DESIGN AND ANALYSIS OF ALGORITHMS MANUAL

(2021-2022)

COURSE TITLE: ***DESIGN AND ANALYSIS OF ALGORITHMS LAB***

COURSE CODE:

SEMESTER: ***IV***

STREAM: ***COMPUTER ENGINEERING***

FACULTY: ***Prof. Khushbu Wanjari***

LAB OBJECTIVES:

- To learn the importance of designing an algorithm in an effective way by considering space and time complexity
- To learn divide and conquer strategy based algorithms
- To learn greedy method based algorithms
- To learn the dynamic programming design techniques
- To develop Recursive backtracking algorithms
- To learn graph search and network flow algorithms

COURSE PREREQUISITES: Expected Prior Knowledge and Skills: Proficiency in a C & C++ programming language, basic program design concepts (e.g, pseudo code), proof techniques, familiarity with trees and graph data structures, familiarity with basic algorithms such as those for searching, and sorting, knowledge of Discrete Structures as minimum cost spanning trees.

LIST OF LAB PROGRAMS

SNO	PROGRAM
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. The elements can be read from a file or can be generated using the random number generator.
2.	Implement a 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 .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 Dijikstra'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 = {s1, s2,….., sN} 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.
12.	Implement N Queen's problem using Back Tracking.

Program Number 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 is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quick Sort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

The key process in quick Sort is partition (). Target of partition() is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

```
1 Algorithm Partition(a,m,p)
2 // Within a[m],a[m+1],... ,a[p-1]the elements are
3 // rearranged in such a manner that if initially t = a[m],
4 // then after completion a[q] = t for some q between m
5 // and p-1,a[k] ≤ t for m ≤ k < g, and a[k] ≥ t
6 // for q <k <p. q is returned. Set a[p] = ∞.
7 {
8       v:=a[m]; i:=m; j :=p;
9       repeat
10      {
11              repeat
12                      i:=i+1;
13              until(a[i] > v);
14               repeat
15                      j: =j-1;
16              until (a[j] <v);
17              if (i <j) then     lnterchange(a,i,j);
18      } until(i >= j);
19       a[m] :=a[j];  a[j] :=v;  return j;
20}
```

```
1 Algorithm Interchange (a,i,j)
2 // Exchange a[i] with a[j].
3 {
4         p: =a[i];
5          a[i]:=a[j];  a[j]:=p;
6
```

```
1 Algorithm QuickSort(p, q)
2 // Sorts the elements a[p],..., a[q] which reside in the global
3 // array a[l:n] into ascending order; a[n+1]is considered to
4 //be defined and must be >= all the elements in a[l:n].
5 {
6         if (p <q) then //If there are more than one element
7         {
8          // divide P into two sub problems.
9                   j :=Partition(a,p, q + 1);
10                  // j is the position of the partitioning element.
11         // Solve the sub problems.
12                  QuickSort(p,j-1);
13                  QuickSort(j+ l,q);
14        // There is no need for combining solutions.
15        }
16}
```

Complexity:

Best case	Average case	Worst case
O (nlogn)	O (nlogn)	O (n^2)

PROGRAM:

```
#include<iostream>
#include<ctime>
#include<stdlib.h>
#include<dos.h>
#define max 500
using namespace std;
void qsort(int [],int,int);
int partition(int [],int,int);
int main()
{
        int a[max],i,n;
        clock_t s,e,z;
        s=clock();
        cout<<"Enter the value of n:";
        cin>>n;
        for(i=0;i<n;i++)
        a[i]=rand()%100;
        cout<<"\nThe array elements before\n";
        for(i=0;i<n;i++)
        cout<<a[i]<<"\t";
        qsort(a,0,n-1);
        cout<<"\nElements of the array after sorting are:\n";
        for(i=0;i<n;i++)
        cout<<a[i]<<"\t";
        e=clock();
        z=e-s;
        cout<<"\nTime taken:"<<z/CLOCKS_PER_SEC<<" seconds";
}
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;
        }
        }
}
```

Program Number 2:

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 is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The Merge() function is used for merging two halves. The Mergesort() is a key process that sorts the A[l..m] and A[m+1..r] into two sorted sub-arrays.

```
1 Algorithm MergeSort(low,high)
2 // a[low :high]is a global array to be sorted.
3 // Small(P)is true if there is only one element
4 // to sort. In this case the list is already sorted.
5 {
6       if (low <high) then //If there are more than one element
7       {
8               // Divide P into sub problems.
9                       // Find where to split the set.
10                      mid:=[(low+high)/2];
11              // Solve the sub problems.
12               MergeSort(low, mid);
13               MergeSort(mid+1 ,high);
14               // Combine the solutions.
15       Merge(low,mid,high);16
         }
17}
```

```
1 Algorithm Merge(low,mid,high)
2 //a[low:high] is a global array containing two sorted
3//subsets in a[low:mid] and in a[mid+1:high].The goal
4//is to merge these two sets into a single set residing
5//in a[low :high]. b[ ]is an auxiliary global array.
6{
7       h :=low; i :=low; j :=mid+ 1;
8       while ((h <=mid) and (j <= high)) do
9       {
10              if (a[h] <=a[j]) then
11              {
12                      b[i]:=a[h];  h :=h + 1;
13              }
14              else
15              {
16                      b[i] =a[j];  j :=j + l;
17              }
18              i :=i +1;
19      }
20      if (h >mid) then
21              for k :=j to high do
22              {
23                      b[i] :=a[k]; i :=i +1;
24              }
25      else
26              for k :=h to mid do
27              {
28                      b[i] :=a[k]; i :=i +1;
29              }
30      for k:=low to high do a[k] :=b[k];
31}
```

Complexity:

Best case	Average case	Worst case
O (nlogn)	O (nlogn)	O (n^2)

PROGRAM:

```
#include<time.h>
#include<stdlib.h>
#include<iostream>
#define max 100
using namespace std;
void  mergesort(int[100],int,int);
void   merge(int[100],int,int,int);
int a[max];
int main()
{
        int i,n;
        clock_t s,e,z;
        s=clock();
        cout<<"Enter the no.of elements\n";
        cin>>n;
        cout<<"Elements of the array before sorting\n";
        for(i=0;i<n;i++)
        {
        a[i]=rand( )%1000;
        cout<<a[i]<<"\t";
        }
        mergesort(a,0,n-1);
        cout<<"\n Elements of the array after sorting\n";
        for(i=0;i<n;i++)
        cout<<a[i]<<"\t";
        e=clock();
        z=e-s;
        cout<<"\nthe time taken="<<z/CLOCKS_PER_SEC<<"seconds";
}
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;
        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)
        {
        for(k=j;k<=high;k++)
        {
        b[i]=a[k];
        i++;
        }
        }
        else
        {
        for(k=h;k <= mid;k++)
        {
        b[i]=a[k];
        i++;
        }
        }
        for(k=low;k<=high;k++)
        a[k]=b[k];
}
```

Program Number 3(a):

Obtain the Topological ordering of vertices in a given digraph.

Topological Ordering is based on decrease and conquer technique a vertex with no incoming edges is selected and deleted along 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 ordering.

1) Find the vertices whose in degree is zero and place them on to 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 in degree[v] gives the number of depended jobs of v is reduced by one.

```
1Algorithm Topological(a[],n,T[])
2// obtains the sequence of jobs to be executed resulting in topological order
3// a[]-adjacency matrix of the given graph
4// n-the number of vertices in the graph
5//T[]-indicates the jobs that are to be executed in the order
6{
7       for j:=0 to n-1 do
8       {
9               sum:=0;
10              for i:=0 to n-1 do
11              sum:=sum+a[i][j];12 top
:=-1;
13              for i:= 0 to n-1 do
14              {
15                      if(indegree [i]==0) then
16                      {
17                              top := top+1;
18                              s[top]:= i ;
19                      }
20              }
21              while(top!=1) do
22              {
23                      u:=s[top] ;
24                      top:=top-1;
25 Add u to solution vector T For each vertex v adjacent to u, decrement indegree [v] by one
```

```
26                      if(indegree [v]==0) then
27                      {
28                              top:=top+1;
29                              s[top]:=v;
30                      }
31              }
32      return T;
33}
```

Complexity: Complexity of topological sort is given by Θ [V+E].

PROGRAM:

```
#include<iostream>
using namespace std;
void findindegree(int [10][10],int[10],int);
void topological(int,int [10][10]);
int main()
{
        int a[10][10],i,j,n;
        cout<<"Enter the number of nodes:";
        cin>>n;
        cout<<"\nEnter the adjacency matrix\n";
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        cin>>a[i][j];
        cout<<"\nThe adjacency matirx is:\n";
        for(i=1;i<=n;i++)
        {
        for(j=1;j<=n;j++)
        {
        cout<<a[i][j]<<"\t";
        }
        cout<<"\n";
        }
        topological(n,a);
}
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;
        }
        }
        }
        }
        cout<<"\nTopological sequence is\n";
        for(i=1;i<=n;i++)
        cout<<t[i]<<"\t";
}
```

Program Number 3(b):

Compute the transitive closure of a given directed graph using 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($1<=i<=n$) and j^{th} column($1<=j<=n$) 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: $R^{(0)}$,….,$R^{(k-1)}$, $R^{(k)}$,….,$R^{(n)}$ 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):= A
        for  k :=1 to n do
        {
              for  i := 1 to n do
              {
                  for j := 1 to n do
                 {
                    R(k)[i,j] := R(k-1)[i,j] or R(k-1)[i,k] and R(k-1)[k,j]
                 }
              }
        }
        return R(n)
}
```

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

PROGRAM:

```
#include<iostream>
using namespace std;
void warshall(int[10][10],int);
int main()
{
	int a[10][10],i,j,n;
	cout<<"Enter the number of nodes:";
	cin>>n;
	cout<<"\nEnter the adjacency matrix:\n";
	for(i=1;i<=n;i++)
	for(j=1;j<=n;j++)
	cin>>a[i][j];
	cout<<"The adjacency matirx is:\n";
	for(i=1;i<=n;i++)
	{
	for(j=1;j<=n;j++)
	{
	cout<<a[i][j]<<"\t";
	}
	cout<<"\n";
	}
	warshall(a,n);
}
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;
	}
	}
	}
```

```
        }
        cout<<"\nThe path matrix is:\n";
        for(i=1;i<=n;i++)
        {
        for(j=1;j<=n;j++)
        {
        cout<<p[i][j]<<"\t";
        }
        cout<<"\n";
        }
}
```

Program Number 4:

Implement 0/1 Knapsack problem using Dynamic Programming.

Given n objects and a knapsack or bag. Object i has a weight w_i and the knapsack has a capacity m. If a fraction x_i, $0 < 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, the total required is the weight of all chosen objects to be at most m.

Dynamic programming approach: To design a dynamic programming algorithm, derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub instances. Consider an instance defined by the first i items, $1<=i<=n$, with weights $w_1,\ldots..,w_i$, values $v_1,\ldots\ldots v_i$ and knapsack capacity j, $1<=j<=W$. Let V [i,j] be the value of an optimal solution to this instance. Divide all the subsets in to two categories: those that include the i^{th} item and those that do not.

1) Among the subsets that do not include the i^{th} item, the value of an optimal subset is, V [i-1,j].

2) Among the subsets that do include the i^{th} item , an optimal subset is made up of this item and an optimal subset of the first i-1 items that fit into the knapsack of capacity $j-w_i$. The value of such a an optimal subset is $v_i + V[i-1,j-w_i]$.

Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values.

The goal is to find v[n,W] , the maximal value of subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

```
1Algorithm 0/1DKnapsack(v,w,n,W)
2//Input: set of items with weight wi and value vi; maximum capacity of knapsack W
3//Output: maximum value of subset with weight at most W.
6{
7       for w := 0 to W do
8       V[0,w] :=0;
9       for i :=1 to n do
10              for w:=0 to W do
11                if (w[i]<=w)
12                      V[i,w]=max{V[i-1,w],v[i]+V[i-1,w-w[i]]};
13                else
14                      V[i,w]=V[i-1,w];
15      return V[n,W];
16      }
```

Complexity: The Time efficiency and Space efficiency of 0/1 Knapsack algorithm is Θ(nW).

PROGRAM:

```
#include<iostream>
#define MAX 50
int p[MAX],w[MAX],n;
int knapsack(int,int);
int max(int,int);
using namespace std;
int main()
{
        int m,i,optsoln;
        cout<<"Enter no. of objects: ";
        cin>>n;
        cout<<"\nEnter the weights:\n";
        for(i=1;i<=n;i++)
        cin>>w[i];
        cout<<"\nEnter the profits:\n";
        for(i=1;i<=n;i++)
        cin>>p[i];
        cout<<"\nEnter the knapsack capacity:";
        cin>>m;
        optsoln=knapsack(1,m);
        cout<<"\nThe optimal soluntion is:"<<optsoln;
}
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;
}
```

Program Number 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.

```
1 Algorithm ShortestPaths(u,cost,dist,n)
2 // dist[j], 1<j <n, is set to the length of the shortest
3 // path from vertex v to vertex j in a digraph G with n
4 // vertices. dist[v] is set to zero. G is represented by its
5 // cost adjacency matrix cost [l : n, 1 : n].
6 {
7       for i: =1to n do
8       {// Initialize S.
9       S[i]:=false;  dist[i]:=cost [v,i];10     }
11      S[v]:=true;   dist[v]:=0.0;  // Put v in S.
12      for num: =2 to n do
13      {
14              // Determine n-1paths from v.
15               Choose u from among those vertices not
16               in S such that dist[u] is minimum;
17              S[u]:=true; // Put u in S.
18              for (each w adjacent to u with S[w] = false) do
19                      // Update distances.
20                      if {dist[w] > dist[u] + cost [u, w])) then
21       dist[w]:=dist[u] +cost [u, w];22      }
23}
```

Complexity: The Time complexity for Dijkstra's algorithm is O (|E| log |V|).

PROGRAM:

```
#include<iostream>
using namespace std;
void dij(int,int [20][20],int [20],int [20],int);
int main()
{
	int i,j,n,visited[20],source,cost[20][20],d[20];
	cout<<"Enter no. of vertices:";
	cin>>n;
	cout<<"Enter the cost adjacency matrix\n";
	for(i=1;i<=n;i++)
	{
	for(j=1;j<=n;j++)
	{
	cin>>cost[i][j];
	}
	}
	cout<<"\nEnter the source node:";
	cin>>source;
	dij(source,cost,visited,d,n);
	for(i=1;i<=n;i++)
	if(i!=source)
	cout<<"\nShortest path from"<<source<<"to"<<i<<"is"<<d[i];
	}
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;
        }
        }
        }
        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];
        }
        }
        }
}
```

Program Number 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 sub graph 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 sub graphs, 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 sub graph, it scans the sorted list adding the next edge on the list to the current sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

```
1 Algorithm Kruskal(E , cost, n ,t)
2 // E is the set of edges in G.G has n vertices. cost [u, v] is the
3 // cost of edge (u,v). t is the set of edges in the minimum-cost
4 // spanning tree. The final cost is returned.
5 {
6       Construct a heap out of the edge costs using Heapify;
7       for i :=1 to n do parent[i]:=-1;
8       // Each vertex is in a different set.
9       i :=0; mincost:=0.0;
10      while ((i < n-1) and (heap not empty)) do
11      {
12              Delete a minimum cost edge(u,v) from the heap
13              and reheapify using Adjust;
14              j :=Find(u); k :=Find(w);
15              if (j != k) then
16              {
17                      i: =i+ l;
18                       t [i, 1]:=u; t [i, 2]:=v;
19                      mincost:=mincost+ cost[u, v];
20                      Union(j,k);
21              }
22      }
23      if (i!=n -1) then write ("No spanning tree");
24      else return mincost;
25}
```

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

PROGRAM:

```
#include <iostream>
int ne=1,mincost=0,parent[20];
int find(int);
int uni(int,int);
using namespace std;
int main()
{
        int n,i,j,min,cost[20][20],a,b,u,v;
        cout<<"\nEnter the no. of vertices:";
        cin>>n;
        cout<<"\nEnter the cost adjacency matrix:\n";
        for(i=1;i<=n;i++)
        {
        for(j=1;j<=n;j++)
        {
        cin>>cost[i][j];
        if(cost[i][j]==0)
        cost[i][j]=999;
        }
        }
        cout<<"The edges of Minimum Cost 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;
        }
        }
        }
        u=find(u);
```

```
        v=find(v);
        if(uni(u,v))
        {
        cout<<ne++<<"edge ("<<a<<","<<b<<") ="<<min<<"\n";
        mincost +=min;
        }
        cost[a][b]=cost[b][a]=999;
        }
        cout<<"\n\t Minimum cost ="<<mincost<<"\n";
    }
    int find(int i)
    {
        while(parent[i])
        parent[i];
        return i;
    }
    int uni(int i,int j)
    {
        if(i!=j)
        {
        parent[j]=i;
        return 1;
        }
        return 0;
    }
```

Program Number 7(a):

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

Breadth First Search: In breadth first search start at a vertex v and mark it as having been reached (visited). The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been explored. The newly visited vertices haven't been explored and are put onto the end of a list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices operates as a queue and can be represented using any of the standard queue representation. 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.

```
1 Algorithm BFS(v)
2 //A breadth first search of G is carried out beginning
3 // at vertex v. For any node i, visited[i] = 1 if i has
4 // already been visited. The graph G and array visited [ ]
5 // are global; visited [ ] is initialized to zero.
6 {
7       u: =v; // q is a queue of unexplored vertices.
8       visited[v]:=1;
9       repeat
10      {
11              for all vertices w adjacent from u do
12              {
13                      if (visited[w] = 0) then
14                      {
15                      Add w to q; // w is unexplored.
16                      visited[w]:= 1;
17                      }
18              }
19              if q is empty then return; // No unexplored vertex.
20              Delete the nest element, u, from q;
21                                      // Get first unexplored vertex.
22      } until (false);
23}
```

Complexity: BFS has the time complexity as $\Theta(V^2)$ for Adjacency matrix representation and $\Theta(V+E)$ for Adjacency linked list representation, where V stands for vertices and E stands for edges.

PROGRAM:

```
#include<iostream>
using namespace std;
void BFS(int [20][20],int,int [20],int);
int main()
{
        int n,a[20][20],i,j,visited[20],source;
        cout<<"Enter the number of vertices:";
        cin>>n;
        cout<<"\nEnter the adjacency matrix:\n";
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        cin>>a[i][j];
        for(i=1;i<=n;i++)
        visited[i]=0;
        cout<<"\nEnter the source node:";
        cin>>source;
        visited[source]=1;
        BFS(a,source,visited,n);
        for(i=1;i<=n;i++)
        {
        if(visited[i]!=0)
        cout<<"\n Node "<<i<<" is reachable";
        else
        cout<<"\n Node "<<i<<" is not reachable";
        }
}
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;
        }
        }
        }
}
```

Program Number 7(b):

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

Depth First Search: A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of v continues. The search terminates when all reached vertices have been fully explored.
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.

```
1 Algorithm DFS(v)
2 // Given an undirected (directed) graph G = (V, E) with
3 // n vertices and an array visited [ ] initially set
4 //to zero, this algorithm visits all vertices
5 // reachable from v. G and visited [] are global.
6 {
7       visited[v]:=1;
8       for each vertex w adjacent from v do
9       {
10              if (visited[w] = 0) then DFS (w);
11      }
12}
```

Complexity: For the adjacency matrix representation, the traversal time efficiency is $\Theta(V^2)$ 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:

```
#include<iostream>
using namespace std;
void DFS(int [20][20],int,int [20],int);
int main()
{
	int n,a[20][20],i,j,visited[20],source;
	cout<<"Enter the number of vertices: ";
	cin>>n;
	cout<<"\nEnter the adjacency matrix:\n";
	for(i=1;i<=n;i++)
	for(j=1;j<=n;j++)
	cin>>a[i][j];
	for(i=1;i<=n;i++)
	visited[i]=0;
	cout<<"\nEnter the source node: ";
	cin>>source;
	DFS(a,source,visited,n);
	for(i=1;i<=n;i++)
	{
	if(visited[i]==0)
	{
	cout<<"\nGraph is not connected";
	exit(0);
	}
	}
	cout<<"\nGraph is connectd\n";
}
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);
	}
}
```

Program Number 8:

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.

Sum of Subsets: Given n distinct positive numbers, desired to find all combinations of these numbers whose sums are m. this is called the sum of subsets problem.Sum of Subset Problem is to 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. 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.

For example , for s={1,2,5,6,8} and d=9, there are two solutions : {1,2,6} and {1,8}.

```
1 Algorithm SumOfSub(s,k,r)
2 // Find all subsets of w [l: n] that sum to m. The values of x[j],
3 // 1<=j <k, have already been determined, s= ∑_{j=1}^{k-1} w[j] * x[j]
4 // and r = ∑_{j=k}^{n} w[j] .The w[j]'s are in non decreasing order.
5 // It is assumed that w[l] <= m and ∑_{i=1}^{n} w[i] ≥ m.
6 {
7       // Generate left child. Note: s+w[k] <= m since B_{k-1} is true.
8        x[k]:=1;
9       if (s+w[k] = m) then write (x[1 : k]);  // Subset found
10              // There is no recursive call here as w[j] > 0, 1<=j <=n.
11      else if (s+w[k] +w [k+1]<=m)
12              then SumOfSub (s+w[k], k+1, r-w[k]);
13      // Generate right child and evaluate B_k.
14       if ((s + r- w[k] > = m) and (s + w [k + 1] <= m)) then
15      {
16              x[k]:=0;
17              SumOfSub (s, k+l , r-w[k]);
18       }
19}
```

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

PROGRAM:

```
#include<iostream>
using namespace std;
void subset(int,int,int);
int x[10],w[10],d,count=0;
int main()
{
	int i,n,sum=0;
	cout<<"Enter the no. of elements:";
	cin>>n;
	cout<<"\nEnter the elements in ascending order:\n";
	for(i=0;i<n;i++)
	cin>>w[i];
	cout<<"\nEnter the sum:";
	cin>>d;
	for(i=0;i<n;i++)
	sum=sum+w[i];
	if(sum<d)
	{
	cout<<"No solution\n";
	}
	subset(0,0,sum);
	if(count==0)
	{
	cout<<"No solution\n";
	}
}
void subset(int cs,int k,int r)
{
	int i;
	x[k]=1;
	if(cs+w[k]==d)
	{
	cout<<"\n\nSubset"<<++count<<"\n";
	for(i=0;i<=k;i++)
	if(x[i]==1)
	cout<<w[i]<<"\t";
	}
	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]);
        }
}
```

Program Number 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. Let G = (V, E) be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $C_{ij} > 0$ for all i and j and $C_{ij} = \infty$ if <i,j> does not belong to E. Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem is to find a tour of minimum cost.

To solve the TSP using the Brute-Force approach, you must calculate the total number of routes and then draw and list all the possible routes. Calculate the distance of each route and then choose the shortest one—this is the optimal solution. This method breaks a problem to be solved into several sub-problems.

The Nearest Neighbor Algorithm is an approximate algorithm for finding a sub-optimal solution to the TSP. In this algorithm, it starts with a city as a starting city and repeatedly visits all the cities until all the cities have been visited exactly once.
Step 1: Choose an arbitrary city as start.
Step 2: Repeat the following operations until all the cities have been visited: go to the unvisited city nearest the one visited last.
Step 3: Return to the starting city.

PROGRAM:

```
#include<iostream>
```

```
using namespace std;
int s,c[100][100],ver;
float optimum=999,sum;
void swap(int v[], int i, int j)
{
        int t;
        t = v[i];
        v[i] = v[j];
        v[j] = t;
}
void brute_force(int v[], int n, int i)
{
        int j,sum1,k;
        if (i == n)
        {
        if(v[0]==s)
        {
        for (j=0; j<n; j++)
        cout<<v[j]<<"\t";
        sum1=0;
        for( k=0;k<n-1;k++)
        {
        sum1=sum1+c[v[k]][v[k+1]];
        }
        sum1=sum1+c[v[n-1]][s];
        cout<<" sum= "<<sum1<<"\n";
        if (sum1<optimum)
        optimum=sum1;
        }
        }
        else
        for (j=i; j<n; j++)
        {
        swap (v, i, j);
        brute_force (v, n, i+1);
        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];
}
int main ()
{
        int ver,v[100],i,j;
        cout<<"Enter n :";
        cin>>ver;
        for (i=0; i<ver; i++)
        v[i] = i+1;
        cout<<"Enter cost matrix\n";
        for(i=1;i<=ver;i++)
        for(j=1;j<=ver;j++)
        cin>>c[i][j];
        cout<<"\nEnter source :";
        cin>>s;
        brute_force (v, ver, 0);
        cout<<"\nOptimum solution with brute force technique
         is="<<optimum<<"\n";
        nearest_neighbour(ver);
        cout<<"\nSolution with nearest neighbour technique is="<<sum<<"\n";
        cout<<"\nThe approximation val is="<<((sum/optimum)-1)*100<<"%";
```

```
}
```

Program Number 10:

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

Prim's algorithm finds the minimum spanning tree for a weighted connected graph G=(V,E) to get an acyclic sub graph 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 sub tree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, 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.

```
1Algorithm Prim(E,cost,n,t)
2 // E is the set of edges in G. cost[1: n , 1: n] is the cost
3 // adjacency matrix of an n vertex graph such that cost [i,j] is
4 // either a positive real number or ∞ if no edge (i,j) exists.
5 //A minimum spanning tree is computed and stored as a set of
6 // edges in the array t [l : n-1,1: 2]  .(t[i,l],t[i,2]) is an edge in
7 //the minimum-cost spanning tree. The final cost is returned.
8 {
9         Let (k,l) be an edge of minimum cost in E;
10        mincost:= cost[k,l];
11        t[1,l]:=k;          t[l,2]:=l;
12        for i :=1 to n do // Initialize near.
13                if (cost[i,l] < cost[i,k]) then near[i]:=l;
14                else near[i]:=k;
15        near[k]:=near[l]:=0;
16        for i :=2 to n-1  do
17       {// Find n-2 additional edges for  t.
18         Let j be an index such that near[j] ≠ 0 and
19         cost[ j, near[j]] is minimum;
20         t[i,l]:=j; t[i,2]:=near[j];
21         mincost:=mincost+ cost[j, near[j]];
22         near[j]:=0;
23         for k :=1to n  do  // Update near[].
24                if ((near[k] ≠ 0) and(cost[k,near[k]] > cost[k,j]))
25                        then near[k]:=j;
26        }
27       return mincost;
28}
```

Complexity: The time complexity of Prim's algorithm will be in O (|E| log |V|).

PROGRAM:

```
#include<iostream>
using namespace std;
int ne=1,min_cost=0;
int main()
{
	int n,i,j,min,cost[20][20],a,u,b,v,source,visited[20];
	cout<<"Enter the no. of nodes:";
	cin>>n;
	cout<<"Enter the cost matrix:\n";
	for(i=1;i<=n;i++)
	{
	for(j=1;j<=n;j++)
	{
	cin>>cost[i][j];
	}
	}
	for(i=1;i<=n;i++)
	visited[i]=0;
	cout<<"Enter the root node:";
	cin>>source;
	visited[source]=1;
	cout<<"\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)
        {
        cout<<"\nEdge"<<ne++<<"\t("<<a<<"->"<<b<<")="<<min;
        min_cost=min_cost+min;
        visited[b]=1;
        }
        cost[a][b]=cost[b][a]=999;
        }
        cout<<"\nMinimum cost="<<min_cost<<"\n";
}
```

Program Number 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 is applicable to both directed and undirected graphs provided that they do not contain a cycle of negative length. 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 length of 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.

```
1Algorithm Floyd(W[1..n,1..n])
2//Implements Floyd's algorithm for the all-pairs shortest paths problem
3//Input: The weight matrix W of a graph
4//Output: The distance matrix of shortest paths length
5{
6       D: = W;
7       for k:=1 to n do
8       {
9               for i: = 1 to n do
10              {
11                      for j ← 1 to n do
12                      {
13                              D [i,j] ← min ( D[i, j], D[i, k]+D[k, j] )
14                      }
15              }
16      }
17      return D
18}
```

Complexity: The time efficiency of Floyd's algorithm is cubic i.e. Θ (n3).

PROGRAM:

```
#include<iostream>
using namespace std;
void floyd(int[10][10],int);
int min(int,int);
int main()
{
        int n,a[10][10],i,j;
```

```
        cout<<"Enter the no.of nodes :";
        cin>>n;
        cout<<"\nEnter the cost adjacency matrix\n";
        for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        cin>>a[i][j];
        floyd(a,n);
}
void floyd(int a[10][10],int n)
{
        int d[10][10],i,j,k;
        for(i=1;i<=n;i++)
        {
        for(j=1;j<=n;j++)
        d[i][j]=a[i][j];
        }
        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]);
        }
        }
        }
        cout<<"\nThe distance matrix is\n";
        for(i=1;i<=n;i++)
        {
        for(j=1;j<=n;j++)
        {
        cout<<d[i][j]<<"\t";
        }
        cout<<"\n";
        }
}
int min (int a,int b)
{
        if(a<b)
        return a;
```

```
        else
        return b;
}
```

Program Number 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 x n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. The queens are arranged in such a way that they do not threaten each other, according to the rules of the game of chess. Every queen on a chessboard square can reach the other square that is 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 diagonal lines. This problem can be solved by the backtracking technique. The concept behind backtracking algorithm used to solve this problem is to successively place the queens in columns. When it is impossible to place a queen in a column, the algorithm backtracks and adjusts a preceding queen.

```
1 Algorithm NQueens (k,n)
2 // Using backtracking, this procedure prints all
3 // possible placements of n queens on an n x n
4 // chessboard so that they are non-attacking.
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[l : n]);
12               else NQueens (k+l , 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: =1to k-1 do
8               if ((x[j] = i)  // Two in the same column
9                       or (Abs(x[j] - i) = Abs (j - k)))
10                              // or in the same diagonal
11                       then return false;
12       return true;
13}
```

Complexity: The time complexity of the N Queens problem is O (n!).

PROGRAM:

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

```
using namespace std;
void  nqueens(int);
int place(int[],int);
void printsolution(int,int[]);
int main()
{
        int n;
        cout<<"Enter the no.of queens:";
        cin>>n;
        nqueens(n);
}
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++;
        cout<<"\nSolution"<<count<<"\n";
        printsolution(n,x);
        }
        else
        {
        k++;
        x[k]=0;
        }
        }
        else
        {
        k--;
        }
        }
}
```

```
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++)
        {
        cout<<c[i][j]<<"\t";
        }
        cout<<"\n";
        }
}
```