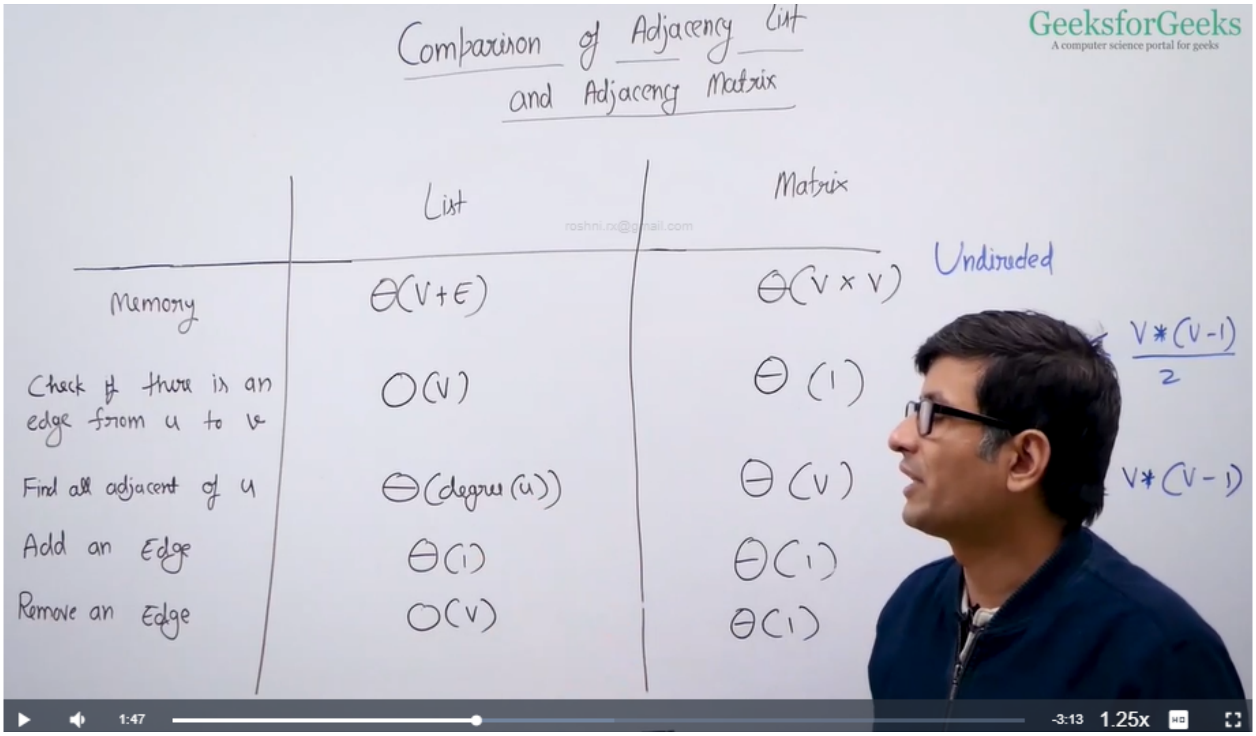
# GRAPH




## 1. ADJACENCY LIST IMPLEMENTATION

```
#include<bits/stdc++.h>
using namespace std;

void addEdge(vector<int> adj[], int u, int v)
{
    adj[u].push_back(v);
    adj[v].push_back(u);
}
void printGraph(vector<int> adj[], int V)
{
    for (int i = 0; i < V; i++)
    {
```

```
        for (int x : adj[i])
          cout << x <<" ";
        cout<<"\n";
    }
}
int main()
{
    int V = 4;
    vector<int> adj[V];
    addEdge(adj, 0, 1);
    addEdge(adj, 0, 2);
    addEdge(adj, 1, 2);
    addEdge(adj, 1, 3);

    printGraph(adj, V);
    return 0;
}
```

2. **Given an undirected graph and a source vertex 's' ,print B.F.S. from given source.**

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

void BFS(vector<int> adj[], int V, int s)
{
        bool visited[V];
        for(int i = 0; i < V; i++)
                visited[i] = false;

        queue<int>  q;

        visited[s] = true;
```

```
        q.push(s);

        while(q.empty()==false)
        {
                int u = q.front();
                q.pop();
                cout << u << " ";

                for(int v:adj[u]){
                    if(visited[v]==false){
                        visited[v]=true;
                        q.push(v);
                    }
                }
        }
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main()
{
        int V=5;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,0,2);
        addEdge(adj,1,2);
        addEdge(adj,2,3);
        addEdge(adj,1,3);
        addEdge(adj,3,4);
        addEdge(adj,2,4);

        cout << "Following is Breadth First Traversal: "<< endl;
```

```
        BFS(adj,V,0);

        return 0;
}
```

## 3. BFS ON DISCONNECTED GRAPHS

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

void BFS(vector<int> adj[], int s, bool visited[])
{       queue<int>  q;

        visited[s] = true;
        q.push(s);

        while(q.empty()==false)
        {
                int u = q.front();
                q.pop();
                cout << u << " ";

                for(int v:adj[u]){
                    if(visited[v]==false){
                        visited[v]=true;
                        q.push(v);
                    }
                }
        }
}

void BFSDin(vector<int> adj[], int V){
    bool visited[V];
```

```
        for(int i = 0;i<V; i++)
            visited[i] = false;

    for(int i=0;i<V;i++){
        if(visited[i]==false)
            BFS(adj,i,visited);
    }
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main()
{
        int V=7;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,0,2);
        addEdge(adj,2,3);
        addEdge(adj,1,3);
        addEdge(adj,4,5);
        addEdge(adj,5,6);
        addEdge(adj,4,6);

        cout << "Following is Breadth First Traversal: "<< endl;
        BFSDin(adj,V);

        return 0;
}
```

4. **Print number of islands in a graph (or number of connected components in a graph).**

```
#include<bits/stdc++.h>
using namespace std;

void BFS(vector<int> adj[], int s, bool visited[])
{       queue<int>  q;

        visited[s] = true;
        q.push(s);

        while(q.empty()==false)
        {
                int u = q.front();
                q.pop();

                for(int v:adj[u]){
                    if(visited[v]==false){
                        visited[v]=true;
                        q.push(v);
                    }
                }
        }
}

int BFSDin(vector<int> adj[], int V){
    bool visited[V]; int count=0;
        for(int i = 0;i<V; i++)
                visited[i] = false;

    for(int i=0;i<V;i++){
        if(visited[i]==false)
            {BFS(adj,i,visited);count++;}
    }

    return count;
}
```

```
void addEdge(vector<int> adj[], int u, int v){
   adj[u].push_back(v);
   adj[v].push_back(u);
}

int main()
{
       int V=7;
       vector<int> adj[V];
       addEdge(adj,0,1);
       addEdge(adj,0,2);
       addEdge(adj,2,3);
       addEdge(adj,1,3);
       addEdge(adj,4,5);
       addEdge(adj,5,6);
       addEdge(adj,4,6);

       cout << "Number of islands: "<<BFSDin(adj,V);

       return 0;
}
```

5. **DFS**

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

void DFSRec(vector<int> adj[], int s, bool visited[])
{
   visited[s]=true;
   cout<< s <<" ";
```

```
    for(int u:adj[s]){
        if(visited[u]==false)
            DFSRec(adj,u,visited);
    }
}

void DFS(vector<int> adj[], int V, int s){
    bool visited[V];
        for(int i = 0;i<V; i++)
                visited[i] = false;

    DFSRec(adj,s,visited);
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main()
{
        int V=5;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,0,2);
        addEdge(adj,2,3);
        addEdge(adj,1,3);
        addEdge(adj,1,4);
        addEdge(adj,3,4);

        cout << "Following is Depth First Traversal: "<< endl;
        DFS(adj,V,0);

        return 0;
```

```
}
```

## 6. DFS FOR DISCONNECTED GRAPHS

```
#include<bits/stdc++.h>
using namespace std;

void DFSRec(vector<int> adj[], int s, bool visited[])
{
    visited[s]=true;
    cout<< s <<" ";

    for(int u:adj[s]){
        if(visited[u]==false)
            DFSRec(adj,u,visited);
    }
}

void DFS(vector<int> adj[], int V){
    bool visited[V];
        for(int i = 0;i<V; i++)
                visited[i] = false;

    for(int i=0;i<V;i++){
        if(visited[i]==false)
            DFSRec(adj,i,visited);
    }
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```
int main()
{
        int V=5;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,0,2);
        addEdge(adj,1,2);
        addEdge(adj,3,4);

        cout << "Following is Depth First Traversal for disconnected
graphs: "<< endl;
        DFS(adj,V);

        return 0;
}
```

7. **DFS FOR FINDING NUMBER OF CONNECTED COMPONENTS IN AN UNDIRECTED GRAPH**

```
#include<bits/stdc++.h>
using namespace std;

void DFSRec(vector<int> adj[], int s, bool visited[])
{
   visited[s]=true;

   for(int u:adj[s]){
      if(visited[u]==false)
         DFSRec(adj,u,visited);
   }
}

int DFS(vector<int> adj[], int V){
   int count=0;
```

```
    bool visited[V];
        for(int i = 0;i<V; i++)
               visited[i] = false;

    for(int i=0;i<V;i++){
       if(visited[i]==false)
          {DFSRec(adj,i,visited);count++;}
    }
    return count;
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main()
{
        int V=5;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,0,2);
        addEdge(adj,1,2);
        addEdge(adj,3,4);

        cout << "Number of connected components: "<< DFS(adj,V);

        return 0;
}
```

## 8. SHORTEST PATH IN AN UNWEIGHTED GRAPH

T:O(V+E)

```
#include<bits/stdc++.h>
```

```
using namespace std;

void BFS(vector<int> adj[], int V, int s,int dist[])
{
	bool visited[V];
	for(int i = 0; i < V; i++)
		visited[i] = false;

	queue<int>  q;

	visited[s] = true;
	q.push(s);

	while(q.empty()==false)
	{
		int u = q.front();
		q.pop();

		for(int v:adj[u]){
		    if(visited[v]==false){
		        dist[v]=dist[u]+1;
		        visited[v]=true;
		        q.push(v);
		    }
		}
	}
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main()
{
```

```
        int V=4;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,1,2);
        addEdge(adj,2,3);
        addEdge(adj,0,2);
        addEdge(adj,1,3);
    int dist[V];
    for(int i=0;i<V;i++){
        dist[i]=INT_MAX;
    }
        dist[0]=0;
        BFS(adj,V,0,dist);

    for(int i=0;i<V;i++){
        cout<<dist[i]<<" ";
    }

        return 0;
}
```

## 9. DETECT CYCLE IN UNDIRECTED GRAPH

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

bool DFSRec(vector<int> adj[], int s,bool visited[], int parent)
{
    visited[s]=true;

    for(int u:adj[s]){
        if(visited[u]==false){
```

```
        if(DFSRec(adj,u,visited,s)==true)
            {return true;}}
    else if(u!=parent)
        {return true;}
  }
  return false;
}

bool DFS(vector<int> adj[], int V){
  bool visited[V];
      for(int i=0;i<V; i++)
              visited[i] = false;

  for(int i=0;i<V;i++){
      if(visited[i]==false)
          if(DFSRec(adj,i,visited,-1)==true)
              return true;
  }
  return false;
}

void addEdge(vector<int> adj[], int u, int v){
  adj[u].push_back(v);
  adj[v].push_back(u);
}

int main()
{
      int V=6;
      vector<int> adj[V];
      addEdge(adj,0,1);
      addEdge(adj,1,2);
      addEdge(adj,2,4);
      addEdge(adj,4,5);
      addEdge(adj,1,3);
```

```
    addEdge(adj,2,3);

    if(DFS(adj,V))
        cout<<"Cycle found";
    else
        cout<<"No cycle found";

    return 0;
}
```

## 10. DETECT CYCLE IN A DIRECTED GRAPH USING DFS

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

bool DFSRec(vector<int> adj[], int s,bool visited[], bool recSt[])
{
    visited[s]=true;
    recSt[s]=true;

    for(int u:adj[s]){
        if(visited[u]==false && DFSRec(adj,u,visited,recSt)==true)
                {return true;}
        else if(recSt[u]==true)
            {return true;}
    }
    recSt[s]=false;
    return false;
}

bool DFS(vector<int> adj[], int V){
    bool visited[V];
```

```
        for(int i=0;i<V; i++)
                visited[i] = false;
        bool recSt[V];
        for(int i=0;i<V; i++)
                recSt[i] = false;

    for(int i=0;i<V;i++){
        if(visited[i]==false)
            if(DFSRec(adj,i,visited,recSt)==true)
                return true;
    }
    return false;
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
}

int main()
{
        int V=6;
        vector<int> adj[V];
        addEdge(adj,0,1);
        addEdge(adj,2,1);
        addEdge(adj,2,3);
        addEdge(adj,3,4);
        addEdge(adj,4,5);
        addEdge(adj,5,3);

        if(DFS(adj,V))
            cout<<"Cycle found";
        else
            cout<<"No cycle found";

        return 0;
```

```
}
```

## 11. TOPOLOGICAL SORTING (BFS)

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

void topologicalSort(vector<int> adj[], int V)
{
    vector<int> in_degree(V, 0);

    for (int u = 0; u < V; u++) {
        for (int x:adj[u])
            in_degree[x]++;
    }

    queue<int> q;
    for (int i = 0; i < V; i++)
        if (in_degree[i] == 0)
            q.push(i);


    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cout<<u<<" ";

        for (int x: adj[u])
            if (--in_degree[x] == 0)
                q.push(x);
    }
}
```

```
void addEdge(vector<int> adj[], int u, int v){
   adj[u].push_back(v);
}

int main()
{
       int V=5;
       vector<int> adj[V];
       addEdge(adj,0, 2);
   addEdge(adj,0, 3);
   addEdge(adj,1, 3);
   addEdge(adj,1, 4);
   addEdge(adj,2, 3);

   cout << "Following is a Topological Sort of\n";
   topologicalSort(adj,V);

       return 0;
}
```

## 12. DETECT CYCLE IN DIRECTED GRAPH (BFS)

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

void topologicalSort(vector<int> adj[], int V)
{
   vector<int> in_degree(V, 0);

   for (int u = 0; u < V; u++) {
      for (int x:adj[u])
```

```
            in_degree[x]++;
    }

    queue<int> q;
    for (int i = 0; i < V; i++)
        if (in_degree[i] == 0)
            q.push(i);

    int count=0;
    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int x: adj[u])
            if (--in_degree[x] == 0)
                q.push(x);
        count++;
    }
    if (count != V) {
        cout << "There exists a cycle in the graph\n";
    }
    else{
        cout << "There exists no cycle in the graph\n";
    }
}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
}

int main()
{
         int V=5;
         vector<int> adj[V];
         addEdge(adj,0, 1);
```

```
    addEdge(adj,4, 1);
    addEdge(adj,1, 2);
    addEdge(adj,2, 3);
    addEdge(adj,3, 1);

    topologicalSort(adj,V);

        return 0;
}
```

**13. TOPOLOGICAL SORT (DFS)**

T:O(V+E)

```
#include<bits/stdc++.h>
using namespace std;

void DFS(vector<int> adj[], int u,stack<int> &st, bool visited[])
{
    visited[u]=true;

    for(int v:adj[u]){
        if(visited[v]==false)
            DFS(adj,v,st,visited);
    }
    st.push(u);
}


void topologicalSort(vector<int> adj[], int V)
{
    bool visited[V];
        for(int i = 0;i<V; i++)
                visited[i] = false;
```

```
        stack<int> st;

    for(int u=0;u<V;u++){
        if(visited[u]==false){
            DFS(adj,u,st,visited);
        }
    }

    while(st.empty()==false){
        int u=st.top();
        st.pop();
        cout<<u<<" ";
    }

}

void addEdge(vector<int> adj[], int u, int v){
    adj[u].push_back(v);
}

int main()
{
        int V=5;
        vector<int> adj[V];
        addEdge(adj,0, 1);
    addEdge(adj,1, 3);
    addEdge(adj,2, 3);
    addEdge(adj,3, 4);
    addEdge(adj,2, 4);

    cout << "Following is a Topological Sort of\n";
    topologicalSort(adj,V);

        return 0;
}
```

## 14. SHORTEST PATH IN DIRECTED ACYCLIC GRAPH

T:O(V+E)

```
#include <bits/stdc++.h>
#define INF INT_MAX
using namespace std;

class AdjListNode
{
	int v;
	int weight;
public:
	AdjListNode(int _v, int _w) { v = _v; weight = _w;}
	int getV()     { return v; }
	int getWeight() { return weight; }
};

class Graph
{
	int V;

	list<AdjListNode> *adj;

	void topologicalSortUtil(int v, bool visited[], stack<int> &Stack);
public:
	Graph(int V);

	void addEdge(int u, int v, int weight);

	void shortestPath(int s);
};

Graph::Graph(int V)
```

```
{
        this->V = V;
        adj = new list<AdjListNode>[V];
}

void Graph::addEdge(int u, int v, int weight)
{
        AdjListNode node(v, weight);
        adj[u].push_back(node);
}

void Graph::topologicalSortUtil(int v, bool visited[], stack<int> &Stack)
{

        visited[v] = true;

        list<AdjListNode>::iterator i;
        for (i = adj[v].begin(); i != adj[v].end(); ++i)
        {
                AdjListNode node = *i;
                if (!visited[node.getV()])
                        topologicalSortUtil(node.getV(), visited, Stack);
        }

        Stack.push(v);
}

void Graph::shortestPath(int s)
{
        stack<int> Stack;
        int dist[V];

        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
                visited[i] = false;
```

```
        for (int i = 0; i < V; i++)
                if (visited[i] == false)
                        topologicalSortUtil(i, visited, Stack);

        for (int i = 0; i < V; i++)
                dist[i] = INF;
        dist[s] = 0;

        while (Stack.empty() == false)
        {
                int u = Stack.top();
                Stack.pop();

                list<AdjListNode>::iterator i;
                if (dist[u] != INF)
                {
                for (i = adj[u].begin(); i != adj[u].end(); ++i)
                        if (dist[i->getV()] > dist[u] + i->getWeight())
                                dist[i->getV()] = dist[u] + i->getWeight();
                }
        }

        for (int i = 0; i < V; i++)
                (dist[i] == INF)? cout << "INF ": cout << dist[i] << " ";
}

int main()
{
        Graph g(6);
        g.addEdge(0, 1, 2);
        g.addEdge(0, 4, 1);
        g.addEdge(1, 2, 3);
        g.addEdge(4, 2, 2);
        g.addEdge(4, 5, 4);
```

```
        g.addEdge(2, 3, 6);
        g.addEdge(5, 3, 1);

        int s = 0;
        cout << "Following are shortest distances from source " << s
<<" \n";
        g.shortestPath(s);

        return 0;
}
```

## 15. PRIM'S ALGORITHM

T: THETA(V*V)

```
#include <bits/stdc++.h>
using namespace std;
#define V 4

int primMST(int graph[V][V])
{

        int key[V];int res=0;
        fill(key,key+V,INT_MAX);
        bool mSet[V]; key[0]=0;

        for (int count = 0; count < V ; count++)
        {
                int u = -1;

                for(int i=0;i<V;i++)
                    if(!mSet[i]&&(u==-1||key[i]<key[u]))
                        u=i;
                mSet[u] = true;
```

```
            res+=key[u];


            for (int v = 0; v < V; v++)

                if (graph[u][v]!=0 && mSet[v] == false)
                    key[v] = min(key[v],graph[u][v]);
        }
    return res;
}

int main()
{
        int graph[V][V] = { { 0, 5, 8, 0},
                            { 5, 0, 10, 15 },
                            { 8, 10, 0, 20 },
                            { 0, 15, 20, 0 },};

        cout<<primMST(graph);

        return 0;
}
```

**16. DIJKSTRA'S ALGORITHM**

T:O(V+E)*logV

```
#include <bits/stdc++.h>
using namespace std;
#define V 4

vector<int> djikstra(int graph[V][V],int src)
{
```

```
        vector<int> dist(V,INT_MAX);
        dist[src]=0;
        vector<bool> fin(V,false);

        for (int count = 0; count < V-1 ; count++)
        {
                int u = -1;

                for(int i=0;i<V;i++)
                    if(!fin[i]&&(u==-1||dist[i]<dist[u]))
                        u=i;
                fin[u] = true;

                for (int v = 0; v < V; v++)

                        if (graph[u][v]!=0 && fin[v] == false)
                                dist[v] = min(dist[v],dist[u]+graph[u][v]);
        }
    return dist;
}

int main()
{
        int graph[V][V] = { { 0, 50, 100, 0},
                                        { 50, 0, 30, 200 },
                                        { 100, 30, 0, 20 },
                                        { 0, 200, 20, 0 },};

        for(int x: djikstra(graph,0)){
            cout<<x<<" ";
        }

        return 0;
}
```

## 17. KOSARAJU ALGORITHM (STRONGLY CONNECTED COMPONENTS)

T:O(V+E)

```
#include <iostream>
#include <list>
#include <stack>
using namespace std;

class Graph
{
	int V;
	list<int> *adj;

	void fillOrder(int v, bool visited[], stack<int> &s);

	void DFSUtil(int v, bool visited[]);
public:
	Graph(int V);
	void addEdge(int v, int w);

	void printSCCs();

	Graph getTranspose();
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::DFSUtil(int v, bool visited[])
```

```
{
	visited[v] = true;
	cout << v << " ";

	list<int>::iterator i;
	for (i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFSUtil(*i, visited);
}

Graph Graph::getTranspose()
{
	Graph g(V);
	for (int v = 0; v < V; v++)
	{
		list<int>::iterator i;
		for(i = adj[v].begin(); i != adj[v].end(); ++i)
		{
			g.adj[*i].push_back(v);
		}
	}
	return g;
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w);
}

void Graph::fillOrder(int v, bool visited[], stack<int> &s)
{
	visited[v] = true;

	list<int>::iterator i;
	for(i = adj[v].begin(); i != adj[v].end(); ++i)
```

```
        if(!visited[*i])
            fillOrder(*i, visited, s);

    s.push(v);
}

void Graph::printSCCs()
{
    stack<int> s;

    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    for(int i = 0; i < V; i++)
        if(visited[i] == false)
            fillOrder(i, visited, s);

    Graph gr = getTranspose();

    for(int i = 0; i < V; i++)
        visited[i] = false;

    while (s.empty() == false)
    {
        int v = s.top();
        s.pop();

        if (visited[v] == false)
        {
            gr.DFSUtil(v, visited);
            cout << endl;
        }
    }
}
```

```
int main()
{
	Graph g(5);
	g.addEdge(1, 0);
	g.addEdge(0, 2);
	g.addEdge(2, 1);
	g.addEdge(0, 3);
	g.addEdge(3, 4);

	cout << "Following are strongly connected components in given
graph \n";
	g.printSCCs();

	return 0;
}
```

## 18. BELLMAN FORD ALGORITHM

```
#include <bits/stdc++.h>

struct Edge {
	int src, dest, weight;
};

struct Graph {
	int V, E;
	struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
	struct Graph* graph = new Graph;
	graph->V = V;
```

```
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
```

```
            int weight = graph->edge[i].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                printf("Graph contains negative weight cycle");
                return;
            }
        }

        printArr(dist, V);

        return;
    }

    int main()
    {
        int V = 4;
        int E = 5;
        struct Graph* graph = createGraph(V, E);

        // add edge 0-1 (or A-B)
        graph->edge[0].src = 0;
        graph->edge[0].dest = 1;
        graph->edge[0].weight = 1;

        // add edge 0-2 (or A-C)
        graph->edge[1].src = 0;
        graph->edge[1].dest = 2;
        graph->edge[1].weight = 4;

        // add edge 1-2 (or B-C)
        graph->edge[2].src = 1;
        graph->edge[2].dest = 2;
        graph->edge[2].weight = -3;

        // add edge 1-3 (or B-D)
        graph->edge[3].src = 1;
```

```
        graph->edge[3].dest = 3;
        graph->edge[3].weight = 2;

        // add edge 2-3 (or C-D)
        graph->edge[4].src = 2;
        graph->edge[4].dest = 3;
        graph->edge[4].weight = 3;

        BellmanFord(graph, 0);

        return 0;
}
```

## 19. ARTICULATION POINT

```
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

class Graph
{
        int V;
        list<int> *adj;
        void APUtil(int v, bool visited[], int disc[], int low[], int parent[],
bool ap[]);
public:
        Graph(int V);
        void addEdge(int v, int w);
        void AP();
};

Graph::Graph(int V)
{
```

```
        this->V = V;
        adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
        adj[v].push_back(w);
        adj[w].push_back(v);
}

void Graph::APUtil(int u, bool visited[], int disc[], int low[], int parent[],
bool ap[])
{

        static int time = 0;

        int children = 0;

        visited[u] = true;

        disc[u] = low[u] = ++time;

        list<int>::iterator i;
        for (i = adj[u].begin(); i != adj[u].end(); ++i)
        {
                int v = *i;

                if (!visited[v])
                {
                        children++;
                        parent[v] = u;
                        APUtil(v, visited, disc, low, parent, ap);

                        low[u] = min(low[u], low[v]);
```

```
                if (parent[u] == NIL && children > 1)
                ap[u] = true;
                if (parent[u] != NIL && low[v] >= disc[u])
                ap[u] = true;
            }

            else if (v != parent[u])
                low[u] = min(low[u], disc[v]);
        }
}

void Graph::AP()
{
        bool *visited = new bool[V];
        int *disc = new int[V];
        int *low = new int[V];
        int *parent = new int[V];
        bool *ap = new bool[V];

        for (int i = 0; i < V; i++)
        {
                parent[i] = NIL;
                visited[i] = false;
                ap[i] = false;
        }

        for (int i = 0; i < V; i++)
                if (visited[i] == false)
                        APUtil(i, visited, disc, low, parent, ap);

        for (int i = 0; i < V; i++)
                if (ap[i] == true)
                        cout << i << " ";
}
```

```
int main()
{
	cout << "Articulation points in first graph \n";
	Graph g(5);
	g.addEdge(1, 0);
	g.addEdge(0, 2);
	g.addEdge(2, 1);
	g.addEdge(0, 3);
	g.addEdge(3, 4);
	g.AP();

	return 0;
}
```

## 20. BRIDGES IN GRAPH

```
// A C++ program to find bridges in a given undirected graph
#include<iostream>
#include <list>
#define NIL -1
using namespace std;

// A class that represents an undirected graph
class Graph
{
	int V; // No. of vertices
	list<int> *adj; // A dynamic array of adjacency lists
	void bridgeUtil(int v, bool visited[], int disc[], int low[],
						int parent[]);
public:
	Graph(int V); // Constructor
	void addEdge(int v, int w); // to add an edge to graph
```

```
	void bridge(); // prints all bridges
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w);
	adj[w].push_back(v); // Note: the graph is undirected
}

// A recursive function that finds and prints bridges using
// DFS traversal
// u --> The vertex to be visited next
// visited[] --> keeps tract of visited vertices
// disc[] --> Stores discovery times of visited vertices
// parent[] --> Stores parent vertices in DFS tree
void Graph::bridgeUtil(int u, bool visited[], int disc[],
								int low[], int parent[])
{
	// A static variable is used for simplicity, we can
	// avoid use of static variable by passing a pointer.
	static int time = 0;

	// Mark the current node as visited
	visited[u] = true;

	// Initialize discovery time and low value
	disc[u] = low[u] = ++time;

	// Go through all vertices aadjacent to this
```

```
	list<int>::iterator i;
	for (i = adj[u].begin(); i != adj[u].end(); ++i)
	{
		int v = *i; // v is current adjacent of u

		// If v is not visited yet, then recur for it
		if (!visited[v])
		{
			parent[v] = u;
			bridgeUtil(v, visited, disc, low, parent);

			// Check if the subtree rooted with v has a
			// connection to one of the ancestors of u
			low[u] = min(low[u], low[v]);

			// If the lowest vertex reachable from subtree
			// under v is below u in DFS tree, then u-v
			// is a bridge
			if (low[v] > disc[u])
			cout << u <<" " << v << endl;
		}

		// Update low value of u for parent function calls.
		else if (v != parent[u])
			low[u] = min(low[u], disc[v]);
	}
}

// DFS based function to find all bridges. It uses recursive
// function bridgeUtil()
void Graph::bridge()
{
	// Mark all the vertices as not visited
	bool *visited = new bool[V];
	int *disc = new int[V];
```

```
        int *low = new int[V];
        int *parent = new int[V];

        // Initialize parent and visited arrays
        for (int i = 0; i < V; i++)
        {
                parent[i] = NIL;
                visited[i] = false;
        }

        // Call the recursive helper function to find Bridges
        // in DFS tree rooted with vertex 'i'
        for (int i = 0; i < V; i++)
                if (visited[i] == false)
                        bridgeUtil(i, visited, disc, low, parent);
}


int main()
{
        cout << "Bridges in first graph \n";
        Graph g(5);
        g.addEdge(1, 0);
        g.addEdge(0, 2);
        g.addEdge(2, 1);
        g.addEdge(0, 3);
        g.addEdge(3, 4);
        g.bridge();

        return 0;
}
```

**21. TARJANS ALGORITHM**

```
#include<iostream>
#include <list>
#include <stack>
#define NIL -1
using namespace std;


class Graph
{
	int V;
	list<int> *adj;

	void SCCUtil(int u, int disc[], int low[], stack<int> *st, bool
stackMember[]);
public:
	Graph(int V);
	void addEdge(int v, int w);
	void SCC();
};

Graph::Graph(int V)
{
	this->V = V;
	adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
	adj[v].push_back(w);
}


void Graph::SCCUtil(int u, int disc[], int low[], stack<int> *st, bool
stackMember[])
{
```

```
static int time = 0;

disc[u] = low[u] = ++time;
st->push(u);
stackMember[u] = true;

list<int>::iterator i;
for (i = adj[u].begin(); i != adj[u].end(); ++i)
{
        int v = *i;

        if (disc[v] == -1)
        {
                SCCUtil(v, disc, low, st, stackMember);

                low[u] = min(low[u], low[v]);
        }

        else if (stackMember[v] == true)
                low[u] = min(low[u], disc[v]);
}

int w = 0;
if (low[u] == disc[u])
{
        while (st->top() != u)
        {
                w = (int) st->top();
                cout << w << " ";
                stackMember[w] = false;
                st->pop();
        }
        w = (int) st->top();
        cout << w << "\n";
        stackMember[w] = false;
```

```
                st->pop();
        }
}

void Graph::SCC()
{
        int *disc = new int[V];
        int *low = new int[V];
        bool *stackMember = new bool[V];
        stack<int> *st = new stack<int>();

        for (int i = 0; i < V; i++)
        {
                disc[i] = NIL;
                low[i] = NIL;
                stackMember[i] = false;
        }

        for (int i = 0; i < V; i++)
                if (disc[i] == NIL)
                        SCCUtil(i, disc, low, st, stackMember);
}

int main()
{
        cout << "SCCs in the graph \n";
        Graph g(5);
        g.addEdge(1, 0);
        g.addEdge(0, 2);
        g.addEdge(2, 1);
        g.addEdge(0, 3);
        g.addEdge(3, 4);
        g.SCC();

        return 0; }
```

## 22. KRUSKAL’S ALGORITHM

```
// C++ program for Kruskal's algorithm to find Minimum Spanning
//Tree of a given connected, undirected and weighted graph
#include <bits/stdc++.h>
using namespace std;
// a structure to represent a weighted edge in graph
class Edge
{
	public:
	int src, dest, weight;
};
// a structure to represent a connected, undirected
// and weighted graph
class Graph
{
	public:
	// V-> Number of vertices, E-> Number of edges
	int V, E;
	// graph is represented as an array of edges.
	// Since the graph is undirected, the edge
	// from src to dest is also edge from dest
	// to src. Both are counted as 1 edge here.
	Edge* edge;
};
// Creates a graph with V vertices and E edges
Graph* createGraph(int V, int E)
{
	Graph* graph = new Graph;
	graph->V = V;
	graph->E = E;
	graph->edge = new Edge[E];
	return graph;
}
```

```
// A structure to represent a subset for union-find
class subset
{
	public:
	int parent;
	int rank;
};
// A utility function to find set of an element i
// (uses path compression technique)
int find(subset subsets[], int i)
{
	// find root and make root as parent of i
	// (path compression)
	if (subsets[i].parent != i)
		subsets[i].parent = find(subsets, subsets[i].parent);
	return subsets[i].parent;
}
// A function that does union of two sets of x and y
// (uses union by rank)
void Union(subset subsets[], int x, int y)
{
	int xroot = find(subsets, x);
	int yroot = find(subsets, y);
	// Attach smaller rank tree under root of high
	// rank tree (Union by Rank)
	if (subsets[xroot].rank < subsets[yroot].rank)
		subsets[xroot].parent = yroot;
	else if (subsets[xroot].rank > subsets[yroot].rank)
		subsets[yroot].parent = xroot;
	// If ranks are same, then make one as root and
	// increment its rank by one
	else
	{
		subsets[yroot].parent = xroot;
		subsets[xroot].rank++;
```

```
        }
}
// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
        Edge* a1 = (Edge*)a;
        Edge* b1 = (Edge*)b;
        return a1->weight > b1->weight;
}
// The main function to construct MST using Kruskal's algorithm
void KruskalMST(Graph* graph)
{
        int V = graph->V;
        Edge result[V]; // Tnis will store the resultant MST
        int e = 0; // An index variable, used for result[]
        int i = 0; // An index variable, used for sorted edges
        // Step 1: Sort all the edges in non-decreasing
        // order of their weight. If we are not allowed to
        // change the given graph, we can create a copy of
        // array of edges
        qsort(graph->edge, graph->E, sizeof(graph->edge[0]),
myComp);
// Allocate memory for creating V ssubsets
        subset *subsets = new subset[( V * sizeof(subset) )];

        // Create V subsets with single elements
        for (int v = 0; v < V; ++v)
        {
                subsets[v].parent = v;
                subsets[v].rank = 0;
        }
    int res =0;
        // Number of edges to be taken is equal to V-1
        while (e < V - 1 && i < graph->E)
```

```
        {
                // Step 2: Pick the smallest edge. And increment
                // the index for next iteration
                Edge next_edge = graph->edge[i++];

                int x = find(subsets, next_edge.src);
                int y = find(subsets, next_edge.dest);

                // If including this edge does't cause cycle,
                // include it in result and increment the index
                // of result for next edge
                if (x != y)
                {
                        result[e++] = next_edge;
                        Union(subsets, x, y);
                        res+=next_edge.weight;
                }
                // Else discard the next_edge
        }

        // print the contents of result[] to display the
        // built MST

        cout<<"Weight of MST is: "<<res<<endl;
        return;
}
int main()
{
       int V = 5; // Number of vertices in graph
                int E = 7; // Number of edges in graph
                Graph* graph = createGraph(V, E);

                // add edge 0-1
                graph->edge[0].src = 0;
                graph->edge[0].dest = 1;
```

```
        graph->edge[0].weight = 10;

        // add edge 0-2
        graph->edge[1].src = 0;
        graph->edge[1].dest = 2;
        graph->edge[1].weight = 8;

        // add edge 0-3
        graph->edge[2].src = 1;
        graph->edge[2].dest = 2;
        graph->edge[2].weight = 5;

        // add edge 1-3
        graph->edge[3].src = 1;
        graph->edge[3].dest = 3;
        graph->edge[3].weight = 3;

        // add edge 2-3
        graph->edge[4].src = 2;
        graph->edge[4].dest = 3;
        graph->edge[4].weight = 4;

        //add egde 2-4
        graph->edge[5].src = 2;
        graph->edge[5].dest = 4;
        graph->edge[5].weight = 12;

        // add edge 3-4
        graph->edge[6].src = 3;
        graph->edge[6].dest = 4;
        graph->edge[6].weight = 15;

    KruskalMST(graph);

    return 0; }
```

# Applications of BFS

1. Shortest Path in an unweighted Graph.
2. Crawlers in Search Engines
3. Peer to Peer Networks
4. Social Networking Search
5. In Garbage Collection (Cheney's Algorithm)
6. Cycle Detection
7. Ford Fulkerson Algorithm
8. Broadcasting



# Applications of DFS

1) Cycle Detection
2) Topological Sorting
3) Strongly Connected Components
4) Solving Maze and Similar Pu
5) Path Finding