**Kruskal’s Algorithm**

Kruskal’s Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

**Algorithm Steps:**

* Sort the graph edges with respect to their weights.
* Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
* Only add edges which doesn't form a cycle , edges which connect only disconnected components.

So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of O(V+E) where V is the number of vertices, E is the number of edges. So the best solution is **"Disjoint Sets":**  
Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>

using namespace std;

const int MAX = 1e4 + 5;

int id[MAX], nodes, edges;

pair <long long, pair<int, int> > p[MAX];

void initialize()

{

for(int i = 0;i < MAX;++i)

id[i] = i;

}

int root(int x)

{

while(id[x] != x)

{

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

x = id[x];

}

return x;

}

void union1(int x, int y)

{

int p = root(x);

int q = root(y);

id[p] = id[q];

}

long long kruskal(pair<long long, pair<int, int> > p[])

{

int x, y;

long long cost, minimumCost = 0;

for(int i = 0;i < edges;++i)

{

// Selecting edges one by one in increasing order from the beginning

x = p[i].second.first;

y = p[i].second.second;

cost = p[i].first;

// Check if the selected edge is creating a cycle or not

if(root(x) != root(y))

{

minimumCost += cost;

union1(x, y);

}

}

return minimumCost;

}

int main()

{

int x, y;

long long weight, cost, minimumCost;

initialize();

cin >> nodes >> edges;

for(int i = 0;i < edges;++i)

{

cin >> x >> y >> weight;

p[i] = make\_pair(weight, make\_pair(x, y));

}

// Sort the edges in the ascending order

sort(p, p + edges);

minimumCost = kruskal(p);

cout << minimumCost << endl;

return 0;

}

**Prim’s Algorithm**

Prim’s Algorithm also use Greedy approach to find the minimum spanning tree. In Prim’s Algorithm we grow the spanning tree from a starting position. Unlike an **edge** in Kruskal's, we add **vertex** to the growing spanning tree in Prim's.

**Algorithm Steps:**

* Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
* Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
* Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.
* #include <iostream>
* #include <vector>
* #include <queue>
* #include <functional>
* #include <utility>
* using namespace std;
* const int MAX = 1e4 + 5;
* typedef pair<long long, int> PII;
* bool marked[MAX];
* vector <PII> adj[MAX];
* long long prim(int x)
* {
* priority\_queue<PII, vector<PII>, greater<PII> > Q;
* int y;
* long long minimumCost = 0;
* PII p;
* Q.push(make\_pair(0, x));
* while(!Q.empty())
* {
* // Select the edge with minimum weight
* p = Q.top();
* Q.pop();
* x = p.second;
* // Checking for cycle
* if(marked[x] == true)
* continue;
* minimumCost += p.first;
* marked[x] = true;
* for(int i = 0;i < adj[x].size();++i)
* {
* y = adj[x][i].second;
* if(marked[y] == false)
* Q.push(adj[x][i]);
* }
* }
* return minimumCost;
* }
* int main()
* {
* int nodes, edges, x, y;
* long long weight, minimumCost;
* cin >> nodes >> edges;
* for(int i = 0;i < edges;++i)
* {
* cin >> x >> y >> weight;
* adj[x].push\_back(make\_pair(weight, y));
* adj[y].push\_back(make\_pair(weight, x));
* }
* // Selecting 1 as the starting node
* minimumCost = prim(1);
* cout << minimumCost << endl;
* return 0;
* }