**Depth first search(dfs)**

**Time complexity of the Depth first search**

**Time complexity ,O (V+E) when implemented using the adjacency list.**

**Using recursion**

**bool visited[sz];**

**vector<int>adj[sz];**

**void dfs(int node)**

**{**

**visited[node]=true;**

**for (auto childnode:adj[node])**

**{**

**if (!visited[childnode])**

**{**

**dfs(childnode);**

**}**

**}**

**}**

**Using stack**

**bool visited[sz];**

**vector<int>adj[sz];**

**void dfs(int node)**

**{**

**stack<int>st;**

**st.push(node);**

**while(!st.empty())**

**{**

**int p=st.top();**

**st.pop();**

**vis[p]=1;**

**for (int i=0;i<adj[p].size();i++)**

**{**

**int ch=adj[p][i];**

**if (!vis[ch])**

**{**

**vis[ch]=1;**

**st.push(ch);**

**}**

**}**

**}**

**}**

**Breadth first search(bfs)**

**Time complexity of the Breadth first search**

**The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges .**

**int id[sz];**

**const int sz=1e5+1;**

**const int OO= 0x3f3f3f3f;**

**int dis[sz];**

**vector<int>adj[sz];**

**void bfs(int node)**

**{**

**dpp(dis,OO);**

**queue<int>q;**

**q.push(node);**

**dis[node]=1;**

**while(!q.empty())**

**{**

**int p=q.front();**

**q.pop();**

**for (int i=0;i<adj[p].size();i++)**

**{**

**int ch=adj[p][i];**

**if (dis[ch]==OO)**

**{**

**dis[ch]=dis[p]+1;**

**q.push(ch);**

**}**

**}**

**}**

**}**

**Disjoint set**

**Time complexity of the disjoint set**

**O(Mα(N))**

**When you use the weighted-union operation with path compression it takes log \* N for each union find operation, where N is the number of elements in the set.**

**log \*N is the iterative function that computes the number of times you have to take the log of N before the value of N reaches 1.**

**α(N): is a slowly growing function <=5 which make the over all complexity O(5\*M)**

**where m is the number of operations n is the number of elements**

**int id[sz];**

**void disjoint\_intialize()**

**{**

**for (int i=0;i<sz;i++)**

**{**

**id[i]=i;**

**}**

**}**

**int root(int x)**

**{**

**while(id[x]!=x)**

**{**

**id[x]=id[id[x]];**

**x=id[x];**

**}**

**return x;**

**}**

**void disjoint\_union(int x,int y)**

**{**

**int p=root(x);**

**int q=root(y);**

**id[p]=q;**

**}**

**bool disjoint\_find(int x,int y) // if both has the same root they are connected and there's cycle here**

**{**

**return (root(x)==root(y));**

**}**

**Kruskal’s Algorithm (minimum spanning tree)**

**Time complexity of the Kruskal’s Algorithm**

**In Kruskal’s algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(E log V), which is the overall Time Complexity of the algorithm .**

**int id[sz];**

**void disjoint\_intialize()**

**{**

**for (int i=0;i<sz;i++)**

**{**

**id[i]=i;**

**}**

**}**

**int root(int x)**

**{**

**while(id[x]!=x)**

**{**

**id[x]=id[id[x]];**

**x=id[x];**

**}**

**return x;**

**}**

**void disjoint\_union(int x,int y)**

**{**

**int p=root(x);**

**int q=root(y);**

**id[p]=q;**

**}**

**bool disjoint\_find(int x,int y)**

**{**

**return (root(x)==root(y));**

**// if both has the same root they are connected and there's cycle here**

**}**

**int kurksal\_algorithm(vector<pair<int,pii>>&adj)**

**{**

**sort(adj.begin(),adj.end());**

**ll minicost=0;**

**for (int i=0;i<adj.size();i++)**

**{**

**int x=adj[i].S.F;**

**int y=adj[i].S.S;**

**int cost=adj[i].F;**

**if (!disjoint\_find(x,y))**

**{**

**disjoint\_union(x,y);**

**minicost+=cost;**

**}**

**}**

**return minicost;**

**}**

**int main()**

**{**

**//myf.open("file.txt");**

**//freopen("task.in", "r", stdin);**

**//freopen("output.txt", "w", stdout);**

**ios\_base::sync\_with\_stdio(0),cin.tie(0),cout.tie(0);**

**int n,m;**

**cin>>n>>m;**

**vector<pair<int,pair<int,int>>>adj;**

**for (int i=0;i<m;i++)**

**{**

**ll x,y,w;**

**cin>>x>>y>>w;**

**adj.push\_back({w,{x,y}});**

**}**

**disjoint\_intialize();**

**cout<<kurksal\_algorithm(adj)<<endl;**

**return 0;**

**}**

****