1.Create a graph and print it:

Method 1:Adjacency list

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int v,e;
    cin>>v>>e;
    vector<int> g[v+1];
    for(int i=0;i<e;i++)
    {
        int x,y;
        cin>>x>>y;
        g[x].push_back(y);
        g[y].push_back(x);
    }
    for(int i=1;i<=v;i++)
    {
        cout<<i<<"->";
        for(int j=0;j<g[i].size();j++)
        {
            cout<<g[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

Complexity;Time=O(V+E)=space

Method 2:Adjacency matrix

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int v,e;
    cin>>v>>e;
    //vector<int> g[v+1];
    int arr[v+1][v+1];
    memset(arr, 0, sizeof(arr));
    for(int i=0;i<e;i++)
    {
        int x,y;
        cin>>x>>y;
        arr[x][y]=1;
    }
    for(int i=0;i<v;i++)
    {
        for(int j=0;j<v;j++)
        {
            cout<<g[i][j]<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

Complexity;Time=O(V^2)=Space

Input;

```
6 5
1 2
1 5
2 3
3 4
3 6
```

Method 3;Print adjacency list

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

Here in the above for loo,we can use like this also;

```
for(auto x;ad[j])v[i].ush_back(x);
```

Method 4:Linked list graph

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

class Graph{
    public:
    int v;
    list<int> *l;
    Graph(int x){
        this->v=x;
        l=new list<int>[v];
    }
    void addedge(int x,int y){
        l[x].push_back(y);
        l[y].push_back(x);
    }
    void printList(){
        for(int i=0;i<v;i++){
            cout<<i<<"->";
            for(auto x:l[i])cout<<x<<" ";
            cout<<endl;
        }
    }

};
int main(){
    Graph g(4);
    g.addedge(0,1);
    g.addedge(0,2);
    g.addedge(2,3);
    g.addedge(1,2);
    g.printList();
    return 0;
}
```

2.DFS:

```
void get_DFS(vector<int>& v,vector<bool> &vis,int
src,vector<int> adj[])
    {
        vis[src]=1;
        v.push_back(src);
        for(auto x:adj[src])
        {
            if(!vis[x])get_DFS(v,vis,x,adj);
        }
        return;
    }
    vector<int> dfsOfGraph(int V, vector<int> adj[])
    {
        vector<int> v;
        vector<bool> vis(V,0);
        get_DFS(v,vis,0,adj);//single component
        return v;
    }
};
```

4.BFS:

```
vector<int> bfsOfGraph(int V, vector<int> adj[]) {
        queue<int> q;
        vector<int> v;
        vector<bool> vis(V,0);
        vis[0]=1;
        q.push(0);
        while(!q.empty())
        {
            int t=q.front();q.pop();
            v.push_back(t);
            for(auto x:adj[t])
            {
                if(!vis[x])
                {
                    vis[x]=1;
                    q.push(x);
                }
            }
        }
        return v;
    }
```

5.Detect cycle in a directed graph>

```
bool check_cycle(int src,vector<int>& order, vector<int>&
vis,vector<int> adj[])
    {
        vis[src]=1;
        order[src]=1;
        for(auto x:adj[src])
        {
            if(!vis[x])
            {
                bool conf=check_cycle(x,order,vis,adj);
                if(conf==true)return true;
            }
            else if(order[x])return true;
        }
        order[src]=0;
        return false;
    }
    bool isCyclic(int V, vector<int> adj[])
    {
        vector<int> order(V,0);
        vector<int> vis(V,0);
        for(int i=0;i<V;i++)
        {
            if(!vis[i]){
            bool c=check_cycle(i,order,vis,adj);
            if(c==true)return true;}
        }
        return false;
    }
```

Note>We are using order vector only to check whether we again visited a node in the path or not.
Cycle exists only if both vis[src] & order[src] is true both.

Complexity:Same like DFS

6.Detect cycle in an undirected graph:

```
bool check_cycle(int src,int par,vector<int>& vis,vector<int>
adj[])
    {
        vis[src]=1;
        for(auto x:adj[src])
        {
            if(!vis[x])
            {
                bool conf=check_cycle(x,src,vis,adj);
                if(conf==true)return true;
            }
            else if (x!=par)return true;
        }
        return false;
    }
    bool isCycle(int V, vector<int> adj[])
    {
        vector<int> vis(V,0);
        for(int i=0;i<V;i++)
        {
            if(!vis[i]){
```

```
            bool c=check_cycle(i,-1,vis,adj);
            if(c==true)return true;}
        }
        return false;
    }
```

Complexity:Same like DFS.

Note:Here we are using extra variable par(parent) to check if we are visiting a node which is already visited then if that is not parent then we are definitely in loop.

7.Rat in a Maze Problem - I:

```
class Solution{
    public:
    vector<string> v;
    void dfs(int i,int j,vector<vector<int>> m,int n,string
s,vector<vector<int>>& vis)
    {
        if(i<0 or j<0 or i>=n or j>=n)return ;
        if(m[i][j]==0 or vis[i][j]==1)return ;

        if(i==n-1 && j==n-1)
        {
            v.push_back(s);
            return;
        }
        vis[i][j]=1;
        dfs(i-1,j,m,n,s+'U',vis);
        dfs(i+1,j,m,n,s+'D',vis);
        dfs(i,j-1,m,n,s+'L',vis);
        dfs(i,j+1,m,n,s+'R',vis);

        vis[i][j]=0;
    }
    vector<string> findPath(vector<vector<int>> &m, int n)
    {
        if(m[0][0]==0 or m[n-1][n-1]==0)return v;
        vector<vector<int>> vis(n,vector<int>(n,0));
        string s="";
        dfs(0,0,m,n,s,vis);
        sort(v.begin(),v.end());
        return v;

    }
};
```

8.Clone a graph:Undirected graph

```
void dfs(Node* node,Node* copy,vector<Node*> &vis)
    {
        vis[node->val]=copy;
        for(auto x:(node->neighbors))
        {
            if(!vis[x->val])
            {
                Node* newnode=new Node(x->val);
                (copy->neighbors).push_back(newnode);
                dfs(x,newnode,vis);
            }
            else (copy->neighbors).push_back(vis[x->val]);
        }

    }
    Node* cloneGraph(Node* node) {
        if(node==NULL)return node;
        vector<Node*> vis(1000,NULL);
        Node* copy=new Node(node->val);
        dfs(node,copy,vis);
        return copy;
    }
```

10.Prim's Algorithm:

```
#include<bits/stdc++.h>
using namespace std;
#define V 5
int min_key(vector<int> key,vector<bool> mstSet)
{
    int min=INT_MAX,min_index=0;
    for(int i=0;i<V;i++)
    {
        if(!mstSet[i] && key[i] < min)
        {
            min=key[i];
            min_index=i;
        }
    }
    return min_index;
}
```

```
void prim(int graph[V][V])
{
   int parent[V];
   vector<int> key(V,INT_MAX);
   vector<bool> mstSet(V,false);
   parent[0]=-1;
   key[0]=0;
   for(int count=0;count<V-1;count++)
   {
      int u=min_key(key,mstSet);

      mstSet[u]=true;

      for(int v=0;v<V;v++)
      {
         if(graph[u][v] && mstSet[v]==false &&
graph[u][v]<key[v])
         key[v]=graph[u][v],parent[v]=u;
      }

   }
   for(int i=1;i<V;i++)
   {
      cout<<parent[i]<<"-"<<i<<"
"<<"weight"<<graph[i][parent[i]]<<endl;
   }

}
int main(){
   int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };

    prim(graph);
    return 0;
}
```

Complexity:O(V^2)

Method 2:Using adjacency list

```
#include<bits/stdc++.h>
using namespace std;
vector<vector<int>>g[100];//CAN USE HASHMAP ALSO

void prim(int source,int v)
{
    int parent[v];
    vector<bool> vis(v,0);
    vector<int> dis(v,INT_MAX);
    vis[source]=1;
    dis[source]=0;
    parent[source]=-1;
    set<vector<int>> s;
    s.insert({0,source});
    //{weight,node_index}
    while(!s.empty())
```

```
        {
            auto x=*(s.begin());
            s.erase(x);
            vis[x[1]]=1;
            int u=x[1];
            int v=parent[x[1]];

            cout<<v<<"->"<<u<<" ->"<<x[0]<<endl;
            for(auto it:g[x[1]])
            {
                if(vis[it[0]])continue;
                if(dis[it[0]] > it[1])
                {
                    auto r=s.find({dis[it[0]],it[0]});
                    if(r!=s.end())
                    s.erase(r);
                    dis[it[0]]=it[1];
                    s.insert({dis[it[0]],it[0]});
                    parent[it[0]]=x[1];
                }
            }
        }
    }
    int main(){

        int n,m;
        cout<<"enter number of vertices and edges"<<endl;
        cin>>n>>m;
        for(int i=0;i<m;i++)
        {
```

```
        int u,v,w;
        cin>>u>>v>>w;
        g[u].push_back({v,w});
        g[v].push_back({u,w});
    }
    prim(0,n);
    return 0;
}
```

Complexity:Time=O(ElogV)
Space=O(E+V)

11.Dijkstra Algo:

```
#include<bits/stdc++.h>
#include<list>
using namespace std;
int main(){
    int n,m;
    cin>>n>>m;
    unordered_map<int,list<pair<int,int>>> adj;
    for(int i=0;i<m;i++)
    {
        int u,v,w;
        cin>>u>>v>>w;
        adj[u].push_back({v,w});
        adj[v].push_back({u,w});
    }
    set<pair<int,int>> s;
    vector<int> dis(n,INT_MAX);
    s.insert({0,0});
    dis[0]=0;
    while(!s.empty())
    {
        auto i=*(s.begin());
        int node_dis=i.first;
        int node=i.second;
        s.erase(s.begin());
        for(auto neighbour:adj[node])
        {
            if(node_dis+neighbour.second<dis[neighbour.first])
            {
                auto r=s.find({dis[neighbour.first],neighbour.first});
                if(r!=s.end())s.erase(r);

                dis[neighbour.first]=node_dis+neighbour.second;
                s.insert({dis[neighbour.first],neighbour.first});
            }
        }


    }
    for(int i=0;i<dis.size();i++)cout<<dis[i]<<" ";
    return 0;
}
```

12.Topological sort:

Method 1:DFS

```
#include<bits/stdc++.h>
using namespace std;
void dfs_topsort(int node,vector<bool>
&vis,unordered_map<int,list<int>> &adj,stack<int> &s)
{
   vis[node]=1;
   for(auto neighbor:adj[node])
   {
      if(!vis[neighbor])dfs_topsort(neighbor,vis,adj,s);
   }

   s.push(node);
}
int main(){
   int n,m;
   cin>>n>>m;
   unordered_map<int,list<int>> adj;
    //can use  vector<int> adj[n];
   for(int i=0;i<m;i++)
   {
      int u,v;
      cin>>u>>v;
      adj[u].push_back(v);
   }
   vector<bool> vis(n,0);
   stack<int> s;
   for(int i=0;i<n;i++)
```

```
    {
        if(!vis[i]) dfs_topsort(i,vis,adj,s);
    }
    while(!s.empty())
    {
        cout<<s.top()<<" ";
        s.pop();
    }
    return 0;
}
```

Complexity:Time=O(E+V)

Same like dfs

Method 2;BFS

Steps:
1.calculate indegree
2.push all elements having indegree 0 to queue.
3.BFS

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

int main(){
    int n,m;
```

```
cin>>n>>m;
unordered_map<int,list<int>> adj;
for(int i=0;i<m;i++)
{
   int u,v;
   cin>>u>>v;
   adj[u].push_back(v);
}
vector<int> indegree(n);
for(auto i:adj)
{
   for(auto j:i.second)indegree[j]++;
}

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

while(!q.empty())
{
   int top=q.front();
   q.pop();

   cout<<top<<" ";
   for(auto neighbor:adj[top])
   {
      indegree[neighbor]--;
      if(indegree[neighbor]==0)q.push(neighbor);
   }
}
```

```
    return 0;
}
```

13.Implement Bellman Ford Algorithm:

Step 1:Use the below formula for n-1 times
(dist[u]+w < dist[v])dist[v]=dist[u]+w;

Step 2:Use this formula one more time to check negative cycle.if in this step any dist got updated the negative cycle is present.

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n,m;
    cin>>n>>m;
    vector<vector<int>> edges;
    for(int i=0;i<m;i++)
    {
        vector<int> vec;
        int u,v,w;
        cin>>u>>v>>w;
        vec.push_back(u);
        vec.push_back(v);
        vec.push_back(w);
        edges.push_back(vec);
    }

    vector<int> dist(n,INT_MAX);
    dist[0]=0;
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<m;j++)
        {
            int u=edges[j][0];
            int v=edges[j][1];
            int w=edges[j][2];
            if(dist[u] != INT_MAX && dist[u]+w <
dist[v])dist[v]=dist[u]+w;
        }
    }
    bool flag=false;

    for(int j=0;j<m;j++)
        {
            int u=edges[j][0];
            int v=edges[j][1];
            int w=edges[j][2];
            if(dist[u] != INT_MAX && dist[u]+w <
dist[v]){dist[v]=dist[u]+w;flag=true;}
        }

    if(flag==true){cout<<"Negative cycle present";return 0;}

    for(int i=0;i<dist.size();i++)cout<<dist[i]<<" ";
    return 0;
}
```

Complexity:Time=O(n*m)

14.Kruskal's Algorithm || Disjoint Set || Union by Rank & Path Compression:

Sort the edges on the basis of weights and then take vertices and check parent just to avoid loop,if parent is same ignore otherwise do union of both components.

```
#include<bits/stdc++.h>
using namespace std;
int find_parent( vector<int>& parent,int node)
{
    if(parent[node]==node)return node;

    return parent[node]=find_parent(parent,parent[node]);
}

void find_union(int u,int v, vector<int> &rank,vector<int>& 
parent)
{
    //int u=find_parent(parent,u1);
    //int v=find_parent(parent,v1);

    if(rank[u] < rank[v])parent[u]=v;
    else if(rank[u] > rank[v])parent[v]=u;
    else {
        parent[u]=v;
        rank[u]++;
    }

}
bool comp(vector<int> &v1,vector<int> &v2)
{
    return v1[2]<v2[2];
}
int main()
{
    int n,m;
    cin>>n>>m;
    vector<vector<int>> edges;
    for(int i=0;i<m;i++)
    {
        vector<int> vec;
        int u,v,w;
        cin>>u>>v>>w;
        vec.push_back(u);
        vec.push_back(v);
        vec.push_back(w);

        edges.push_back(vec);
    }
    sort(edges.begin(),edges.end(),comp);
    vector<int> rank(n,0);
    vector<int> parent(n);
    for(int i=0;i<n;i++)parent[i]=i;
    int minST_wt=0;
    for(int i=0;i<m;i++)
    {
        int u=find_parent(parent,edges[i][0]);
        int v=find_parent(parent,edges[i][1]);
        int wt=edges[i][2];

        if(u!=v)
        {
            find_union(u,v,rank,parent);
            minST_wt+=wt;
        }
    }
    cout<<"hello"<<endl;
    cout<<minST_wt;
    return 0;
}
```

15.Bridges in Graph | Cut Edge:

```
#include <bits/stdc++.h>
using namespace std;
void dfs(int node, int parent, vector<int> &vis, vector<int>
&tin, vector<int> &low, int &timer, vector<int> adj[]) {
    vis[node] = 1;
    tin[node] = low[node] = timer++;
    for(auto it: adj[node]) {
        if(it == parent) continue;

        if(!vis[it]) {
            dfs(it, node, vis, tin, low, timer, adj);
            low[node] = min(low[node], low[it]);
            if(low[it] > tin[node]) {
                cout << node << " " << it << endl;
            }
        } else {
            low[node] = min(low[node], tin[it]);
        }
    }
}
int main() {
    int n, m;
    cin >> n >> m;
        vector<int> adj[n];
        for(int i = 0;i<m;i++) {
            int u, v;
            cin >> u >> v;
            adj[u].push_back(v);
            adj[v].push_back(u);
        }

        vector<int> tin(n, -1);//time of insertion
        vector<int> low(n, -1);
        vector<int> vis(n, 0);
        int timer = 0;
        for(int i = 0;i<n;i++) {//multiple components
            if(!vis[i]) {
                dfs(i, -1, vis, tin, low, timer, adj);
            }
        }

        return 0;
```

```
}
```

Complexity:Time=O(n+m)

16.Bipartite Graph:

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

bool bipartiteDfs(int node, vector<int> adj[], int color[]) {
    for(auto it : adj[node]) {
        if(color[it] == -1) {
            color[it] = 1 - color[node];
            if(!bipartiteDfs(it, adj, color)) {
                return false;
            }
        } else if(color[it] == color[node]) return false;
    }
    return true;
}
bool checkBipartite(vector<int> adj[], int n) {
    int color[n];
    memset(color, -1, sizeof color);
    for(int i = 0;i<n;i++) {
        if(color[i] == -1) {
            color[i] = 1;
            if(!bipartiteDfs(i, adj, color)) {
                return false;
            }
        }
    }
    return true;
}
void addedge(vector<int> adj[],int u,int v)
{
        adj[u].push_back(v);
        adj[v].push_back(u);
}
int main() {

    vector<int> adj[6];
    addedge(adj,0,1);
    addedge(adj,1,2);
    addedge(adj,1,4);
    addedge(adj,1,5);
    addedge(adj,2,3);
    addedge(adj,3,4);
    addedge(adj,3,5);

    if(checkBipartite(adj, 6)) {
        cout << "It is a Bipartite Graph";
    } else {
        cout << "It is not a Bipartite Graph";
    }
    return 0;
}
```

17.Strongly connected components:

Step 1:Topological sort on the basis of finishing time.

Step2:transpose of graph.

step3:Rev DFS

```
#include <bits/stdc++.h>
using namespace std;
void dfs(int node, stack<int> &st, vector<int> &vis,
vector<int> adj[]) {
    vis[node] = 1;
    for(auto it: adj[node]) {
        if(!vis[it]) {
            dfs(it, st, vis, adj);
        }
    }

    st.push(node);
}
void revDfs(int node, vector<int> &vis, vector<int>
transpose[]) {
    cout << node << " ";
    vis[node] = 1;
    for(auto it: transpose[node]) {
        if(!vis[it]) {
            revDfs(it, vis, transpose);
        }
    }
}
int main() {
    int n=6, m=7;
        vector<int> adj[n+1];
        adj[1].push_back(3);
        adj[2].push_back(1);
```

```
adj[3].push_back(2);
adj[3].push_back(5);
adj[4].push_back(6);
adj[5].push_back(4);
adj[6].push_back(5);

stack<int> st;
vector<int> vis(n+1, 0);
for(int i = 1;i<=n;i++) {
    if(!vis[i]) {
        dfs(i, st, vis, adj);
    }
}

vector<int> transpose[n+1];

for(int i = 1;i<=n;i++) {
    vis[i] = 0;
    for(auto it: adj[i]) {
        transpose[it].push_back(i);
    }
}



while(!st.empty()) {
    int node = st.top();
    st.pop();
```

```
            if(!vis[node]) {
                cout << "SCC: ";
                revDfs(node, vis, transpose);
                cout << endl;
            }
        }


        return 0;
}
```