DESIGN AND ANALYSIS OF ALGORITHMS LAB RECORD

(2023-2024)

DECCAN COLLEGE OF ENGINEERING &TECHNOLOGY

Darussalam, Hyderabad-500001

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

DESIGN AND ANALYSIS OF ALGORITHMS LAB RECORD

CERTIFICATE

This is to certify that Mr / Miss …………………………………………………….
Of B.E. VI Semester (AICTE), C.S.E branch bearing roll number 1603-21-733-……………… has successfully completed the Design and Analysis of Algorithms Lab work during the academic year 2023 – 2024.

Internal Examiner **External Examiner**

LIST OF LAB PROGRAMS

Program Number 1:

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

```
Algorithm BFS(v)
//A breadth first search of G is carried out beginning
// at vertex v. For any node i, visited[i] = 1if i has
// already been visited. The graph G and array visited []
// are global; visited []is initialized to zero.
{
    u: =v; // q is a queueof unexploredvertices.
    visited[v]:=1;
    repeat
    {
        for all vertices w adjacent from u do
        {
            if (visited[w]= 0) then
            {
            Add w to q; // w is unexplored.
            visited[w]:=1;
            }
            }
            if q is empty then return; // No unexplored vertex.
            Delete the nest element, u,from q;
                                    // Get first unexploredvertex.
        } until (false);
}
```

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

<u>Complexity:</u> BFS has the time complexity as Θ (V^2) for adjacency matrix representation and Θ (V+E) for Adjacency linked list representation, where V stands for vertices and E stands for edges.

<u>Complexity:</u> DFS has the time efficiency is Θ (V^2) and for the adjacency linked list representation, it is in Θ(V+E), where V and E are the number of graph's vertices and edges respectively.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
void BFS(int [20][20],int,int [20],int);
void DFS(int [20][20],int,int [20],int);

int main()
{
int n,a[20][20],i,j,visited[20],source;
printf(" REACHABLE NODES FROM STARTING NODES USING BFS AND GRAPH
IS CONNECTED OR NOT USING DFS\n\n");
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);
}
DFS(a,source,visited,n);
for(i=1;i<=n;i++)
{
if(visited[i]==0)
{
printf("\n Graph is not connected");
exit(0);
}
}
printf("\n Graph is connected\n");
return 0;
}

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;
}
}
}
}

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 2:

Sort a given set of elements using the Quick sort method and determine the time required to sort the elements.

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.

```
Algorithm Partition(a,m,p)
// Within a[m],a[m+1],... ,a[p-1]the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p-1,a[k] ≤ t for m ≤  k < g, and a[k] ≥ t
// for q <k <p. q is returned. Set a[p] = ∞.
{
    v:=a[m]; i:=m; j :=p;
    repeat
    {
        repeat
            i:=i+1;
        until(a[i]>v);
         repeat
            j: =j-1;
        until (a[j] <v);
        if (i <j) then     lnterchange(a,i,j);
    } until(i >= j);
     a[m] :=a[j];  a[j] :=v;  return j;
}
```

```
Algorithm Interchange (a,i,j)
// Exchange a[i] with a[j].
{
    p: =a[i];
     a[i]:=a[j];  a[j]:=p;

AlgorithmQuickSort(p, q)
```

```
// Sorts the elements a[p],..., a[q] which residein the global
// array a[l:n] into ascendingorder;a[n+1]is consideredto
//bedefinedand must be >= allthe elementsin a[l:n].
{
    if (p <q) then //If therearemorethan oneelement
    {
     // divide P into two sub problems.
          j :=Partition(a,p, q + 1);
          // j is the positionof the partitioningelement.
     // Solvethe subproblems.
          QuickSort(p,j-1);
          QuickSort(j+ l,q);
    // Thereis no needfor combining solutions.
    }
}
```

Complexity:

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

PROGRAM:

```
#include<time.h>
#include<stdio.h>
#define max 100
void qsort(int [],int,int);
int partition(int [],int,int);
int a[max];

int main()
{
int i,n;
clock_t s,e,z;
s=clock();
printf("QUICK SORT\n\n");
printf("Enter HOW many numbers you want to sort:\n");
scanf("%d",&n);
for(i=0;i<n;i++)
a[i]=rand()%100;
printf("\nThe array elements before sorting are:\n");
for(i=0;i<n;i++)
printf("%d \t",a[i]);
qsort(a,0,n-1);
printf("\nThe array elements after sorting are:\n");
for(i=0;i<n;i++)
printf("%d \t",a[i]);
e=clock();
z=e-s;
printf("\nTime taken: %d seconds",z/CLOCKS_PER_SEC);
}

void qsort(int a[100], 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 3:

Sort a given set of elements using the Merge sort method and determine the time required to sort the elements.

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.

```
Algorithm MergeSort(low,high)
// a[low :high]is a global array to be sorted.
// Small(P)is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low <high) then //If there are more than one element
    {
        // Divide P into sub problems.
            // Find where to split the set.
            mid:=[(low+high)/2];
        // Solve the subproblems.
         MergeSort(low, mid);
         MergeSort(mid+1 ,high);
         // Combine the solutions.
        Merge(low,mid,high);
     }
}
```

```
Algorithm Merge(low,mid,high)
//a[low:high] is a global array containing two sorted
//subsets in a[low:mid] and in a[mid+1:high].The goal
//is to merge these two sets into a single set residing
//in a[low :high]. b[ ]is an auxiliary global array.
{
    h :=low; i :=low; j :=mid+ 1;
    while ((h <=mid) and (j <= high)) do
    {
        if (a[h] <=a[j]) then
        {
            b[i]:=a[h];  h :=h + 1;
        }
        else
        {
            b[i] =a[j];  j :=j + l;
        }
        i :=i +1;
    }
    if (h >mid) then
        for k :=j to high do
        {
            b[i] :=a[k]; i :=i +1;
        }
    else
        for k :=h to mid do
        {
            b[i] :=a[k]; i :=i +1;
        }
    for k:=low to high do a[k] :=b[k];
}
```

Complexity:

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

PROGRAM:

```
#include<time.h>
#include<stdio.h>
#define max 100
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();
printf("MERGE SORT\n\n");
printf("Enter HOW many numbers you want to sort\n");
scanf("%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]);
}
mergesort(a,0,n-1);
printf("\nElements of the array after sorting\n");
for(i=0;i<n;i++)
printf("%d \t",a[i]);
e=clock();
z=e-s;
printf("\nThe time taken= %d seconds",z/CLOCKS_PER_SEC);
}

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 4:
Implement Traveling Salesperson problem using Brute Force.

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.

```
Algorithm TSPBruteforce( )
{
Choose a city to act as a starting point (let's say city1).
Find all possible routes using permutations. For n cities, it will be (n-1)!.
Calculate the cost of each route
return the one with the minimum cost.
}
```

Complexity: The time complexity of TSP using brute force technique is O (n!).

PROGRAM:

```
#include<stdio.h>
int source,c[100][100],n;
int cost=999,sum;
void tspbrute(int[], int, int);
void swap(int [],int,int);

int main ()
{
int n,v[100],i,j;
printf("TRAVELLING SALESPERSON PROBLEM - BRUTE FORCE\n\n");
printf("Enter the no. of vertices :");
scanf("%d",&n);
for (i=0; i<n; i++)
v[i] = i+1;
printf("Enter the adjacency cost matrix\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
scanf("%d",&c[i][j]);
printf("\nEnter the source vertex :");
scanf("%d",&source);
tspbrute(v, n, 0);
printf("\n Minimum cost=%d\n",cost);
}

void swap(int v[], int i, int j)
{
int t;
t = v[i];
v[i] = v[j];
v[j] = t;
}

void tspbrute(int v[], int n, int i)
{
int j,sum1,k;
if (i == n)
{
if(v[0]==source)
{
for (j=0; j<n; j++)
printf("%d \t",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]][source];
printf("Cost= %d\n",sum1);
if (sum1<cost)
cost=sum1;
}
}
else
for (j=i; j<n; j++)
{
swap (v, i, j);
tspbrute(v, n, i+1);
swap (v, i, j);
}
}
```

Program Number 5:
Implement Knapsack problem using Brute Force.

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved. If there are n items to choose from, then there will be 2^n possible combinations of items for the knapsack. An item is either chosen or not chosen. A bit string of 0's and 1's is generated which is of length n. If the ith symbol of a bit string is 0, then the ith item is not chosen and if it is 1, the ith item is chosen.

```
Algorithm BruteForce (Weights [1 … N], Values [1 … N], A[1…N])
//Finds the best possible combination of items for the KP
//Input: Array Weights contains the weights of all items
//Array Values contains the values of all items
//Array A initialized with 0s is used to generate the bit strings
//Output: Best possible combination of items in the knapsack bestChoice [1 .. N]
{
        for i := 1 to $2^n$ do
        j:=n ;
        tempWeight:= 0; tempValue :=0;
        while ( A[j] != 0 and j > 0)
        A[j] := 0;
        j :=j -1;
        A[j]:=1;
        for k :=1 to n do
        if (A[k] := 1) then
        tempWeight := tempWeight + Weights[k];
        tempValue :=tempValue + Values[k];
        if ((tempValue > bestValue) AND (tempWeight <= Capacity)) then
        bestValue :=tempValue;
        bestWeight := tempWeight;
        bestChoice := A;
        return bestChoice;
}
```

Complexity: The time complexity of the Brute Force Knapsack Problem is O ($n2^n$).

PROGRAM:

```
#include<stdio.h>
int max(int,int);
int knapsackbrute(int,int [],int [],int);

int main()
{
int profit[] = { 42, 12, 40,25 };
int weight[] = { 7, 3, 4,5 };
int m = 10,i;
int n = sizeof(profit) / sizeof(profit[0]);
printf("KNAPSACK USING BRUTE FORCE\n\n");
printf("Profits of objects are:\n");
for(i=0;i<n;i++)
printf("%d\t",profit[i]);
printf("\n\n Weights of objects are:\n");
for(i=0;i<n;i++)
printf("%d\t",weight[i]);
printf("\n\n Optimal Solution is:");
printf("%d",knapsackbrute(m, weight, profit, n));
return 0;
}

int max(int a, int b)
{
return (a > b) ? a : b;
}

int knapsackbrute(int m, int w[], int p[], int n)
{
if (n == 0 || m == 0)
return 0;
if (w[n - 1] > m)
return knapsackbrute(m, w, p, n - 1);
else
return max(p[n - 1]+ knapsackbrute(m - w[n - 1],w,p,n - 1),knapsackbrute(m,w,p,n - 1));
}
```

Program Number 6 :
Implement 0/1 Knapsack problem using Greedy Method.

In this method, the Knapsack's filling is done so that the maximum capacity of the knapsack is utilized so that maximum profit can be earned from it. The knapsack problem using the Greedy Method is referred to as: Given a list of n objects, say {I_1, I_2,……, I_n) and a knapsack (or bag). The capacity of the knapsack is M. Each object I_j has a weight w_j and a profit of p_j If a fraction x_j (where $x \in \{0...., 1)$) of an object I_j is placed into a knapsack, then a profit of p_jx_j is earned. The problem (or Objective) is to fill the knapsack (up to its maximum capacity M), maximizing the total profit earned.

```
1Algorithm KnapsackGreedy ( )
2{
3       Consider all the items with their weights and profits mentioned respectively.
4       Calculate $P_i/W_i$ of all the items and sort the items in descending order based on their
        $P_i/W_i$ values.
5   Without exceeding the limit, add the items into the knapsack.
6   If the knapsack can still store some weight, but the weights of other items exceed the limit,
    the fractional part of the next time can be added.
7}
```

Complexity: The Time complexity of Knapsack algorithm is O (N log N).

PROGRAM:

```
# include<stdio.h>
void knapsack(int,float[ ],float [ ],float);

int main()
{
float weight[20], profit[20], capacity;
int num, i, j;
float ratio[20], temp;
printf("KNAPSACK USING GREEDY METHOD\n\n");
printf("\nEnter the no. of objects:\n");
scanf("%d", &num);
printf("\nEnter the weights of each object:\n ");
for (i = 0; i < num; i++)
{
scanf("%f", &weight[i]);
}
printf("\nEnter the profits of each object:\n ");
for (i = 0; i < num; i++)
{
scanf("%f", &profit[i]);
}
printf("\nEnter the capacity of knapsack:\n");
scanf("%f", &capacity);
for (i = 0; i < num; i++)
{
ratio[i] = profit[i] / weight[i];
}
for (i = 0; i < num; i++)
{
for (j = i + 1; j < num; j++)
{
if (ratio[i] < ratio[j])
{
temp = ratio[j];
ratio[j] = ratio[i];
ratio[i] = temp;
temp = weight[j];
weight[j] = weight[i];
weight[i] = temp;
temp = profit[j];
profit[j] = profit[i];
profit[i] = temp;
}
}
}
knapsack(num, weight, profit, capacity);
return(0);
}

void knapsack(int n, float weight[], float profit[], float capacity)
{
float x[20], tp = 0;
int i, j, u;
u = capacity;
for (i = 0; i < n; i++)
x[i] = 0.0;
for (i = 0; i < n; i++)
{
if (weight[i] > u)
break;
else
{
x[i] = 1.0;
tp = tp + profit[i];
u = u - weight[i];
}
}
if (i < n)
x[i] = u / weight[i];
tp = tp + (x[i] * profit[i]);
printf("\nMaximum profit is: %f", tp);
}
```

Program Number 7:
Find Minimum Cost Spanning Tree of a given undirected graph using 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.

```
Algorithm Kruskal(E,cost, n ,t)
// E is the set of edgesin G.G has n vertices. cost [u, v] is the
cost of edge (u,v). t is the set of edges in the minimum-cost
// spanning tree. The final cost is returned.
{
    Construct a heap out of the edge costs using Heapify;
    for i :=1to n do parent[i]:=-1;
     // Each vertex is in a different set.
    i :=0;mincost:=0.0;
    while ((i < n-1) and (heap not empty)) do
    {
        Delete a minimum cost edge (u,v) from the heap
        and reheapify using Adjust;
        j :=Find(u); k :=Find(w);
        if (j != k) then
        {
            i: =i+ l;
             t [i, 1]:=u; t [i, 2]:=v;
            mincost:=mincost+ cost[u, v];
            Union(j,k);
        }
    }
    if (i!=n -1)then write ("No spanning tree”);
    else return mincost;
}
```

<u>Complexity:</u> 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,mincost=0,parent[20];
int find(int);
int uni(int,int);

int main()
{
int n,i,j,min,cost[20][20],a,b,u,v;
printf("KRUSKALS ALGORITHM \n\n");
printf("\nEnter the no. of vertices:");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix:\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
}
printf("\n 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))
{
printf("\n Edge:%d %d --> %d Cost: %d",ne++,a,b,min);
mincost +=min;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n\t Minimum cost = %d",mincost);
}

int find(int i)
{
while(parent[i])
i=parent[i];
return i;
}

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

Program Number 8:

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.

```
Algorithm Prim (E,cost,n,t)
// E is the set of edges in G. cost[1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i,j]is
// either a positive real number or ∞ if no edge (i,j)exists.
//A minimum spanning tree is computed and stored as a set of
// edges in the array t [l:n-1,1:2].(t[i,l],t[i,2])is an edge in
//the  minimum-cost spanning tree. The final cost is returned.
{
     Let (k,l)be an edge of minimum cost in E;
    mincost:= cost[k,l];
     t[1,l]:=k;        t[l,2]:=l;
    for i :=1to n do // Initialize near.
            if (cost[i,l] <cost[i,k])then near[i]:=l;
            else near[i]:=k;
     near[k]:=near[l]:=0;
    for i :=2 to n-1do
    {// Find n-2 additional edges for t.
    Let j bean index such that near[j] ≠ 0 and
      cost[ j, near[j]] is minimum;
    t[i,l]:=j; t[i,2]:=near[j];
      mincost:=mincost+ cost[j, near[j]];
    near[j]:=0;
    for k :=1to n do// Update near[].
            if ((near[k] ≠ 0) and(cost[k,near[k]]>cost[k,j]))
                    then near[k]:=j;
     }
    return mincost;
}
```

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

PROGRAM:

```
#include<stdio.h>
int a,b,u,v,n,i,j,ne=1;
int visited[10]={0},mincost=0,cost[10][10];

int main()
{
int min;
printf("PRIMS ALGORITHM \n\n");
printf("\nEnter the number of verices:");
scanf("%d",&n);
printf("\nEnter the cost adjacency matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j]==0)
cost[i][j]=999;
}
visited[1]=1;
printf("\n 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)
if(visited[i]!=0)
{
min=cost[i][j];
a=u=i;
b=v=j;
}
if(visited[u]==0 || visited[v]==0)
{
printf("\n Edge: %d %d --> %d cost: %d",ne++,a,b,min);
mincost+=min;
visited[b]=1;
}
cost[a][b]=cost[b][a]=999;
}
printf("\n \t Minimum cost=%d",mincost);
}
```

Program Number 9:

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 ShortestPaths (u,cost,dist,n)
// dist[j], 1<j <n, is set to the length of the shortest
// path from vertex v to vertex j in a digraph G with n
// vertices. dist[v] is set to zero. G is represented by its
// cost adjacency matrixcost [l:n, 1:n].
{
    for i: =1 to n do
    {// Initialize S.
        S[i]:=false;   dist[i]:=cost [v,i];
    }
    S[v]:=true; dist[v]:=0.0;   // Put v in S.
    for num: =2 to n do
    {
        // Determinen-1pathsfrom v.
         Choose u from among those vertices not
         in S such that dist[u] is minimum;
        S[u]:=true;// Put u in S.
        for (each w adjacent to u with S[w]= false) do
            // Update distances.
            if {dist[w] >dist[u]+cost [u, w])) then
                dist[w]:=dist[u] +cost [u, w];
     }
}
```

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

PROGRAM:

```
#include<stdio.h>
#define infinity 999
void dij(int,int,int[20][20],int[20]);

int main()
{
int n,v,i,j,cost[20][20],dist[20];
printf("DIJKSTRAS ALGORITHM \n \n");
printf("Enter the number of vertices:\n");
scanf("%d",&n);
printf("\nEnter the cost Adjacency Matrix:\n");
for(i=1;i<=n;i++)
for(j=1;j<=n;j++)
{
scanf("%d",&cost[i][j]);
if(cost[i][j] == 0)
cost[i][j]=infinity;
}
printf("\nEnter the Source Vertex:");
scanf("%d",&v);
dij(n,v,cost,dist);
printf("\nShortest paths : \n");
for(i=1;i<=n;i++)
if(i!=v)
printf("\nFrom %d to %d is %d",v,i,dist[i]);
}

void dij(int n, int v,int cost[20][20], int dist[])
{
int i,u,count,w,flag[20],min;
for(i=1;i<=n;i++)
flag[i]=0, dist[i]=cost[v][i];
count=2;
while(count<=n)
{
min=99;
for(w=1;w<=n;w++)
if(dist[w]<min && !flag[w])
{
min=dist[w];
u=w;
}
flag[u]=1;
count++;
for(w=1;w<=n;w++)
if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
dist[w]=dist[u]+cost[u][w];
}
}
```

Program Number 10:
Implement Knapsack problem using Dynamic Programming.

Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible. The knapsack problem is useful in solving resource allocation problem. Let X = < x_1, x_2, x_3, ... , x_n> be the set of n items. Sets W = <w_1, w_2, w_3, . . . , w_n> and V = < v_1, v_2, v_3, . . . , v_n> are weight and value associated with each item in X. Knapsack capacity is M unit. The knapsack problem is to find the set of items which maximizes the profit such that collective weight of selected items does not cross the knapsack capacity. Select items from X and fill the knapsack such that it would maximize the profit. Knapsack problem has two variations. 0/1 knapsack, that does not allow breaking of items. Either add an entire item or reject it. It is also known as a binary knapsack. Fractional knapsack allows breaking of items. Profit will be earned proportionally.

```
1 Algorithm DynamicKnap( )
2{
3 Set $S^0 = \{(0, 0)\}$
4 $S_1^i = \{(p, w) \mid (p - p_i) \in S^i, (w - w_i) \in S^i \}$
5 // Obtain $S^{i+1}$ by invoking $x_{i+1}$
6//If $x_i = 0$ (item $x_i$ is excluded) then $S_1^i = S^i$
7   //If $x_i = 1$ (item $x_i$ is included) then $S_1^i$ is computed by adding $(p_{i+1}, w_{i+1})$ in each state of $S^i$.
8   $S^{i+1}$ = MERGE_PURGE($S^i$, $S_1^i$).
9   // MERGE_PURGE does following:
10  // For two pairs $(p_x, w_x) \in S^{i+1}$ and $(p_y, w_y) \in S^{i+1}$, if $p_x \le p_y$ and $w_x \ge w_y$, we say that $(p_x, w_x)$ is
11  // dominated by $(p_y, w_y)$. And the pair $(p_x, w_x)$ is discarded.It also purges all the pairs (p, w) from $S^{i+1}$ if
12  //w > M, i.e. it weight exceeds knapsack capacity.
13  Repeat step 8 n times
14  $f_n(M) = S^n$. This will find the solution of  KNAPSACK(1, n, M) for each pair $(p, w) \in S^n$
15  if $(p, w) \in S^{n-1}$, then set $x_n = 0$
16  if $(p, w) \notin S^{n-1}$, then set $x_n = 1$, update $p = p - x_n$ and $w = w - w_n$
```

Complexity: The Time complexity of Knapsack algorithm is O (Nw).

PROGRAM:

```
#include<stdio.h>
int max(int,int);
int knapSack(int, int [],int [],int);

int main()
{
int i, n, val[20], wt[20], W;
printf("KNAPSACK USING DYNAMIC PROGRAMMING:\n\n");
printf("Enter number of items:");
scanf("%d", &n);
printf("Enter the weights of %d items:\n",n);
for(i = 0;i < n; ++i)
{
scanf("%d", &wt[i]);
}
printf("Enter the profits of %d items:\n",n);
for(i = 0;i < n; ++i)
{
scanf("%d", &val[i]);
}
printf("Enter the knapsack capacity:");
scanf("%d", &W);
printf("%d", knapSack(W, wt, val, n));
}

int max(int a, int b)
{
return (a > b)? a : b;
}

int knapSack(int W, int wt[], int val[], int n)
{
int i, w;
int K[n+1][W+1];
for (i = 0; i <= n; i++)
{
for (w = 0; w <= W; w++)
{
if (i==0 || w==0)
K[i][w] = 0;
else if (wt[i-1] <= w)
K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
else
K[i][w] = K[i-1][w];
}
}
return K[n][W];
}
```

Program Number 11:
Implement All-Pairs Shortest Paths Problem using Floyd's algorithm.

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.

```
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. Θ (n3).

PROGRAM:

```
#include<stdio.h>
void floyd(int[10][10],int);
int min(int,int);

int main()
{
int n,a[10][10],i,j;
printf("FLOYD WARSHALL ALGORITHM\n \n");
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);
}

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]);
}
}
}
printf("\nThe distance matrix is\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n;j++)
{
if (d[i][j] == 999)
printf(" ");
else
printf("%d \t",d[i][j]);
}
printf("\n");
}
}

int min (int a,int b)
{
if(a<b)
return a;
else
return b;
}
```

Program Number 12:
Implement Traveling Salesperson problem using Dynamic Programming.

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.
Dynamic Programming: Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output. For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once. Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1. Finally, we return the minimum of all [cost(i) + dist(i, 1)] values.

```
1Algorithm TSPDynamic( )
2{
3 A graph G {V, E} as an input and declare another graph as the output (say G') which will
record the path 4 the salesman is going to take from one node to another.
5 Sort all the edges in the input graph G from the least distance to the largest distance.
6 The first edge selected is the edge with least distance, and one of the two vertices (say A and
B) being the 7 origin node (say A).
8 Then among the adjacent edges of the node other than the origin node (B), find the least cost
edge and add  9 it onto the output graph.
10 Continue the process with further nodes making sure there are no cycles in the output graph
and the path 11 reaches back to the origin node A.
12}
```

Complexity: The time complexity of TSP using **Dynamic Programming is O ($n^2 2^n$).**

PROGRAM:

```
#include<stdio.h>
int ary[10][10],visited[10],n,cost=0;
void takeinput();
int findnextnode(int);
void tsp(int);

int main()
{
takeinput();
printf("the path is:\n");
tsp(0);
printf("\n\n Minimum cost is %d",cost);
return 0;
}

void takeinput()
{
int i,j;
printf("TRAVELLING SALEPERSON PROBLEM - DYNAMIC PROGRAMMING
\n\n");
printf("Enter the number of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency cost Matrix\n");
for(i=0;i<n;i++)
{
for( j=0;j<n;j++)
scanf("%d",&ary[i][j]);
visited[i]=0;
}
}

int findnextnode(int node)
{
int nd=999,min=999,minind,i;
for(i=0;i<n;i++)
{
if((ary[node][i]!=0)&&(visited[i]==0))
if(ary[node][i]+ary[i][node]<min)
{
min=ary[i][0]+ary[node][i];
minind=ary[node][i];
nd=i;
}
}
if(min!=999)
cost+=minind;
return nd;
}

void tsp(int node)
{
int i,nextnode;
visited[node]=1;
printf("%d--> ",node+1);
nextnode=findnextnode(node);
if (nextnode==999)
{
nextnode=0;
printf("%d",nextnode+1);
cost+=ary[node][nextnode];
return;
}
tsp(nextnode);
}
```

Program Number 13 :
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.

```
Algorithm NQueens (k,n)
// Usingbacktracking,this procedureprints all
// possibleplacementsof n queenson an n x n
// chessboardsothat they arenon-attacking.
{
    for i: =1to n do
    {
        if Place (k,i)then
        {
            x[k]:=i;
            if (k = n) then write (x[l:n]);
            else NQueens (k+l,n);
        }
    }
}
```

```
Algorithm Place (k,i)
// Returnstrueif a queencan beplacedin kth row and
// ith column.Otherwiseit returnsfalse.x []is a
// globalarray whose first (k-1) values have beenset.
// Abs(r) returnsthe absolutevalue of r.
{
    for j: =1to k-1do
        if ((x[j] = i)  // Two in the samecolumn
            or (Abs(x[j] -i) = Abs (j- k)))
                // orin the samediagonal
            thenreturnfalse;
    return true;
}
```

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

PROGRAM:

```
#include<stdio.h>
#include<math.h>
void nqueens(int);
int place(int[],int);
void printsolution(int,int[]);

int main()
{
int n;
printf("N-QUEENS PROBLEM-BACKTRACKING\n\n");
printf("Enter the no.of queens:");
scanf("%d",&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++;
printf("\n Solution %d \n",count);
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++)
{
printf("%c \t",c[i][j]);
}
printf("\n");
}
}
```

<u>Program Number 14 :</u>

<u>Implement Graph coloring problem using Back Tracking</u>.

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the vertex coloring problem. If coloring is done using at most k colors, it is called k-coloring. The smallest number of colors required for coloring graph is called its chromatic number.
In this approach, we color a single vertex and then move to its adjacent (connected) vertex to color it with different color. After coloring, we again move to another adjacent vertex that is uncolored and repeat the process until all vertices of the given graph are colored. In case, we find a vertex that has all adjacent vertices colored and no color is left to make it color different, we backtrack and change the color of the last colored vertices and again proceed further. If by backtracking, we come back to the same vertex from where we started and all colors were tried on it, then it means the given number of colors (i.e. ‘m’) is insufficient to color the given graph and we require more colors (i.e. a bigger chromatic number).

```
1Algorithm Graphcoloring( )
2{
3 Confirm whether it is valid to color the current vertex with the current color (by checking
whether 4 4 any of its adjacent vertices are colored with the same color).
5 If yes then color it and otherwise try a different color.
6 Check if all vertices are colored or not.
7 if not then move to the next adjacent uncolored vertex.
8  If no other color is available then backtrack (i.e. un-color last colored vertex).
9}
```

<u>Complexity:</u> The time complexity of the Graph coloring problem is $O(m^V)$.

PROGRAM:

```
#include<stdio.h>
static int m, n, c=0,count=0;
int g[50][50],x[50];
void nextValue(int k);
void GraphColoring(int k);

int main()
{
int i, j,temp;
printf("GRAPH COLORING-BACKTRACKING\n\n");
printf("\nEnter the number of nodes: " );
scanf("%d", &n);
printf("\nIf edge exists then enter 1 else enter 0 \n");
for(i=1; i<=n; i++)
{
x[i]=0;
for(j=1; j<=n; j++)
{
if(i==j)
g[i][j]=0;
else
{
printf("%d -> %d: " , i, j);
scanf("%d", &temp);
g[i][j]=g[j][i]=temp;
}
}
}
printf("\nPossible Solutions are\n");
for(m=1;m<=n;m++)
{
if(c==1)
{
break;
}
GraphColoring(1);
}
printf("\nThe chromatic number is %d", m-1);
printf("\nThe total number of solutions is %d", count);
}

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

void nextValue(int k)
{
int j;
while(1)
{
x[k]=(x[k]+1)%(m+1);
if(x[k]==0)
{
return;
}
for(j=1;j<=n;j++)
{
if(g[k][j]==1&&x[k]==x[j])
break;
}
if(j==(n+1))
{
return;
}
}
}
```

Program Number 15 :
Implement Hamiltonian Cycle using Back Tracking.

A Hamiltonian graph is a connected graph that contains a Hamiltonian cycle/circuit. Hamiltonian cycle is a path that visits each and every vertex exactly once and goes back to starting vertex.
The backtracking solution vector (X_1, ...,X_n) is defined so that x_i represents the ith visited vertex of the proposed cycle. Now we have to find how to compute the set of possible vertices for X_k if X_1, ..., X_{k-1}, have already been chosen. If k = 1 the x, can be any of the n vertices. To avoid printing the same cycle n times, we require that xy = 1. If 1 <k<n, then xk can be any vertex v that is distinct from X_1, X_2, ...,X_{k-1} and v is connected by an edge to X_{k-1}. Only the vertex X_n be the one remaining vertex and it must be connected to both Xn-1 and X1. Algorithm 1 NextValue (k) determines a possible next vertex of the proposed cycle.
Backtracking Approach: In this approach, we start from the vertex 0 and add it as the starting of the cycle. Now, for the next node to be added after 0, we try all the nodes except 0 which are adjacent to 0, and recursively repeat the procedure for each added node until all nodes are covered where we check whether the last node is adjacent to the first or not if it is adjacent to the first we declare it to be a Hamiltonian path else we reject this configuration.

```
Algorithm NextValue (k)
// x [1 : k-1] is a path of k – 1 distinct vertices. If
ll x [k] = 0, then no vertex has as yet been assigned to x [k]. After
// execution, x[k] is assigned to the next highest numbered vertex
// which does not already appear in x [1 : k-1] and is connected by
// an edge to x [k - 1]. Otherwise x [k] = 0. If k= n, then
// in addition x [k] is connected to x [1].
{
repeat
{
    x[k] : = (x [k] + 1) mod (n+1); //Next vertex.
    if (x [k] = 0) then return;
    if (G[x [k – 1], x [k]] + 0) then
    {// Is there an edge ?
    for j := 1 to k-1 do if (x [j] = x[k]) then break;
                // Check for distinctness
    if (j = k) then // If true, then the vertex is distinct.
                        if ((k <n) or ((k = n) and G [x [n], x [1]] + 0))
                            then return;
    }
} until (false);
}
```

```
Algorithm Hamiltonian (k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1:n). All cycles begin at node 1.
{
repeat
        { // Generate values for x [k].
  NextValue (k); // Assign a legal next value to x[k].
  if (x [k] = 0) then return;
  if (k = n) then write (x [1 : n]);
  else Hamiltonian (k+ 1);
  } until (False);
}
```

Complexity: The time complexity of the Hamiltonian cycle is O(N!), where N is the number of vertices.

PROGRAM:

```
#include<stdio.h>
int graph[10][10];
int path[100];
int NODE;
void displayCycle();
bool isValid(int, int);
bool cycleFound(int);
bool hamiltonianCycle();

int main()
{
int i,j;
printf("HAMILTONIAN CYCLE-BACKTRACKING\n\n");
printf("Enter the number of nodes in the graph:\n");
scanf("%d",&NODE);
printf("Enter the Adjacency Matrix :\n");
for (i=0;i<NODE;i++)
{
for (j=0;j<NODE;j++)
{
scanf("%d",&graph[i][j]);
}
}
printf("\n");
hamiltonianCycle();
}

void displayCycle()
{
printf("The Hamiltonian Cycle :\n");
for (int i = 0; i < NODE; i++)
printf("%d ",path[i]);
printf("%d",path[0]);
}

bool isValid(int v, int k)
{
if (graph [path[k-1]][v] == 0)
return false;
for (int i = 0; i < k; i++)
if (path[i] == v)
return false;
return true;
}
```

```
bool cycleFound(int k)
{
if (k == NODE)
{
if (graph[path[k-1]][ path[0] ] == 1 )
return true;
else
return false;
}

for (int v = 1; v < NODE; v++)
{
if (isValid(v,k))
{
path[k] = v;
if (cycleFound (k+1) == true)
return true;
path[k] = -1;
}
}
return false;
}

bool hamiltonianCycle()
{
for (int i = 0; i < NODE; i++)
path[i] = -1;
path[0] = 0;
if ( cycleFound(1) == false )
{
printf("Solution does not exist \n");
return false;
}
displayCycle();
return true;
}
```

LAB VIVA QUESTIONS

1. Explain is divide and conquer.
2. What is the average case time complexity of quick sort.
3. List different ways of selecting pivot element.
4. What is the running time of merge sort?
5. What technique is used to sort elements in merge sort?
6. Define Brute Force Method with an example.
7. Define Knapsack problem.
8. Define the Greedy Method.
9. What do you understand by Dynamic Programming?
10. Define the principle of optimality.
11. What is the optimal solution for knapsack problem?
12. What is the time complexity of knapsack problem?
13. What is the difference between Greedy knapsack and Fractional Knapsack?
14. What is the time complexity of Dijkstra's algorithm?
15. Define cost matrix.
16. Define directed graph.
17. Define connected graph.
18. What is the time complexity of Kruskal's algorithm?
19. Explain Prims Algorithm.
20. What is time complexity of Prim's algorithm.
21. what is the difference between Prims and Kruskals?
22. Define Distance Matrix.
23. Define spanning tree.
24. Define minimum cost spanning tree.
25. Define graph and its types.
26. List the different graph traversals.
27. Explain DFS traversal.
28. Explain BFS traversal.
29. What is the difference between DFS and BFS?
30. What are the time complexities of BFS and DFS algorithms?
31. Define is Back-Tracking.
32. Define Optimal Solution.
33. What is Floyd's algorithm?
34. What is the time complexity of Floyd's algorithm?
35. Explain Travelling Sales Person Problem.
36. What is the time complexity of Travelling Sales Person Problem?
37. Define a live node.
38. Define a dead node.
39. Define implicit and explicit constraints in backtracking.
40. What is the N-Queens problem.
41. What is the time complexity of n-queens problem.
42. What is Graph coloring with an example?
43. Define Hamiltonian cycle with an example.
44. what is time complexity of Graph coloring problem?
45. what is the time complexity of Hamiltonian cycle?