1. **Write a C program to input a graph and perform traversal using Depth First Search (DFS) on it.**

Algorithm::

i. DFS(G, u)
ii. u.visited = true
iii. for each v ∈ G.Adj[u]
iv. if v.visited == false
v. DFS(G,v)

vi. init() {
vii. For each u ∈ G
viii. u.visited = false
ix. For each u ∈ G
x. DFS(G, u)
xi. }

Source code::

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

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int v);
```

```
struct Graph {
  int numVertices;
  int* visited;

  struct node** adjLists;
};

void DFS(struct Graph* graph, int vertex) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  graph->visited[vertex] = 1;
  printf("Visited %d \n", vertex);

  while (temp != NULL) {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
    temp = temp->next;
  }
}

struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
```

```
}

struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

void printGraph(struct Graph* graph) {
```

```
  int v;
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}

int main() {
  struct Graph* graph = createGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 3);

  printGraph(graph);

  DFS(graph, 2);

  return 0;
}
```

Output::

```
Adjacency list of vertex 0
2 -> 1 ->

Adjacency list of vertex 1
2 -> 0 ->

Adjacency list of vertex 2
3 -> 1 -> 0 ->

Adjacency list of vertex 3
2 ->
Visited 2
Visited 3
Visited 1
Visited 0
```

2. **Write a C program to input a graph and perform traversal using Breadth First Search (BFS) on it. Find the distance of the starting vertex from another specific vertex given by user.**

Algorithm::

1. create a queue Q

2. mark v as visited and put v into Q

3. while Q is non-empty

4. remove the head u of Q

5. mark and enqueue all (unvisited) neighbours of u

SOURCE CODE::

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue {
  int items[SIZE];
  int front;
  int rear;
};

struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int);
```

```
struct Graph {
  int numVertices;
  struct node** adjLists;
  int* visited;
};

void bfs(struct Graph* graph, int startVertex) {
  struct queue* q = createQueue();

  graph->visited[startVertex] = 1;
  enqueue(q, startVertex);

  while (!isEmpty(q)) {
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);

    struct node* temp = graph->adjLists[currentVertex];

    while (temp) {
      int adjVertex = temp->vertex;

      if (graph->visited[adjVertex] == 0) {
        graph->visited[adjVertex] = 1;
        enqueue(q, adjVertex);
      }
      temp = temp->next;
    }
```

```
  }
}

struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));
  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }

  return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
```

```
  graph->adjLists[src] = newNode;

  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

struct queue* createQueue() {
  struct queue* q = malloc(sizeof(struct queue));
  q->front = -1;
  q->rear = -1;
  return q;
}

int isEmpty(struct queue* q) {
  if (q->rear == -1)
    return 1;
  else
    return 0;
}

void enqueue(struct queue* q, int value) {
  if (q->rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (q->front == -1)
      q->front = 0;
    q->rear++;
    q->items[q->rear] = value;
```

```
  }
}

int dequeue(struct queue* q) {
  int item;
  if (isEmpty(q)) {
    printf("Queue is empty");
    item = -1;
  } else {
    item = q->items[q->front];
    q->front++;
    if (q->front > q->rear) {
      printf("Resetting queue ");
      q->front = q->rear = -1;
    }
  }
  return item;
}

void printQueue(struct queue* q) {
  int i = q->front;

  if (isEmpty(q)) {
    printf("Queue is empty");
  } else {
    printf("\nQueue contains \n");
    for (i = q->front; i < q->rear + 1; i++) {
      printf("%d ", q->items[i]);
    }
```

```
  }
}

int main() {
  struct Graph* graph = createGraph(6);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 1, 4);
  addEdge(graph, 1, 3);
  addEdge(graph, 2, 4);
  addEdge(graph, 3, 4);

  bfs(graph, 0);

  return 0;
}
```

Output::

```
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3
```

3. **Write a C program to input a connected weighted graph and use Kruskal's algorithm to find the cost of the minimum spanning tree (MST), along with the edges of the MST.**

ALGORITHM::

1. Begin
2. Create the edge list of given graph, with their weights.
3. Sort the edge list according to their weights in ascending order.
4. Draw all the nodes to create skeleton for spanning tree.
5. Pick up the edge at the top of the edge list (i.e. edge with minimum weight).

6. Remove this edge from the edge list.

7. Connect the vertices in the skeleton with given edge. If by connecting the vertices, a cycle is created in the skeleton, then discard this edge.

8. Repeat steps 5 to 7, until n-1 edges are added or list of edges is over.

9. Return

SOURCE CODE::

```
#include<stdio.h>

#define MAX 30

typedef struct edge
{
	int u,v,w;
}edge;

typedef struct edgelist
{
	edge data[MAX];
	int n;
}edgelist;

edgelist elist;

int G[MAX][MAX],n;
edgelist spanlist;
```

```
void kruskal();
int find(int belongs[],int vertexno);
void union1(int belongs[],int c1,int c2);
void sort();
void print();

int main()
{
	int i,j,total_cost;
	printf("\nEnter number of vertices:");
	scanf("%d",&n);
	printf("\nEnter the adjacency matrix:\n");
	for(i=0;i<n;i++)
	for(j=0;j<n;j++)
	scanf("%d",&G[i][j]);
	kruskal();
	print();
}

void kruskal()
{
	int belongs[MAX],i,j,cno1,cno2;
	elist.n=0;

	for(i=1;i<n;i++)
	for(j=0;j<i;j++)
	{
		if(G[i][j]!=0)
		{
```

```
                    elist.data[elist.n].u=i;
                    elist.data[elist.n].v=j;
                    elist.data[elist.n].w=G[i][j];
                    elist.n++;
                }
        }

        sort();
        for(i=0;i<n;i++)
        belongs[i]=i;
        spanlist.n=0;
        for(i=0;i<elist.n;i++)
        {
                cno1=find(belongs,elist.data[i].u);
                cno2=find(belongs,elist.data[i].v);
                if(cno1!=cno2)
                {
                        spanlist.data[spanlist.n]=elist.data[i];
                        spanlist.n=spanlist.n+1;
                        union1(belongs,cno1,cno2);
                }
        }
}

int find(int belongs[],int vertexno)
{
        return(belongs[vertexno]);
}
```

```
void union1(int belongs[],int c1,int c2)
{
	int i;
	for(i=0;i<n;i++)
	if(belongs[i]==c2)
	belongs[i]=c1;
}

void sort()
{
	int i,j;
	edge temp;
	for(i=1;i<elist.n;i++)
	for(j=0;j<elist.n-1;j++)
	if(elist.data[j].w>elist.data[j+1].w)
	{
		temp=elist.data[j];
		elist.data[j]=elist.data[j+1];
		elist.data[j+1]=temp;
	}
}

void print()
{
	int i,cost=0;
	for(i=0;i<spanlist.n;i++)
	{
		printf("\n%d\t%d\t%d",spanlist.data[i].u,spanlist.data[i].v,spanlist.data[i].w);
		cost=cost+spanlist.data[i].w;
```

```
	}

printf("\n\nCost of the spanning tree=%d",cost);
}
```

OUTPUT::

```
Enter the no. of vertices:6

Enter the cost adjacency matrix:
0 3 1 6 0 0
3 0 5 0 3 0
1 5 0 5 6 4
6 0 5 0 0 2
0 3 6 0 0 6
0 0 4 2 6 0
The edges of Minimum Cost Spanning Tree are
1 edge (1,3) =1
2 edge (4,6) =2
3 edge (1,2) =3
4 edge (2,5) =3
5 edge (3,6) =4

        Minimum cost = 13
```

4**. Write a C program to input a graph and weighted graph (connected or disconnected) and use Prim's algorithm to find the cost of the minimum spanning tree (or forest), along with the edges.**

Algorithm::

i. T = ∅;

ii. U = { 1 };

iii. while (U ≠ V)

iv. let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U;

v. T = T ∪ {(u, v)}

vi. U = U ∪ {v}

Source code::

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

#define INF 9999999

#define V 5

int G[V][V] = {
```

```
 {0, 9, 75, 0, 0},
 {9, 0, 95, 19, 42},
 {75, 95, 0, 51, 66},
 {0, 19, 51, 0, 31},
 {0, 42, 66, 31, 0}};

int main() {
 int no_edge;

 int selected[V];

 memset(selected, false, sizeof(selected));

 no_edge = 0;

 selected[0] = true;

 int x;
 int y;

 printf("Edge : Weight\n");

 while (no_edge < V - 1) {

  int min = INF;
  x = 0;
  y = 0;

  for (int i = 0; i < V; i++) {
```

```
      if (selected[i]) {
        for (int j = 0; j < V; j++) {
          if (!selected[j] && G[i][j]) {
            if (min > G[i][j]) {
              min = G[i][j];
              x = i;
              y = j;
            }
          }
        }
      }
    }
    printf("%d - %d : %d\n", x, y, G[x][y]);
    selected[y] = true;
    no_edge++;
  }

  return 0;
}
```

Output::

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

5. **Write a C program to input a graph and apply Warshall's algorithm to determine the existence of path between all pair of vertices.**

Algorithm::

i. Algorithm Warshall(A[1...n, 1...n]) // A is the adjacency matrix

ii. R(0) ← A

iii. for k ← 1 to n do

iv. for i ← 1 to n do

v. for j ← to n do

vi. R(k)[i, j] ← R(k-1)[i, j] or (R(k-1)[i, k] and R(k-1)[k, j])

vii. return R(n)

source code::

```
#include<stdio.h>
#include<math.h>
int max(int,int);
void warshal(int p[10][10],int n) {
	int i,j,k;
	for (k=1;k<=n;k++)
	  for (i=1;i<=n;i++)
	   for (j=1;j<=n;j++)
	    p[i][j]=max(p[i][j],p[i][k]&&p[k][j]);
}
int max(int a,int b) {
	;
	if(a>b)
	 return(a); else
	 return(b);
}
int main() {
	int p[10][10]= {
		0
	}
	,n,e,u,v,i,j;
	printf("\n Enter the number of vertices:");
	scanf("%d",&n);
	printf("\n Enter the number of edges:");
	scanf("%d",&e);
	for (i=1;i<=e;i++) {
		printf("\n Enter the end vertices of edge %d:",i);
		scanf("%d%d",&u,&v);
		p[u][v]=1;
```

```
	}
	printf("\n Matrix of input data: \n");
	for (i=1;i<=n;i++) {
		for (j=1;j<=n;j++)
		   printf("%d\t",p[i][j]);
		printf("\n");
	}
	warshal(p,n);
	printf("\n Transitive closure: \n");
	for (i=1;i<=n;i++) {
		for (j=1;j<=n;j++)
		   printf("%d\t",p[i][j]);
		printf("\n");
	}
	return 0;
}
```

Output::

```
 Enter the number of vertices:3

 Enter the number of edges:4

 Enter the end vertices of edge 1:7
8

 Enter the end vertices of edge 2:9
4

 Enter the end vertices of edge 3:5
6

 Enter the end vertices of edge 4:1
2

 Matrix of input data:
0       1       0
0       0       0
0       0       0

 Transitive closure:
0       1       0
0       0       0
0       0       0
```

6. **Write a C program to input a graph and apply Floyd's Warshall's algorithm to find the shortest distance between all pair of vertices present in the graph.**

Algorithm::

1. n = no of vertices
2. A = matrix of dimension n*n
3. for k = 1 to n
4. for i = 1 to n
5. for j = 1 to n
6. Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])
7. return A

Source code::

```
#include<stdio.h>
int min(int,int);
void floyds(int p[10][10],int n)
{
 int i,j,k;
 for(k=1;k<=n;k++)
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    if(i==j)
     p[i][j]=0;
    else
     p[i][j]=min(p[i][j],p[i][k]+p[k][j]);
}
int min(int a,int b)
{
 if(a<b)
  return(a);
 else
```

```
 return(b);
}
int main()
{
 int p[10][10],w,n,e,u,v,i,j;;
 printf("\n Enter the number of vertices:");
 scanf("%d",&n);
 printf("\n Enter the number of edges:\n");
 scanf("%d",&e);
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
   p[i][j]=999;
 }
 for(i=1;i<=e;i++)
 {
  printf("\n Enter the end vertices of edge%d with its weight \n",i);
  scanf("%d%d%d",&u,&v,&w);
  p[u][v]=w;
 }
 printf("\n Matrix of input data:\n");
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
   printf("%d \t",p[i][j]);
  printf("\n");
 }
 floyds(p,n);
 printf("\n Transitive closure:\n");
```

```
for(i=1;i<=n;i++)
{
 for(j=1;j<=n;j++)
  printf("%d \t",p[i][j]);
 printf("\n");
}
printf("\n The shortest paths are:\n");
for(i=1;i<=n;i++)
 for(j=1;j<=n;j++)
 {
  if(i!=j)
   printf("\n <%d,%d>=%d",i,j,p[i][j]);
 }
 return 0;
}
```

Output::

```
 Enter the number of vertices:2

 Enter the number of edges:
2

 Enter the end vertices of edge1 with its weight
5
8
6

 Enter the end vertices of edge2 with its weight
9
3
4

 Matrix of input data:
999     999
999     999

 Transitive closure:
0       999
999     0

 The shortest paths are:

 <1,2>=999
 <2,1>=999
```

7. **Write a C program to input a graph along with a source vertex. Use Dijkstra's algorithm to find the shortest distance between the source vertex to all other vertex.**

Algorithm::

i. function dijkstra(G, S)

ii. for each vertex V in G

iii. distance[V] <- infinite

iv. previous[V] <- NULL

v. If V != S, add V to Priority Queue Q

vi. distance[S] <- 0

vii. while Q IS NOT EMPTY

viii. U <- Extract MIN from Q

ix. for each unvisited neighbour V of U

x. tempDistance <- distance[U] + edge_weight(U, V)

xi. if tempDistance < distance[V]

xii. distance[V] <- tempDistance

xiii. previous[V] <- U

xiv. return distance[], previous[]

Source code::

```
              #include <stdio.h>
#define INFINITY 9999
#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
  int cost[MAX][MAX], distance[MAX], pred[MAX];
  int visited[MAX], count, mindistance, nextnode, i, j;

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

```
  for (j = 0; j < n; j++)
    if (Graph[i][j] == 0)
      cost[i][j] = INFINITY;
    else
      cost[i][j] = Graph[i][j];

for (i = 0; i < n; i++) {
  distance[i] = cost[start][i];
  pred[i] = start;
  visited[i] = 0;
}

distance[start] = 0;
visited[start] = 1;
count = 1;

while (count < n - 1) {
  mindistance = INFINITY;

  for (i = 0; i < n; i++)
    if (distance[i] < mindistance && !visited[i]) {
      mindistance = distance[i];
      nextnode = i;
    }

  visited[nextnode] = 1;
  for (i = 0; i < n; i++)
    if (!visited[i])
      if (mindistance + cost[nextnode][i] < distance[i]) {
```

```
        distance[i] = mindistance + cost[nextnode][i];
        pred[i] = nextnode;
      }
    count++;
  }

  for (i = 0; i < n; i++)
    if (i != start) {
      printf("\nDistance from source to %d: %d", i, distance[i]);
    }
}
int main() {
  int Graph[MAX][MAX], i, j, n, u;
  n = 7;

  Graph[0][0] = 0;
  Graph[0][1] = 0;
  Graph[0][2] = 1;
  Graph[0][3] = 2;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 0;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 3;
```

```
Graph[1][6] = 0;

Graph[2][0] = 1;
Graph[2][1] = 2;
Graph[2][2] = 0;
Graph[2][3] = 1;
Graph[2][4] = 3;
Graph[2][5] = 0;
Graph[2][6] = 0;

Graph[3][0] = 2;
Graph[3][1] = 0;
Graph[3][2] = 1;
Graph[3][3] = 0;
Graph[3][4] = 0;
Graph[3][5] = 0;
Graph[3][6] = 1;

Graph[4][0] = 0;
Graph[4][1] = 0;
Graph[4][2] = 3;
Graph[4][3] = 0;
Graph[4][4] = 0;
Graph[4][5] = 2;
Graph[4][6] = 0;

Graph[5][0] = 0;
Graph[5][1] = 3;
Graph[5][2] = 0;
```

```
  Graph[5][3] = 0;
  Graph[5][4] = 2;
  Graph[5][5] = 0;
  Graph[5][6] = 1;

  Graph[6][0] = 0;
  Graph[6][1] = 0;
  Graph[6][2] = 0;
  Graph[6][3] = 1;
  Graph[6][4] = 0;
  Graph[6][5] = 1;
  Graph[6][6] = 0;

  u = 0;
  Dijkstra(Graph, n, u);

  return 0;
}
```

Output::

```
Vertex   Distance from Source
0                 0
1                 4
2                 12
3                 19
4                 21
5                 11
6                 9
7                 8
8                 14
```