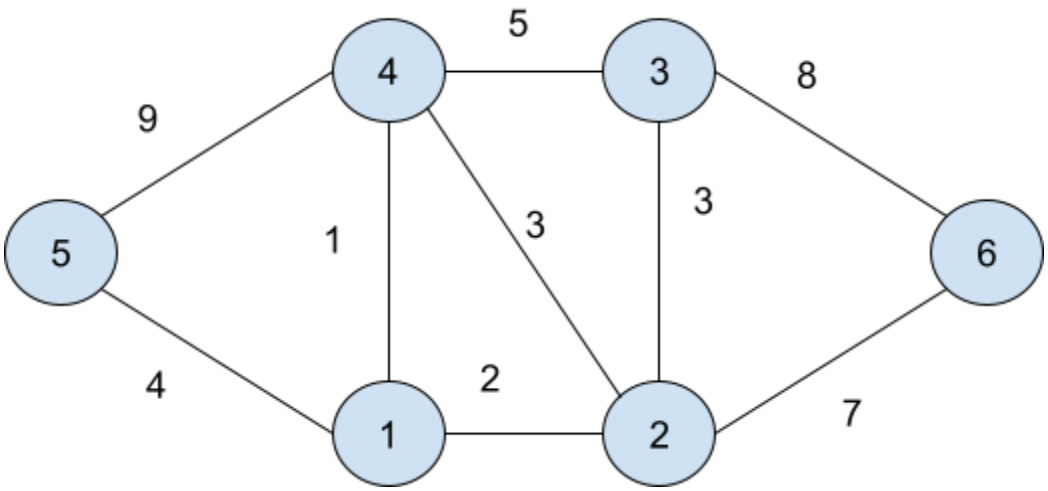
MEDIUM

Kruskal's Algorithm

Intuition

Kruskal's Algorithm is used to Find the **Minimum Spanning Tree**

eg.



If this graph can be represented in n nodes and n-1 edges there is a possible MST.

Algorithm :

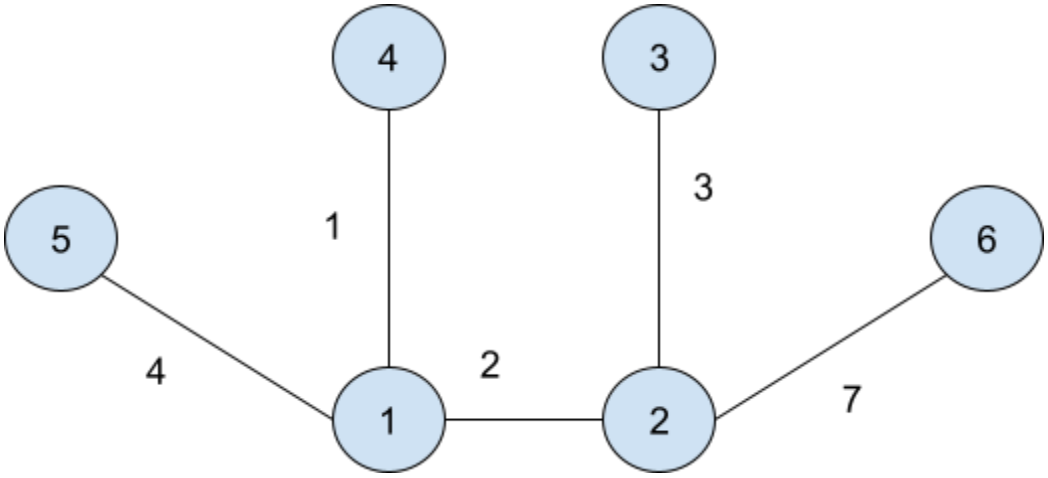
- Sort all the edges according to weights : [weight , u, v]
  - 1 1 4
  - 2 1 2
  - 3 2 3
  - 3 2 4
  - 4 1 5
  - 5 3 4
  - 7 2 6
  - 8 3 6
  - 9 4 5
- Disjoint set provides us with the following options unionByRank( ) and findParent( ), when we will create a DS using these 6 nodes we have initial configuration.

Eg.

1 1 4
1 4 // different parents so can take it

2	1	2
	1	2 // different parents so can take it
3	2	3
	1	3 // different parents so can take it
3	2	4
	1	1 // Not take it same parents
4	1	5
	1	5 // different parents so can take it
5	3	4
	1	1 // Not take it same parents
7	2	6
	1	6 // different parents so can take it
8	3	6
	1	1 // Not take it same parents
9	4	5
	1	1 // same parents not take it

MST :



Approach

- Creating an edge vector containing edges as distance, { from, to }
- Sorting the edge vector
- Creating a MSTweight variable initialized with 0
- Traversing the edge vector :
    - Check if parent of the edges are not same :
        - Update MST weight as MSTweight+=edgeweight
        - Update disjoint set with this edge
- Return the MSTweight

Function Code

```
class DisjointSet
{
    private:
    vector<int> parent;
    vector<int> rank;
    public:
    // creating constructor
    DisjointSet(int n)
    {
        // resizing rank and initializing with 0
        rank.resize(n,0);
        // resizing parent and initializing with node itself
        parent.resize(n);
        for(int i=0;i<n;i++)
        {
            parent[i]=i;
        }
    }
    // creating the find parent function
    int findParent(int node)
    {
        // checking if the node is we reached is already a parent
        if(parent[node]==node)
        {
            // returning the node
            return node;
        }
        // implementing path compression and returning recursive call to
find ultimate parent
        return parent[node] = findParent(parent[node]);
    }
    // implementing union by rank
    void unionbyrank(int u,int v)
    {
        // calculating the ultimate parents
        int pu = findParent(u);
        int pv = findParent(v);
        // checking if the parents are already same
        if(pu==pv)
        {
            return;// parents are same
        }
```

```
        // u rank is greater
        if(rank[pu]<rank[pv])
        {
            parent[pv] = pu;
            rank[pu]+=1;
        }
        else if(rank[pu]==rank[pv])
        {
            parent[pv] = pu;
        }
        else
        {
            parent[pu]=pv;
            rank[pv]+=1;
        }
    }
};
class Solution
{
      public:
      //Function to find sum of weights of edges of the Minimum Spanning
Tree.
    int spanningTree(int n, vector<vector<int>> adj[])
    {
        // We have to store the edges
        vector<pair<int, pair<int, int>>> edges;
        for(int i=0;i<n;i++)
        {
      // traversing adjacency list to get edges
            for(auto it : adj[i])
            {
                int adjnode = it[0];
                int weight = it[1];
                int node = i;
                edges.push_back({weight,{node,adjnode}});
            }
        }
        // creating a disjoint set
        DisjointSet ds(n);
        // sorting edges according to weights
        sort(edges.begin(),edges.end());
        int mstweight = 0;
        // traversing through edges to update the MST
```

```
        for(auto it:edges)
        {
            int weight = it.first;
            int from = it.second.first;
            int to = it.second.second;
      // checking if parents are different
            if(ds.findParent(from)!=ds.findParent(to))
            {
                // updating the MST weight and creating the graph using DS
                mstweight+=weight;
                ds.unionbyrank(from,to);
            }
        }
        // returning the weight of the MST
        return mstweight;
    }
};
```

Time Complexity

O(Nlog(N))