Graph types:

- Directed
- Undirected

Nodes/Vertex: represents by number

Edge: connection between two nodes

- Directed edge
- Undirected edge

When will we call something Graph ?

- If it has nodes and edge

Is BINARY TREE graph?

- Yes, it is.

Directed/Undirected Graph:

- Cyclic Graph: if there's at least one cycle on the graph.. cycle means ze node theke shoru hobe abar ghure sei node e asbe.
- Acyclic Graph

Different between Graph & Tree ?

- Tree is kinda a graph which doesn't contain a cycle.

Path:

- The sequence of nodes to reach from one node to another.. Such as, 1 2 3 5..1 to 2 to 3 to 5…
- A node can't be appeared twice.. suppose, 1 2 3 2 1 is not a path, because there have multiple same nodes.
- Suppose, theres no edge between 1 and 3.. so 1 3 5 is not a valid path..

Degree of graph:

For Undirected Graph:

The degree of a node in undirected graph is how many edges connected with the node.

The total degree of the Graph = 2*Total number of Edge

Why do we multiple twice ? – Every edge has two direction in a undirected graph.

For directed graph:

-indegree : direction towards the node

-outdegree : direction opposite of the node

How to code for a graph?

- Using matrix

```
Int node, edge;
Int arr[note+1][edge+1];
Arr[..][..] = 0;
For(i<=edge)
        Cin x,y;
        //for undirected graph
        Arr[i][j] = 1;
        Arr[j][i] = 1;
        //For directed graph
        Arr[i][j] = 1;
```

- Using list

```
Int vector<int> adj[edge+1];

//undirected

Adj[i].push(j);

Adj[j].push(i);

//directed

Adj[i].push(j);
```

Connected Components:

- Different graphs in a single graph are called components of the single graph.

Breadth First Search(BFS):

Also known as level wise.

Only 1 vertex can be in level 0.

```
int n, e;
  cin >> n >> e;
```

```
    vector<int> adj[n];
    int i, j;

    for (int i = 0; i < e; i++)
    {
        cin >> i >> j;
        adj[i].push_back(j);
        adj[j].push_back(i);
    }

    int vis[n] = {0};
    vis[0] = 1;
    queue<int> q;
    q.push(0);
    vector<int> bfs;

    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        bfs.push_back(node);
        for (auto l : adj[node])
        {
            if (vis[l] != 1)
            {
                vis[l] = 1;
                q.push(l);
            }
        }
    }

    for (auto i : bfs)
        cout << i << " ";
```

Depth First Search:

```
void dfs(int node, vector<int> adj[], int vis[], vector<int> &ans)
{
    vis[node] = 1;
    ans.push_back(node);
    for (auto newNode : adj[node])
```

```
    {
        if (vis[newNode] != 1)
            dfs(newNode, adj, vis, ans);
    }
}
```

ShortCut Code for traverse a [row][col] neighbors of horizontally, vertically or diagonally..

for(r =-1 to 1)

for(c=-1 to 1)

n_row = row+r;

n_col= col+c;

Shortcut for only horizontally and vertically

```
    for (int x = -1; x <= 1; x++)
    {
        for (int y = -1; y <= 1; y++)
        {
            int newI = i + x;
            int newJ = j + y;

            if (x != y && (-x) != y && (-y) != x)
            {

            }
        }
    }
```

Flood Fill Algorithm

```
void floodFill(int i, int j, vector<vector<int>> &arr, int initVal, int newColor)
{
    arr[i][j] = newColor;
    for (int x = -1; x <= 1; x++)
    {
        for (int y = -1; y <= 1; y++)
        {
            int newI = i + x;
```

```
            int newJ = j + y;

            if (x != y && (-x) != y && (-y) != x && newI >= 0 && newI <
arr.size() && newJ >= 0 && newJ < arr[0].size() && arr[newI][newJ] == initVal)
            {

                floodFill(newI, newJ, arr, initVal, newColor);
            }
        }
    }
}
```

Check cycle in a graph using BFS

```
#include <bits/stdc++.h>
using namespace std;
// 1 base indexing
bool checkCycle(int src, vector<int> adj[], int vis[])
{
    vis[src] = 1;
    queue<pair<int, int>> q;
    q.push({src, -1});
    while (!q.empty())
    {
        int node = q.front().first;
        int prevNode = q.front().second;
        q.pop();
        for (auto l : adj[node])
        {
            if (vis[l] == 1 && prevNode != l)
            {

                return true;
            }
            if (vis[l] != 1)
            {
                vis[l] = 1;

                q.push({l, node});
            }
        }
    }
    return false;
```

```
}
int main()
{
    int r, e;
    cin >> r >> e;
    vector<int> adj[r + 1];
    int x, y;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    int vis[r + 1] = {0};
    for (int i = 1; i <= r; i++)
    {

        cout << endl;
        if (vis[i] != 1)
        {
            if (checkCycle(i, adj, vis))
            {
                cout << "Cycle Found";
                return 0;
            }
        }
    }

    cout << "NO CYCLE FOUND" << endl;
}
```

Count the island

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

void dfs(int i, int j, vector<vector<char>> &grid, vector<vector<int>> &vis)
{
    vis[i][j] = 1;
    for (int deltaRow = -1; deltaRow <= 1; deltaRow++)
    {
        for (int deltaCol = -1; deltaCol <= 1; deltaCol++)
```

```
        {
            int nborRow = i + deltaRow;
            int nborCol = j + deltaCol;
            if (nborRow >= 0 && nborRow < grid.size() && nborCol >= 0 && nborCol
< grid[0].size() && grid[nborRow][nborCol] == '1' && vis[nborRow][nborCol] == 0)
            {
                // cout << nborRow << " " << (nborCol >= 0) << " " <<
vis[nborRow][nborCol] << endl;
                dfs(nborRow, nborCol, grid, vis);
            }
        }
    }
}

int main()
{
    int row, col;
    cin >> row >> col;
    vector<vector<char>> grid(row, vector<char>(col, '0'));
    char x;
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {

            cin >> x;
            grid[i][j] = x;
        }
    }
    vector<vector<int>> vis(row, vector<int>(col, 0));
    int cnt = 0;

    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {

            if (vis[i][j] == 0 && grid[i][j] == '1')
            {
                //  cout << i << " " << j << " " << vis[i][j] << endl;

                cnt++;
                dfs(i, j, grid, vis);
            }
        }
```

```
    }

    cout << cnt;
}
```

Bipartite Graph:

Definition: The graph can be colored **by two colors** such a way that **the adjacent nodes and the node** don't contain the same color.

Linear Graphs with 0 cycle are always bipartite graph.

Any graph **with even cycle length** can also be bipartite.

USING BFS:

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

bool checkBiPartial(int i, int n, vector<int> adj[], vector<int> &colored)
{

    queue<int> q;
    q.push(i);
    colored[i] = 0;
    while (!q.empty())
    {
        int node = q.front();
        q.pop();
        for (auto l : adj[node])
        {
            if (colored[l] == colored[node])
            {
                return false;
            }

            if (colored[l] == -1)
            {
                colored[l] = (colored[node] == 0 ? 1 : 0);
                q.push(l);
            }
        }
```

```
    }
    return true;
}

int main()
{

    int node, edge;
    cin >> node >> edge;
    vector<int> adj[node];
    int x, y;
    for (int i = 0; i < node; i++)
    {
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
    vector<int> colored(node, -1);

    for (int i = 0; i < node; i++)
    {
        if (colored[i] == -1)
        {
            if (checkBiPartial(i, node, adj, colored) == false)
            {
                cout << "NO BIPARTITE";
                return 0;
            }
        }
    }
    cout << " BIPARTITE";

    return 0;
}
```

Using DFS:

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

bool dfs(int i, int i_color, int node, vector<int> adj[], vector<int> &colored)
{
    colored[i] = i_color;
```

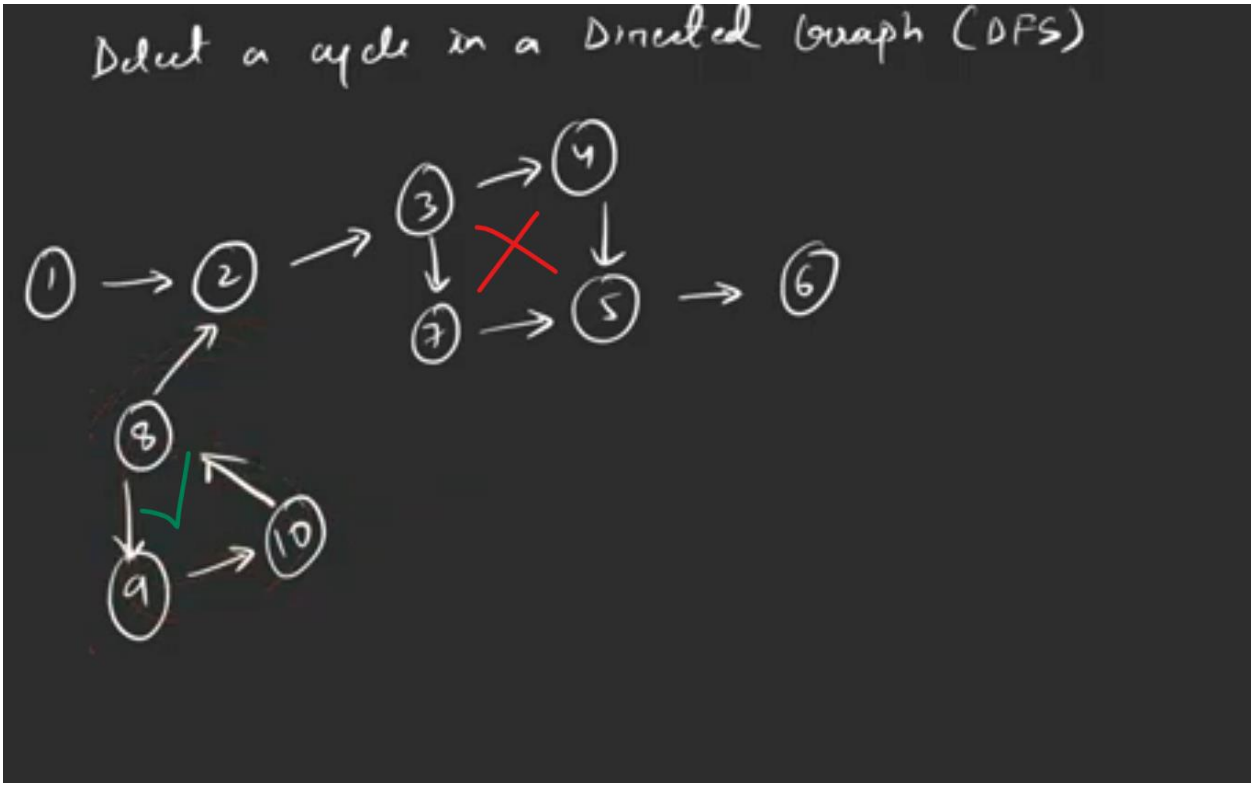
```
    for (auto l : adj[i])
    {
        if (colored[l] == i_color)
            return false;
        else if (colored[l] == -1)
        {
            dfs(l, i_color == 0 ? 1 : 0, node, adj, colored);
        }
    }
    return true;
}

int main()
{
    int node, edge;
    cin >> node >> edge;
    vector<int> adj[node];
    int x, y;
    for (int i = 0; i < edge; i++)
    {
        cin >> x >> y;
        adj[x].push_back(y);
        adj[y].push_back(x);
    }

    vector<int> colored(node, -1);
    bool ans;
    for (int i = 0; i < node; i++)
    {
        if (colored[i] == -1)
            if (dfs(i, 0, node, adj, colored) == false)
            {
                cout << "NOT BIPARTITE";

                return 0;
            }
    }
    cout << " BIPARTITE";
    return 0;
}
```

Check Cycle exists or not using DFS:




```cpp
#include <bits/stdc++.h>
using namespace std;
// DirectedGraph, 0 base indexing graph

bool dfs(int i, vector<int> &vis, vector<int> &pathVis, vector<int> adj[])
{
    vis[i] = 1;
    pathVis[i] = 1;
    for (auto l : adj[i])
    {
        if (vis[l] == 0)
        {
            dfs(l, vis, pathVis, adj);
        }
```

```
        else if (vis[l] == 1 && pathVis[l] == 1)
        {
            return true;
        }
    }
    pathVis[i] = 0;
    return false;
}

int main()
{
    int node, edge;
    cin >> node >> edge;
    vector<int> adj[node];
    int x, y;

    for (int i = 0; i < edge; i++)
    {

        cin >> x >> y;

        adj[x].push_back(y);
    }
    vector<int> vis(node, 0);
    vector<int> pathVis(node, 0);

    for (int i = 0; i < node; i++)
    {
        if (dfs(i, vis, pathVis, adj) == true)
        {
            cout << "Cycle found";
            return 0;
        }
    }
    cout << "Not Found";
    return 0;
}
```

Tropological Sorting

Definition: if there's an edge between u and v, u appears before v in linear ordering.

- Only applicable in DAG (Directed Acyclic Graph)

Using DFS:

```cpp
#include <bits/stdc++.h>
using namespace std;
// It's applicable in DAG: Directed Acyclic Graph

void dfs(int i, vector<int> adj[], vector<int> &vis, stack<int> &st)
{
    vis[i] = 1;
    for (auto l : adj[i])
    {
        if (vis[l] == 0)
            dfs(l, adj, vis, st);
    }
    st.push(i);
}

int main()
{

    int n, e;
    cin >> n >> e;
    vector<int> adj[n];
    int x, y;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y;
        adj[x].push_back(y);
    }
    vector<int> vis(n, 0);
    stack<int> st;
    for (int i = 0; i < n; i++)
    {
        if (vis[i] != 1)
            dfs(i, adj, vis, st);
    }
```

```
-       while (!st.empty())
-       {
-           cout << st.top() << " ";
-           st.pop();
-       }
-   }
```

Using BFS:

- Insert all nodes which have indegree 0
- Take them out of the queue and reduce 1 of adj nodes

```
-   #include <bits/stdc++.h>
-   using namespace std;
-
-   int main()
-   {
-       int n, e;
-       cin >> n >> e;
-       vector<int> adj[n];
-       vector<int> indegree(n, 0);
-       int x, y;
-       for (int i = 0; i < e; i++)
-       {
-           cin >> x >> y;
-           adj[x].push_back(y);
-           indegree[y]++;
-       }
-       queue<int> q;
-       for (int i = 0; i < n; i++)
-       {
-           if (indegree[i] == 0)
-               q.push(i);
-       }
-       vector<int> ans;
-       while (!q.empty())
-       {
-           int node = q.front();
-           q.pop();
-           ans.push_back(node);
```

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

Directed Graph Cyclic or Not Cyclic Check by BFS/TOPOSORTING :

```
#include <bits/stdc++.h>
using namespace std;
// Concept : Topo sorting is not possible in cyclic graph.
int main()
{
    int n, e;
    cin >> n >> e;
    vector<int> adj[n];
    vector<int> ans;
    vector<int> indegree(n, 0);
    int x, y;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y;
        adj[x].push_back(y);
        indegree[y]++;
    }
    queue<int> q;
    for (int i = 0; i < n; i++)
    {
        if (indegree[i] == 0)
        {
            q.push(i);
        }
    }

    while (!q.empty())
```

```
    {
        int node = q.front();
        q.pop();
        ans.push_back(node);
        for (auto l : adj[node])
        {
            indegree[l]--;
            if (indegree[l] == 0)
                q.push(l);
        }
    }
    if (ans.size() == n)
        cout << "YES";
    else
        cout << "NO";
}
```

Alien Dictionary:

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

string topologicalSort(int k, vector<int> adj[])
{
    vector<int> indegree(k, 0);
    for (int i = 0; i < k; i++)
    {
        for (auto l : adj[i])
        {
            indegree[l]++;
        }
    }

    queue<int> q;
    for (int i = 0; i < k; i++)
    {
        if (indegree[i] == 0)
            q.push(i);
    }
    vector<int> topo;
    while (!q.empty())
    {
```

```
        int node = q.front();
        q.pop();
        topo.push_back(node);
        for (auto l : adj[node])
        {
            indegree[l]--;
            if (indegree[l] == 0)
                q.push(l);
        }
    }
    string s;
    for (auto x : topo)
    {
        s += x + 'a';
    }

    return s;
}

int main()
{
    int n, k;
    cin >> n >> k;
    string s[n];
    for (int i = 0; i < n; i++)
    {
        cin >> s[i];
    }

    vector<int> adj[k];
    for (int i = 0; i < n - 1; i++)
    {
        string s1 = s[i];
        string s2 = s[i + 1];
        int len = min(s1.length(), s2.length());
        for (int j = 0; j < n; j++)
        {
            if (s1[j] != s2[j])
            {
                adj[s1[j] - 'a'].push_back(s2[j] - 'a');
                break;
            }
        }
    }
    cout << topologicalSort(k, adj);
```

```
}
```

Shortest path of DAG using TopoSoring

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

void topoSort(int i, stack<int> &s, vector<pair<int, int>> adj[], vector<int>
&vis)
{
    vis[i] = 1;

    for (auto l : adj[i])
    {
        if (vis[l.first] != 1)
            topoSort(i, s, adj, vis);
    }
    s.push(i);
}

int main()
{
    int n, e;
    cin >> n >> e;
    vector<pair<int, int>> adj[n];
    int x, y, z;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y >> z;
        adj[x].push_back({y, z});
    }
    stack<int> s;
    vector<int> vis(n, 0);
    for (int i = 0; i < n; i++)
    {
        if (vis[i] != 1)
            topoSort(i, s, adj, vis);
    }

    vector<int> dis(n, INT_MAX);
    dis[0] = 0;
    while (!s.empty())
    {
```

```
        int node = s.top();
        s.pop();

        for (auto l : adj[node])
        {
            int i = l.first;
            int j = l.second;
            dis[i] = min(dis[i], j + dis[node]);
        }
    }
}
```

Shortest path of UNDIRECTED GRAPH using topoSort

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

int main()
{
    int n, e;
    cin >> n >> e;
    vector<int> adj[n];
    int x, y;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y;
        adj[x].push_back(y);
    }

    queue<pair<int, int>> q;
    vector<int> vis(n, 0);
    q.push({0, 0});
    vis[0] = 1;
    vector<int> dis(n, 0);
    while (!q.empty())
    {
        int node = q.front().first;
        int d = q.front().second;
        q.pop();
        for (auto l : adj[node])
        {
            if (vis[l] != 1)
```

```
            {
                vis[l] = 1;
                q.push({l, d + 1});
                dis[l] = d + 1;
            }
        }
    }

    for (auto x : dis)
    {
        cout << x << " ";
    }
}
```

Dijkstra

- Single source shortest path
- Relaxation : Updating the minimum shortest path when you get new one
- Doesn’t work in negative graph.

প্রায়োরিটি কিউঃ এটি মূলত একটি min heap। এখান থেকে সবচেয়ে কম দুরত্বের নোডটা বের করে নেওয়া যায়। এবং এটি ব্যবহার করার কারণ হল এটি কাজ করে O(log n)-এ। যেখানে সাধারণ লুপ চালিয়ে করলে লাগতো O(n)। প্রায়োরিটি কিউ ব্যবহার করার কারণ ব্যাখ্যা করি নাই। তবে মোটামুটি একটা বড় গ্রাফ নিয়ে হাতে কাজ করলেই বুঝতে পারবে যে, কিউ ব্যবহার করলে যতবার আপডেট করতে হচ্ছে, প্রায়োরিটি কিউ ব্যবহার করলে তার চেয়ে অনেক কম আপডেটেই কাজ হয়ে যাবে।

পাথ রিলাক্সেশন (Path Relaxation): যখন কোনো একটি এজ ব্যবহারের কারণে কোনো নোডের দুরত্ব কমে, আমরা শুধুতখনই ওই এজটি ব্যবহার করছি। এ ধারণাকে বলা হয় পাথ রিলাক্সেশন

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

void dijsktra(int s, vector<int> &dis, priority_queue<pair<int, int>,
vector<pair<int, int>>, greater<pair<int, int>>> &pq, vector<pair<int, int>>
adj[])
{
```

```
    dis[s] = 0;
    pq.push({0, s});

    while (!pq.empty())
    {
        int curDis = pq.top().first;
        int node = pq.top().second;
        pq.pop();
        if (dis[node] > curDis)
            continue;
        for (auto l : adj[node])
        {

            if ((l.second + curDis) < dis[l.first])
            {

                pq.push({l.second + curDis, l.first});
                dis[l.first] = l.second + curDis;
            }
        }
    }
}

int main()
{
    int n, e, s;
    cin >> n >> e >> s;
    vector<pair<int, int>> adj[n];
    int x, y, wt;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y >> wt;
        adj[x].push_back({y, wt});
    }

    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
    vector<int> dis(n, INT_MAX);

    dijsktra(s, dis, pq, adj);
    for (auto l : dis)
    {
        cout << l << " ";
    }
}
```

Bellman Ford

- Single source shortest path
- Work on only DG
- Can be used negative weight
- Doesn’t work on Negative Cycle

NEGATIVE CYCLE : Ze cycle e ekbar ghure asle total cost (total weight) er value negative hoy tai negative cycle. :D

THE DISADVANGAES OF NEGATIVE CYCLE: You can’t never ever find a shortest path . WHY ? Because, the distance reduce in every cycle..

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

int main()
{
    int node, edges, source;
    cin >> node >> edges >> source;
    vector<pair<int, pair<int, int>>> adj;
    int x, y, wt;
    for (int i = 0; i < edges; i++)
    {
        cin >> x >> y >> wt;
        adj.push_back({x, {y, wt}});
    }
    vector<int> dis(node, INT_MAX);
    dis[source] = 0;
    for (int i = 1; i <= node - 1; i++)
    {
        for (auto l : adj)
        {
```

```
            if (dis[l.first] + l.second.second < dis[l.second.first])
                dis[l.second.first] = dis[l.first] + l.second.second;
        }
    }
    int f = 0;
    for (auto l : adj)
    {
        if (dis[l.first] + l.second.second < dis[l.second.first])
        {
            cout << "NEGATIVE CYCLE";
            f = 1;
            break;
        }
    }

    if (f == 0)
    {
        for (auto x : dis)
            cout << x << " ";
    }
}
```

Floyd Warshall Algorithm

(All pair shortest path)

TASK : It can determine the shortest distance of all nodes from each node.

Concept: matrix[i][j] = min(matrix[i][j] , matrix[i][k] + matrix[k][j])

TC : n^3

```
#include <bits/stdc++.h>
using namespace std;
const int INF = 1e9 + 10;
int main()
{
    int n, e;
    cin >> n >> e;
    int adj[n][n];
    int x, y, wt;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            adj[i][j] = i == j ? 0 : INF;
    for (int i = 0; i < e; i++)
    {
        cin >> x >> y >> wt;
        adj[x][y] = wt;
    }

    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (adj[i][j] != INF || (adj[i][k] != INF && adj[k][j] != INF))
                    adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (adj[i][j] == INF)
            {
                cout << "I ";
            }
            else
            {
                cout << adj[i][j] << " ";
```

```
            }
        }
        cout << endl;
    }
}
```

Disjoint Set union (DSU)

- Make -> creating new node
- Find -> Find the root/parent of the group
- Union -> union two node

```
#include <bits/stdc++.h>
using namespace std;
#define MAX 100
int parent[MAX];
int Size[MAX];

void make(int v)
{
    parent[v] = 1;
    Size[v] = 1;
}

int find(int v)
{
    if (v == parent[v])
        return v;
    return parent[v] = find(parent[v]);
}
void Union(int a, int b)
{
     a = find(a);
   b = find(b);
    if (a != b)
    {
        if (Size[a] < Size[b])
            swap(a, b);
        parent[b] = a;
        Size[a] += Size[b];
    }
}

int main()
{
}
```

Kruskal's Algorithm

- Determine the minimum spanning tree
- Start from minimum edge
- Don't take the edge if it creates a loop(Condition: parent will be same)

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

int parent[1000];
int Size[1000];

void make(int i)
{
    parent[i] = i;
    Size[i] = i;
}

int find(int a)
{
    if (parent[a] == a)
        return a;
    return parent[a] = find(parent[a]);
}

void Uniod(int a, int b)
{
    a = find(a);
    b = find(b);
    if (a != b)
    {
        if (Size[a] < Size[b])
            swap(a, b);
        parent[b] = a;
        Size[a] += Size[b];
    }
}

int main()
{
    int m, n;
    cin >> m >> n;
    vector<pair<int, pair<int, int>>> edges;
```

```
    int x, y, wt;
    for (int i = 0; i < n; i++)
    {
        cin >> x >> y >> wt;
        edges.push_back({wt, {x, y}});
    }
    for (int i = 1; i <= n; i++)
        make(i);
    sort(edges.begin(), edges.end());
    int total_cost = 0;
    for (auto &edge : edges)
    {
        wt = edge.first;
        x = edge.second.first;
        y = edge.second.second;
        if (find(x) == find(y))
            continue;
        Uniod(x, y);
        total_cost += wt;
    }
    cout << total_cost;
}
```