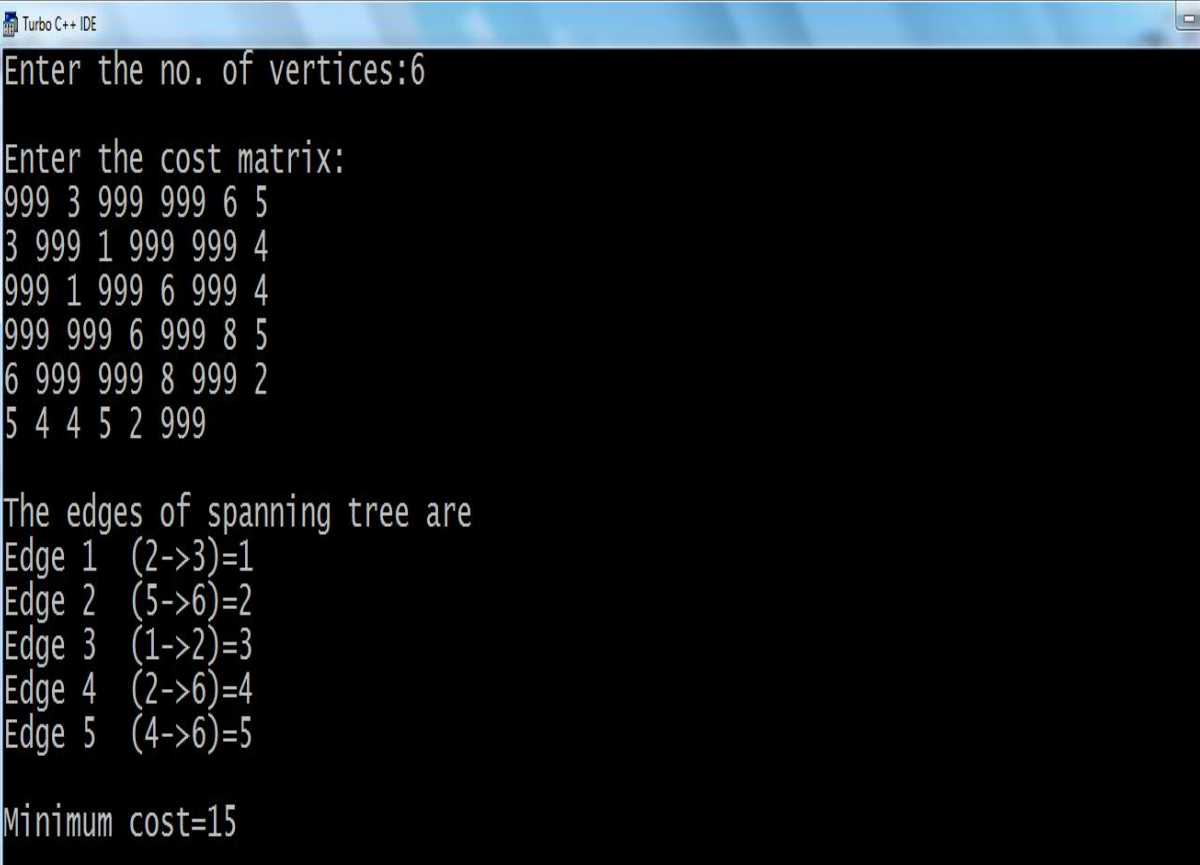
printf("\nThe edges of spanning tree are\n");
while(ne<n)
{
    min=999;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            if(cost[i][j]<min)
            {
                min=cost[i][j];
                a=u=i;
                b=v=j;
            }
        }
    }
    while(parent[u])
        u=parent[u];

    while(parent[v])
        v=parent[v];

    if(u!=v)
    {
        printf("Edge %d\t(%d->%d)=%d\n",ne++,a,b,min);
        min_cost=min_cost+min;
        parent[v]=u;
    }
    cost[a][b]=cost[a][b]=999;
}
printf("\nMinimum cost=%d\n",min_cost);
getch();
}

```



**OUTPUT:**

```
Turbo C++ IDE
Enter the no. of vertices:6

Enter the cost matrix:
999 3 999 999 6 5
3 999 1 999 999 4
999 1 999 6 999 4
999 999 6 999 8 5
6 999 999 8 999 2
5 4 4 5 2 999

The edges of spanning tree are
Edge 1 (2->3)=1
Edge 2 (5->6)=2
Edge 3 (1->2)=3
Edge 4 (2->6)=4
Edge 5 (4->6)=5

Minimum cost=15
```

**7a. Print all the nodes reachable from a given starting node in a digraph using BFS method.****Breadth First Search:**

BFS explores graph moving across to all the neighbors of last visited vertex traversals i.e., it proceeds in a concentric manner by visiting all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it and so on, until all the vertices in the same connected component as the starting vertex are visited. Instead of a stack, BFS uses queue.

**Algorithm : BFS(G)**

//Implements a breadth-first search traversal of a given graph

//Input: Graph  $G = (V, E)$

//Output: Graph  $G$  with its vertices marked with consecutive integers in the order they

//have been visited by the BFS traversal

```
{
    mark each vertex with 0 as a mark of being "unvisited"
    count ← 0
    for each vertex v in V do
    {
        if v is marked with 0
            bfs(v)
    }
}
```

**Algorithm : bfs(v)**

//visits all the unvisited vertices connected to vertex v and assigns them the numbers

//in order they are visited via global variable *count*

```
{
    count ← count + 1
    mark v with count and initialize queue with v
    while queue is not empty do
    {
        a := front of queue
        for each vertex w adjacent to a do
        {
            if w is marked with 0
            {
                count ← count + 1
                mark w with count
                add w to the end of the queue
            }
        }
        remove a from the front of the queue
    }
}
```

**Complexity:**

BFS has the same efficiency as DFS: it is  $\Theta(V^2)$  for Adjacency matrix representation and  $\Theta(V+E)$  for Adjacency linked list representation.

**Program:**

```
/*Print all the nodes reachable from a given starting node in a digraph using BFS method.*/
```

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

```
void BFS(int [20][20],int,int [20],int);
```

```
void main()
```

```
{
    int n,a[20][20],i,j,visited[20],source;
    clrscr();

    printf("Enter the number of vertices:");
    scanf("%d",&n);

    printf("\nEnter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);

    for(i=1;i<=n;i++)
        visited[i]=0;

    printf("\nEnter the source node:");
    scanf("%d",&source);
    visited[source]=1;

    BFS(a,source,visited,n);

    for(i=1;i<=n;i++)
    {
        if(visited[i]!=0)
            printf("\n Node %d is reachable",i);
        else
            printf("\n Node %d is not reachable",i);
    }
    getch();
}
```

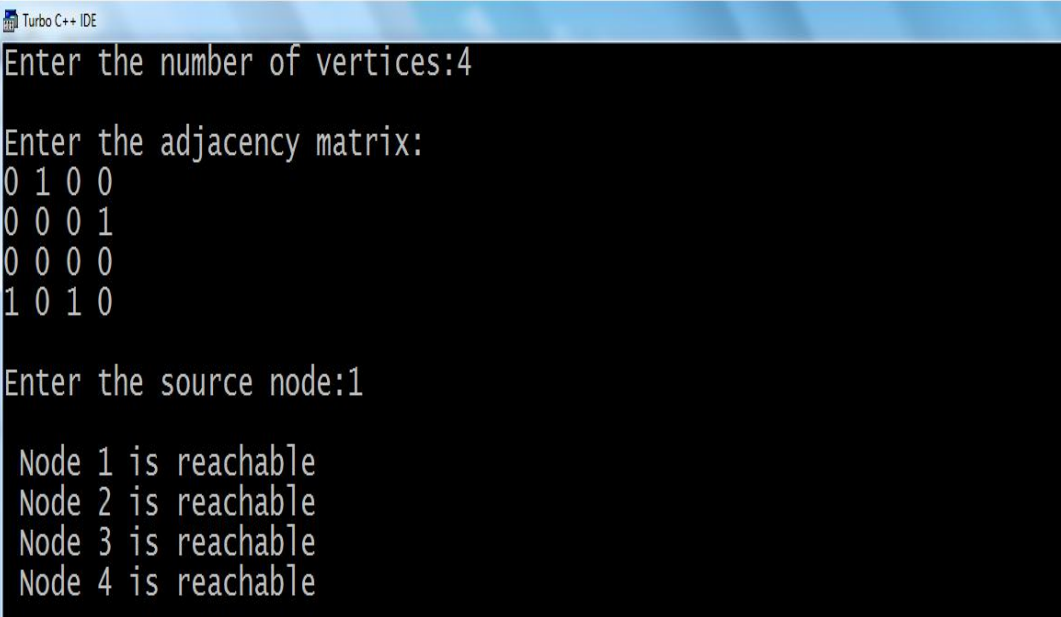
```
void BFS(int a[20][20],int source,int visited[20],int n)
```

```
{
    int queue[20],f,r,u,v;

    f=0;
    r=-1;
    queue[++r]=source;

    while(f<=r)
    {
        u=queue[f++];
        for(v=1;v<=n;v++)
        {
            if(a[u][v]==1 && visited[v]==0)
            {
```

```
        queue[++r]=v;  
        visited[v]=1;  
    }  
} //for v  
} // while  
}
```

**OUTPUT:**

The screenshot shows the Turbo C++ IDE window with the following text:

```
Turbo C++ IDE  
Enter the number of vertices:4  
  
Enter the adjacency matrix:  
0 1 0 0  
0 0 0 1  
0 0 0 0  
1 0 1 0  
  
Enter the source node:1  
  
Node 1 is reachable  
Node 2 is reachable  
Node 3 is reachable  
Node 4 is reachable
```

**7b. Check whether a given graph is connected or not using DFS method.****Depth First Search:**

Depth-first search starts visiting vertices of a graph at an arbitrary vertex by marking it as having been visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. This process continues until a vertex with no adjacent unvisited vertices is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end.

**Algorithm : DFS(G)**

```
//Implements a depth-first search traversal of a given graph
//Input : Graph G = (V,E)
//Output : Graph G with its vertices marked with consecutive integers in the order they
//have been first encountered by the DFS traversal
{
    mark each vertex in V with 0 as a mark of being "unvisited".
    count ← 0
    for each vertex v in V do
        if v is marked with 0
            dfs(v)
}
```

**Algorithm : dfs(v)**

```
//visits recursively all the unvisited vertices connected to vertex v by a path
//and numbers them in the order they are encountered via global variable count
{
    count ← count+1
    mark v with count
    for each vertex w in V adjacent to v do
        if w is marked with 0
            dfs(w)
}
```

**Complexity:** For the adjacency matrix representation, the traversal time efficiency is in  $\Theta(|V|^2)$  and for the adjacency linked list representation, it is in  $\Theta(|V|+|E|)$ , where  $|V|$  and  $|E|$  are the number of graph's vertices and edges respectively.

**Program:**

```

/* Check whether a given graph is connected or not using DFS method.*/

#include<stdio.h>

void DFS(int [20][20],int,int [20],int);

void main()
{
    int n,a[20][20],i,j,visited[20],source;
    clrscr();

    printf("Enter the number of vertices: ");
    scanf("%d",&n);

    printf("\nEnter the adjacency matrix:\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);

    for(i=1;i<=n;i++)
        visited[i]=0;

    printf("\nEnter the source node: ");
    scanf("%d",&source);

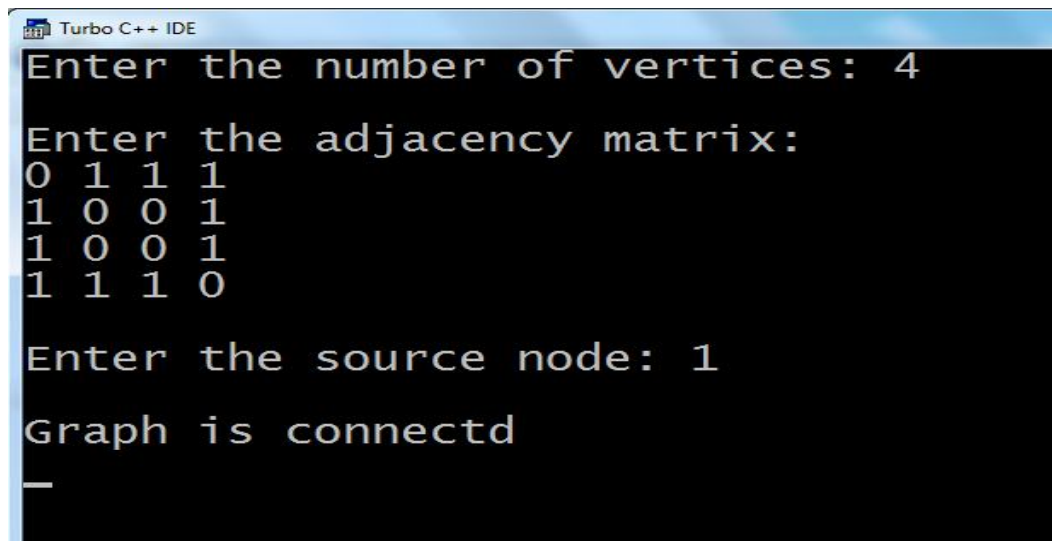
    DFS(a,source,visited,n);

    for(i=1;i<=n;i++)
    {
        if(visited[i]==0)
        {
            printf("\nGraph is not connected");
            getch();
            exit(0);
        }
    }
    printf("\nGraph is connectd\n");
    getch();
}

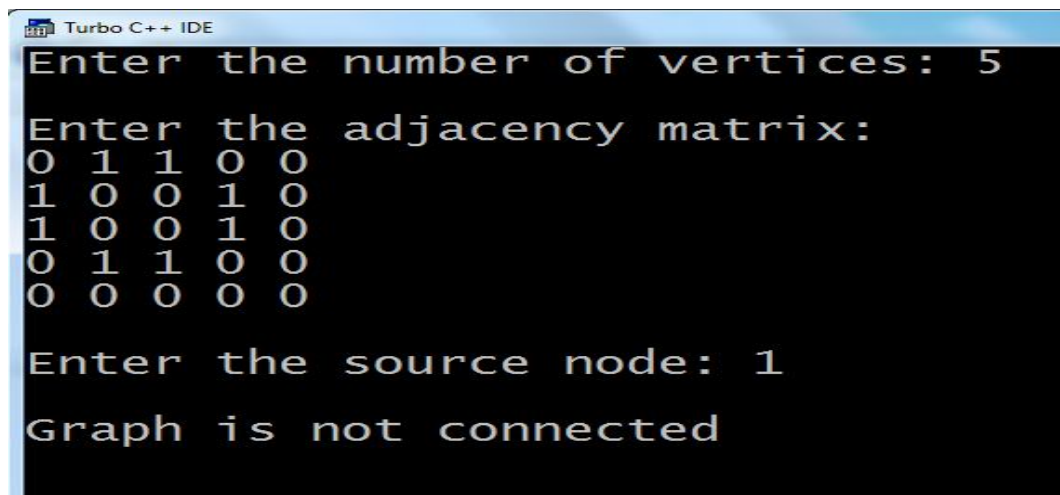
void DFS(int a[20][20],int u,int visited[20],int n)
{
    int v;

    visited[u]=1;
    for(v=1;v<=n;v++)
    {
        if(a[u][v]==1 && visited[v]==0)
            DFS(a,v,visited,n);
    }
}

```

**OUTPUT:**

```
Turbo C++ IDE
Enter the number of vertices: 4
Enter the adjacency matrix:
0 1 1 1
1 0 0 1
1 0 0 1
1 1 1 0
Enter the source node: 1
Graph is connectd
_
```



```
Turbo C++ IDE
Enter the number of vertices: 5
Enter the adjacency matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 1 0
0 1 1 0 0
0 0 0 0 0
Enter the source node: 1
Graph is not connected
```

8. Find a subset of a given set  $S = \{s_1, s_2, \dots, s_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ . For example, if  $S = \{1, 2, 5, 6, 8\}$  and  $d = 9$  there are two solutions  $\{1, 2, 6\}$  and  $\{1, 8\}$ . A suitable message is to be displayed if the given problem instance doesn't have a solution.

### Sum of Subsets

Subset-Sum Problem is to find a subset of a given set  $S = \{s_1, s_2, \dots, s_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ . It is assumed that the set's elements are sorted in increasing order. The state-space tree can then be constructed as a binary tree and applying backtracking algorithm, the solutions could be obtained. Some instances of the problem may have no solutions

#### **Algorithm** SumOfSub( $s, k, r$ )

//Find all subsets of  $w[1 \dots n]$  that sum to  $m$ . The values of  $x[j]$ ,  $1 \leq j < k$ , have already //been determined.  $s = \sum_{j=1}^{k-1} w[j] * x[j]$  and  $r = \sum_{j=k}^n w[j]$ . The  $w[j]$ 's are in ascending order.

```
{
  x[k] ← 1 //generate left child
  if (s+w[k] = m)
    write (x[1...n]) //subset found
  else if ( s + w[k]+w[k+1] <= m)
    SumOfSub( s + w[k], k+1, r-w[k])
  //Generate right child
  if( (s + r - w[k] >= m) and (s + w[k+1] <= m) )
  {
    x[k] ← 0
    SumOfSub( s, k+1, r-w[k] )
  }
}
```

#### **Complexity:**

Subset sum problem solved using backtracking generates at each step maximal two new subtrees, and the running time of the bounding functions is linear, so the running time is  $O(2^n)$ .



**Program:**

/\* Find a subset of a given set  $S=\{s_1,s_2,...s_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ . A suitable message is to be displayed if the given problem instance doesn't have a solution.\*/

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

```
void subset(int,int,int);
int x[10],w[10],d,count=0;
```

```
void main()
{
    int i,n,sum=0;
    clrscr();

    printf("Enter the no. of elements: ");
    scanf("%d",&n);

    printf("\nEnter the elements in ascending order:\n");
    for(i=0;i<n;i++)
        scanf("%d",&w[i]);

    printf("\nEnter the sum: ");
    scanf("%d",&d);

    for(i=0;i<n;i++)
        sum=sum+w[i];

    if(sum<d)
    {
        printf("No solution\n");
        getch();
        return;
    }

    subset(0,0,sum);

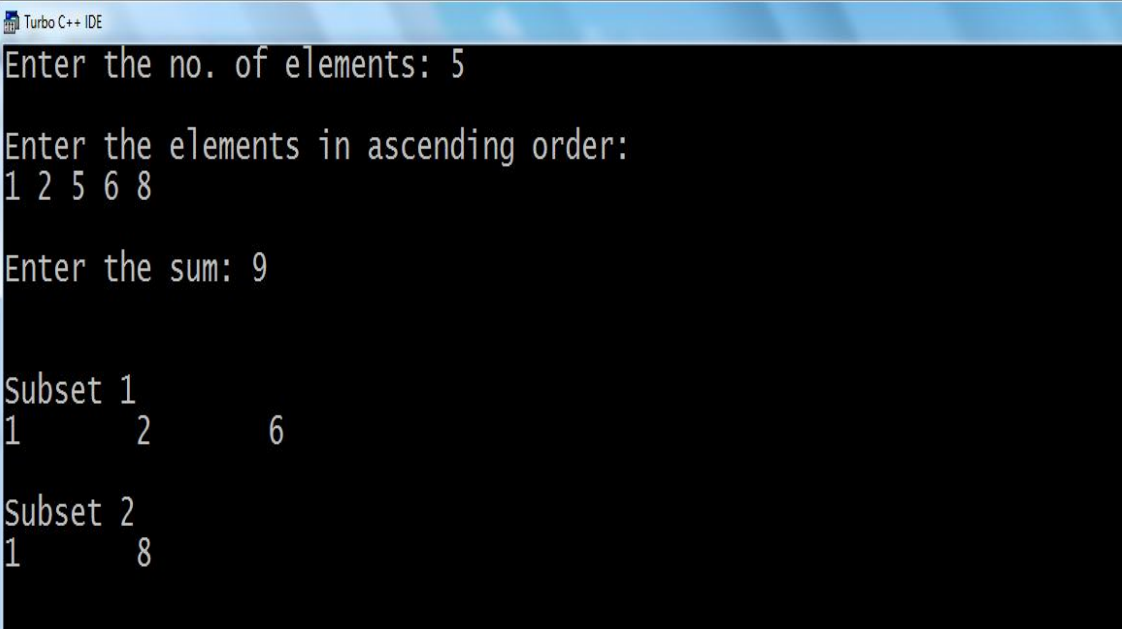
    if(count==0)
    {
        printf("No solution\n");
        getch();
        return;
    }
    getch();
}
```

```
void subset(int cs,int k,int r)
{
    int i;

    x[k]=1;

    if(cs+w[k]==d)
```

```
{
    printf("\n\nSubset %d\n",++count);
    for(i=0;i<=k;i++)
        if(x[i]==1)
            printf("%d\t",w[i]);
    }
else
    if(cs+w[k]+w[k+1]<=d)
        subset(cs+w[k],k+1,r-w[k]);
if(cs+r-w[k]>=d && cs+w[k]<=d)
{
    x[k]=0;
    subset(cs,k+1,r-w[k]);
}
}
```

**OUTPUT:**

The screenshot shows the Turbo C++ IDE window with the following text displayed in the console:

```
Turbo C++ IDE
Enter the no. of elements: 5
Enter the elements in ascending order:
1 2 5 6 8
Enter the sum: 9

Subset 1
1      2      6

Subset 2
1      8
```

9. Implement any scheme to find the optimal solution for the Traveling Salesperson problem and then solve the same problem instance using any approximation algorithm and determine the error in the approximation.

### **Traveling Salesperson problem:**

Given n cities, a salesperson starts at a specified city (source), visit all n-1 cities only once and return to the city from where he has started. The objective of this problem is to find a route through the cities that minimizes the cost and thereby maximizing the profit.

**Algorithm** TSP(start city, current city,next city, path)  
**//Purpose:** To find the solution for TSP problem using exhaustive search  
**//Input:** The start city, current city,next city and the path  
**//Output:** The minimum distance covered along with the path

**Step 1:** Check for the disconnection between the current city and the next city  
**Step 2:** Check whether the travelling sales person has visited all the cities  
**Step 3:** Find the next city to be visited  
**Step 4:** Find the solution and terminate

### **Program:**

```
#include<stdio.h>

int s,c[100][100],ver;
float optimum=999,sum;

/* function to swap array elements */
void swap(int v[], int i, int j)
{
    int t;
    t = v[i];
    v[i] = v[j];
    v[j] = t;
}
```

```

/* recursive function to generate permutations */
void brute_force(int v[], int n, int i)
{
// this function generates the permutations of the array from element i to element n-1
    int    j,sum1,k;
//if we are at the end of the array, we have one permutation
    if (i == n)
    {
        if(v[0]==s)
        {
            for (j=0; j<n; j++)
                printf ("%d ", v[j]);
            sum1=0;
            for( k=0;k<n-1;k++)
            {
                sum1=sum1+c[v[k]][v[k+1]];
            }
            sum1=sum1+c[v[n-1]][s];
            printf("sum = %d\n",sum1);
            getch();

            if (sum1<optimum)
                optimum=sum1;
        }
    }
    else
// recursively explore the permutations starting at index i going through index n-1*/
        for (j=i; j<n; j++)
        { /* try the array with i and j switched */
            swap (v, i, j);
            brute_force (v, n, i+1);
            /* swap them back the way they were */
            swap (v, i, j);
        }
}

void nearest_neighbour(int ver)
{
    int min,p,i,j,vis[20],from;

    for(i=1;i<=ver;i++)
        vis[i]=0;

    vis[s]=1;
    from=s;
    sum=0;

    for(j=1;j<ver;j++)
    {
        min=999;

        for(i=1;i<=ver;i++)
            if(vis[i] !=1 && c[from][i]<min && c[from][i] !=0 )

```

```
        {
            min= c[from][i];
            p=i;
        }
        vis[p]=1;
        from=p;
        sum=sum+min;
    }
    sum=sum+c[from][s];
}

void main ()
{
    int  ver,v[100],i,j;
    clrscr();

    printf("Enter n : ");
    scanf("%d",&ver);

    for (i=0; i<ver; i++)
        v[i] = i+1;

    printf("Enter cost matrix\n");
    for(i=1;i<=ver;i++)
        for(j=1;j<=ver;j++)
            scanf("%d",&c[i][j]);

    printf("\nEnter source : ");
    scanf("%d",&s);
    brute_force (v, ver, 0);
    printf("\nOptimum solution with brute force technique is=%f\n",optimum);

    nearest_neighbour(ver);
    printf("\nSolution with nearest neighbour technique is=%f\n",sum);

    printf("The approximation val is=%f",((sum/optimum)-1)*100);
    printf(" % ");

    getch();
}
```

**OUTPUT:**

Turbo C++ IDE

```
Enter n : 5
Enter cost matrix
0 3 1 5 8
3 0 6 7 9
1 6 0 4 2
5 7 4 0 3
8 9 2 3 0
```

```
Enter source : 1
1 2 3 4 5 sum = 24
1 2 3 5 4 sum = 19
1 2 4 3 5 sum = 24
1 2 4 5 3 sum = 16
1 2 5 4 3 sum = 20
1 2 5 3 4 sum = 23
1 3 2 4 5 sum = 25
1 3 2 5 4 sum = 24
1 3 4 2 5 sum = 29
1 3 4 5 2 sum = 20
1 3 5 4 2 sum = 16
1 3 5 2 4 sum = 24
1 4 3 2 5 sum = 32
1 4 3 5 2 sum = 23
1 4 2 3 5 sum = 28
1 4 2 5 3 sum = 24
1 4 5 2 3 sum = 24
1 4 5 3 2 sum = 19
1 5 3 4 2 sum = 24
1 5 3 2 4 sum = 28
1 5 4 3 2 sum = 24
1 5 4 2 3 sum = 25
1 5 2 4 3 sum = 29
1 5 2 3 4 sum = 32
```

Optimum solution with brute force technique is=16.000000

Solution with nearest neighbour technique is=16.000000

The approximation val is=0.000000 %

## 10. Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.

### Prim's Algorithm:

Prim's algorithm finds the minimum spanning tree for a weighted connected graph  $G=(V,E)$  to get an acyclic subgraph with  $|V|-1$  edges for which the sum of edge weights is the smallest. Consequently the algorithm constructs the minimum spanning tree as expanding sub-trees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set  $V$  of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

#### **Algorithm :** Prim( $G$ )

// Prim's algorithm for constructing a minimum spanning tree

//Input: A weighted connected graph  $G=(V,E)$

//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$

```
{
     $V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex
     $E_T \leftarrow \emptyset$ 

    for  $i \leftarrow 0$  to  $|V| - 1$  do
        find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
        such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ 
         $V_T \leftarrow V_T \cup \{u^*\}$ 
         $E_T \leftarrow E_T \cup \{e^*\}$ 

    return  $E_T$ 
}
```

**Complexity:** The time efficiency of prim's algorithm will be in  $O(|E| \log |V|)$ .

**Program:**

```

#include<stdio.h>

int ne=1,min_cost=0;

void main()
{
    int n,i,j,min,cost[20][20],a,u,b,v,source,visited[20];
    clrscr();

    printf("Enter the no. of nodes:");
    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]);
        }
    }

    for(i=1;i<=n;i++)
        visited[i]=0;

    printf("Enter the root node:");
    scanf("%d",&source);
    visited[source]=1;

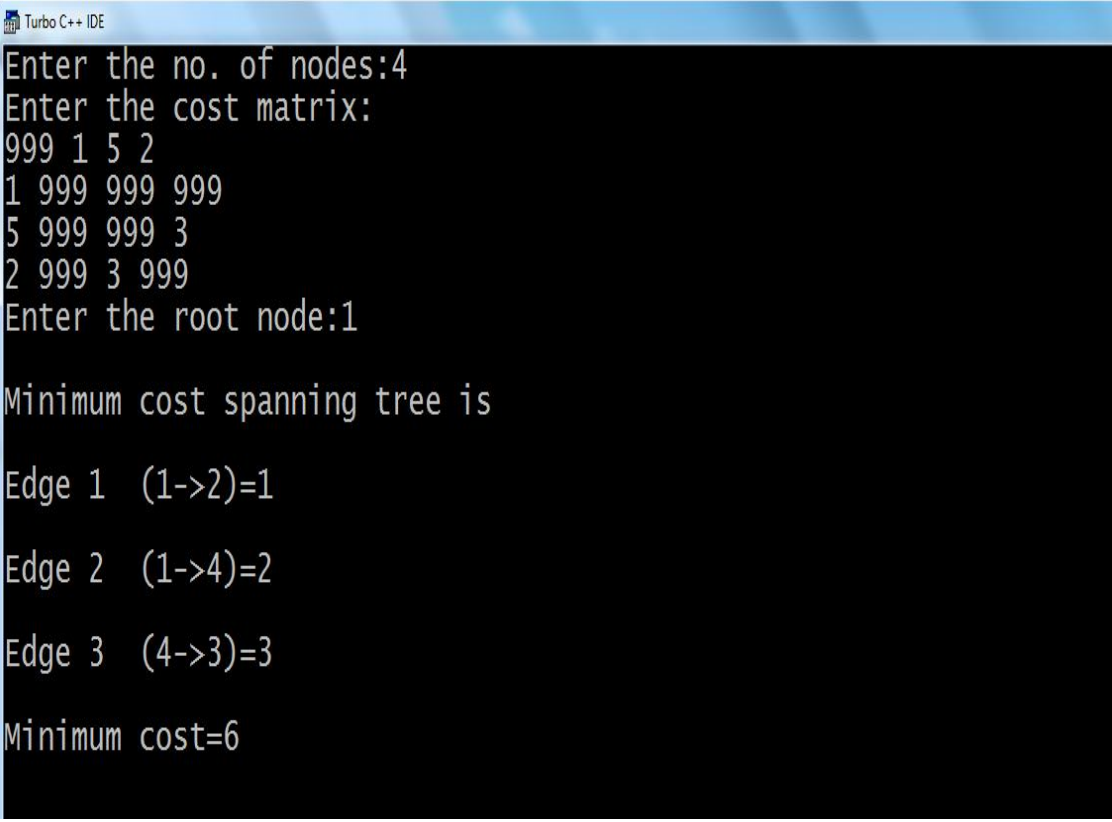
    printf("\nMinimum cost spanning tree is\n");
    while(ne<n)
    {
        min=999;
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
            {
                if(cost[i][j]<min)
                    if(visited[i]==0)
                        continue;
                else
                {
                    min=cost[i][j];
                    a=u=i;
                    b=v=j;
                }
            }
        }

        if(visited[u]==0||visited[v]==0)
        {

```



```
        printf("\nEdge %d\t(%d->%d)=%d\n",ne++,a,b,min);
        min_cost=min_cost+min;
        visited[b]=1;
    }
    cost[a][b]=cost[b][a]=999;
}
printf("\nMinimum cost=%d\n",min_cost);
getch();
}
```

**OUTPUT:**

The screenshot shows the Turbo C++ IDE with a black background and white text. The program prompts the user for the number of nodes (4), the cost matrix, and the root node (1). It then displays the minimum cost spanning tree with three edges: (1->2)=1, (1->4)=2, and (4->3)=3, resulting in a minimum cost of 6.

```
Turbo C++ IDE
Enter the no. of nodes:4
Enter the cost matrix:
999 1 5 2
1 999 999 999
5 999 999 3
2 999 3 999
Enter the root node:1

Minimum cost spanning tree is

Edge 1  (1->2)=1
Edge 2  (1->4)=2
Edge 3  (4->3)=3

Minimum cost=6
```

**11. Implement All-Pairs Shortest Paths Problem using Floyd's algorithm. Parallelize this algorithm, implement it using OpenMP and determine the speed-up achieved.**

**Floyd's Algorithm:**

Floyd's algorithm is applicable to both directed and undirected graphs provided that they do not contain a cycle. It is convenient to record the lengths of shortest path in an  $n$ - by-  $n$  matrix  $D$  called the distance matrix. The element  $d_{ij}$  in the  $i$ th row and  $j$ th column of matrix indicates the shortest path from the  $i$ th vertex to  $j$ th vertex ( $1 \leq i, j \leq n$ ). The element in the  $i$ th row and  $j$ th column of the current matrix  $D^{(k-1)}$  is replaced by the sum of elements in the same row  $i$  and  $k$ th column and in the same column  $j$  and the  $k$ th column if and only if the latter sum is smaller than its current value.

**Algorithm** Floyd( $W[1..n, 1..n]$ )

//Implements Floyd's algorithm for the all-pairs shortest paths problem

//Input: The weight matrix  $W$  of a graph

//Output: The distance matrix of shortest paths length

```
{
    D ← W
    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
}
```

**Complexity:** The time efficiency of Floyd's algorithm is cubic i.e.  $\Theta(n^3)$

**Program:**

```

#include "stdafx.h"
#include<stdio.h>
#include<iostream>
#include<omp.h>
#include<conio.h>

void floyd(int[10][10],int);
int min(int,int);

void main()
{
    int n,a[10][10],i,j;
    printf("Enter the no.of nodes : ");
    scanf("%d",&n);

    printf("\nEnter the cost adjacency matrix\n");
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);

    floyd(a,n);
    getch();
}

void floyd(int a[10][10],int n)
{
    int d[10][10],i,j,k;

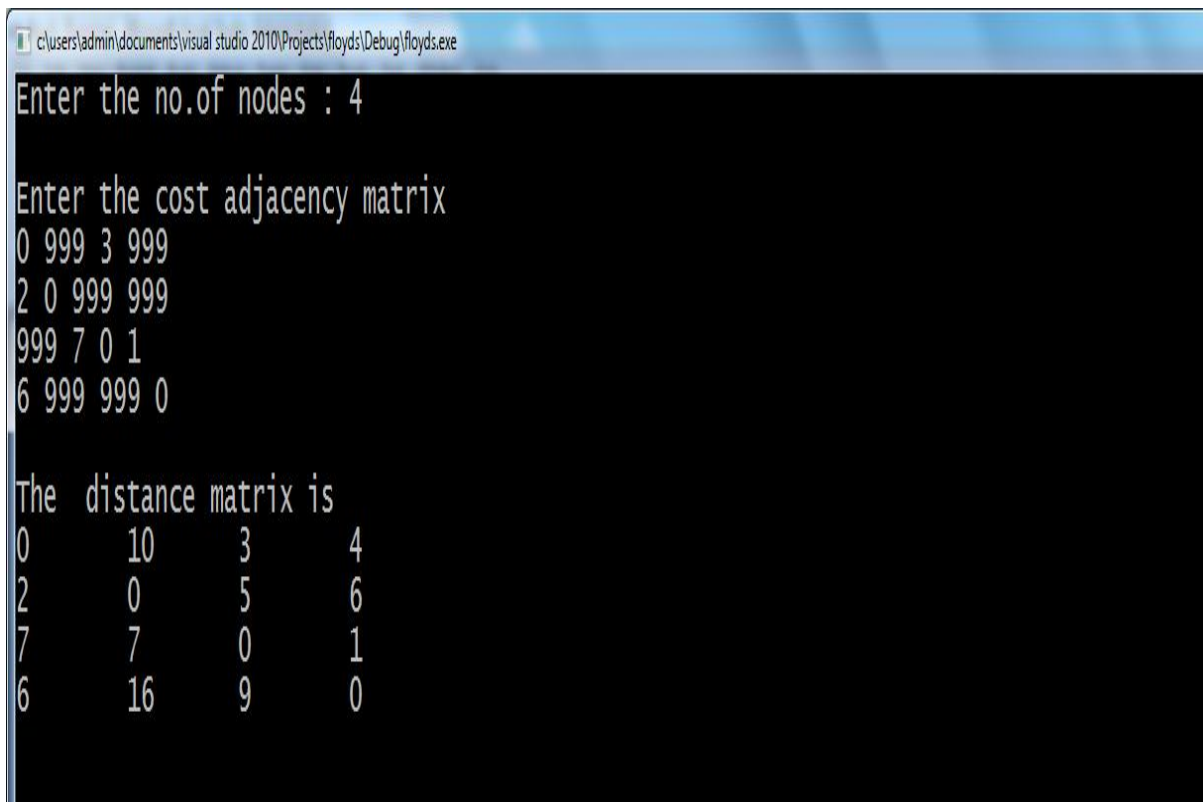
    #pragma omp parallel for
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            d[i][j]=a[i][j];
    }

    #pragma omp parallel for
    for(k=1;k<=n;k++)
    {
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
            {
                d[i][j]=min(d[i][j],d[i][k]+d[k][j]);
            }
        }
    }

    printf("\nThe distance matrix is\n");
    #pragma omp parallel for
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)

```

```
        {  
            printf("%d\t",d[i][j]);  
        }  
        printf("\n");  
    }  
}  
  
int min (int a,int b)  
{  
    if(a<b)  
        return a;  
    else  
        return b;  
}
```

**OUTPUT:**

```
c:\users\admin\documents\visual studio 2010\Projects\floyds\Debug\floyds.exe  
Enter the no.of nodes : 4  
  
Enter the cost adjacency matrix  
0 999 3 999  
2 0 999 999  
999 7 0 1  
6 999 999 0  
  
The distance matrix is  
0      10      3      4  
2      0      5      6  
7      7      0      1  
6      16     9      0
```

## 12. Implement N Queen's problem using Back Tracking.

### N Queen's problem:

The  $n$ -queens problem consists of placing  $n$  queens on an  $n \times n$  checker board in such a way that they do not threaten each other, according to the rules of the game of chess. Every queen on a checker square can reach the other squares that are located on the same horizontal, vertical, and diagonal line. So there can be at most one queen at each horizontal line, at most one queen at each vertical line, and at most one queen at each of the  $4n-2$  diagonal lines. Furthermore, since we want to place as many queens as possible, namely exactly  $n$  queens, there must be exactly one queen at each horizontal line and at each vertical line. The concept behind backtracking algorithm which is used to solve this problem is to successively place the queens in columns. When it is impossible to place a queen in a column (it is on the same diagonal, row, or column as another token), the algorithm backtracks and adjusts a preceding queen

#### **Algorithm** NQueens (k, n)

//Using backtracking, this procedure prints all possible placements of n queens

//on an  $n \times n$  chessboard so that they are non-attacking

```
{
    for i ← 1 to n do
    {
        if(Place(k,i) )
        {
            x[k] ← i
            if (k=n)
                write ( x[1...n])
            else
                Nqueens (k+1, n)
        }
    }
}
```

#### **Algorithm** Place( k, i)

//Returns true if a queen can be placed in kth row and ith column. Otherwise it

//returns false. x[] is a global array whose first (k-1) values have been set. Abs(r)

//returns the absolute value of r.

```
{
    for j ← 1 to k-1 do
    {
        if ( (x[j]=i or Abs(x[j]-i) = Abs(j-k) )
        {
            return false
        }
    }
}
```

### **Complexity:**

The power of the set of all possible solutions of the  $n$  queen's problem is  $n!$  and the bounding function takes a linear amount of time to calculate, therefore the running time of the  $n$  queens problem is  $O(n!)$ .

**Program:**

```

/* Implement N Queen's problem using Back Tracking.*/

#include<stdio.h>

void nqueens(int);
int place(int[],int);
void printsolution(int,int[]);

void main()
{
    int n;
    clrscr();
    printf("Enter the no.of queens: ");
    scanf("%d",&n);
    nqueens(n);
    getch();
}

void nqueens(int n)
{
    int x[10],count=0,k=1;
    x[k]=0;

    while(k!=0)
    {
        x[k]=x[k]+1;

        while(x[k]<=n&&(!place(x,k)))
            x[k]=x[k]+1;

        if(x[k]<=n)
        {
            if(k==n)
            {
                count++;
                printf("\nSolution %d\n",count);
                printsolution(n,x);
            }
            else
            {
                k++;
                x[k]=0;
            }
        }
        else
        {
            k--; //backtracking
        }
    }
    return;
}

```

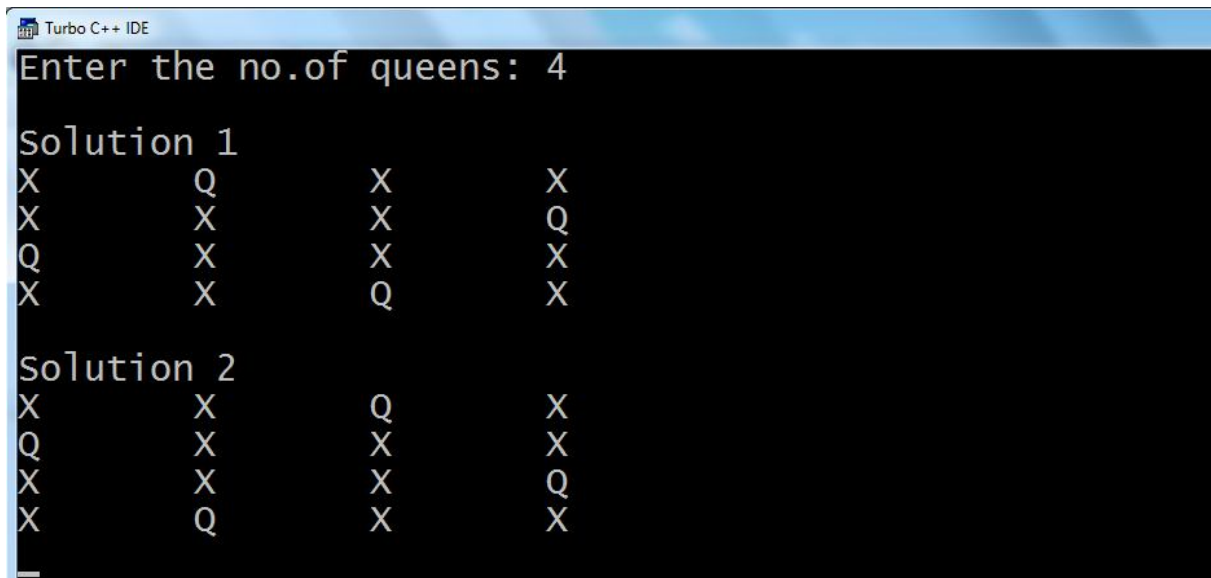
```
int place(int x[],int k)
{
    int i;
    for(i=1;i<k;i++)
        if(x[i]==x[k]||(abs(x[i]-x[k]))==abs(i-k))
            return 0;
    return 1;
}
```

```
void printsolution(int n,int x[])
{
    int i,j;
    char c[10][10];

    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            c[i][j]='X';
    }

    for(i=1;i<=n;i++)
        c[i][x[i]]='Q';

    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
        {
            printf("%c\t",c[i][j]);
        }
        printf("\n");
    }
}
```

**OUTPUT:**

```
Turbo C++ IDE
Enter the no.of queens: 4

Solution 1
X      Q      X      X
X      X      X      Q
Q      X      X      X
X      X      Q      X

Solution 2
X      X      Q      X
Q      X      X      X
X      X      X      Q
X      Q      X      X
```