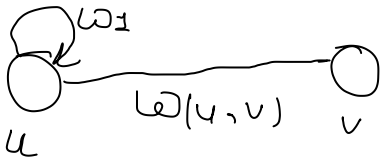
Code -

```
#include <bits/stdc++.h>
vector<int> dijkstra(vector<vector<int>> &vec, int vertices, int edges, int source) {
    // Write your code here.
    unordered_map<int, list<pair<int, int>>> adj;
    // Pushing of element
    /// 1 2 3
    for(int i=0;i<edges;i++){
        int u = vec[i][0];
        int v = vec[i][1];
        int w= vec[i][2];

        adj[u].push_back(make_pair(w,v));
        adj[v].push_back(make_pair(w,u));

    }

    /// Creating of Distance array and pushing infinity to all
    vector<int> distance(vertices);
    //unordered_map<int, int> distance;
    for(int i=0;i<vertices;i++){
        distance[i] = INT_MAX;
    }

    distance[source] = 0;
    set<pair<int,int>> s;
    s.insert(make_pair(distance[source], source));

    while(!s.empty()){
        auto first_el = *(s.begin());
        int w1 = first_el.first;
        int u = first_el.second;
        s.erase(s.begin());

        for(auto i: adj[u]){
            int w = i.first;
            int v = i.second;

            if(w1 + w < distance[v] ){
                auto record = s.find(make_pair(w,v));
                //if record found
                    if(record != s.end()){
                        s.erase(record);

                    }
                    distance[v] = w1 + w;
                    s.insert(make_pair(distance[v], v ));
            }
        }
    }

    return distance;

}
```

→ return type is vector so we need to use vector.

→ just like in queue.



⇒ distan(u) + w < distan(v)
↓ w1 ↓ (u,v) ↓ previously stored distance.

* Use of 'auto' :

↳ This automatically detects the type of a STL from its initialization & reduces complexity

Eg -

```
std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}};

// Traditional way
std::map<int, std::string>::iterator it = myMap.begin();

// Using auto
auto it = myMap.begin();
```

```
std::map<int, std::string> myMap = {{1, "one"}, {2, "two"}};

// Traditional way
std::pair<const int, std::string>& entry = *myMap.begin();

// Using auto
auto& entry = *myMap.begin();
```

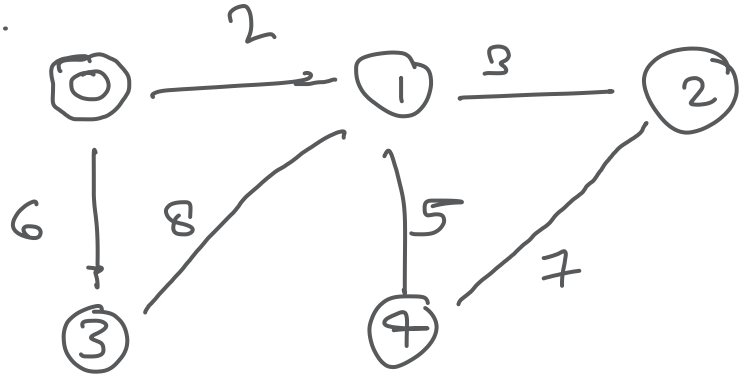
>Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge weights.
>It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956 and published three years later.
>The algorithm finds the shortest path from a given source node to all other nodes in the graph.

* lecture.12: Prim's Algo For Graph: (Minimum Spanning Tree)

>Prim's algorithm is a greedy algorithm used for finding the Minimum Spanning Tree (MST) of a weighted, undirected graph.

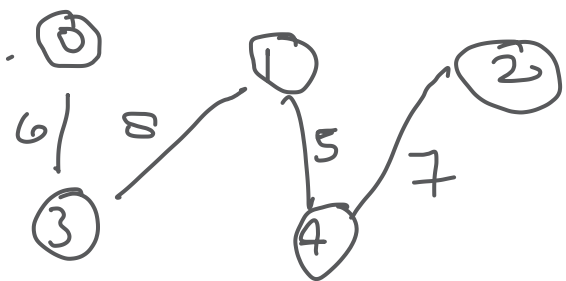
>The main objective of Prim's algorithm is to connect all the vertices of the graph with the minimum possible total edge weight, ensuring there are no cycles, and the graph remains connected.

MST: Minimum Spanning Tree (MST): A spanning tree of a graph is a subgraph that includes all the vertices with the minimum number of edges needed to keep the graph connected and without cycles. The MST is the spanning tree with the minimum total edge weight.



Spanning Tree
n nodes
(n-1) edges
Connected

- lets make a Spanning Tree from above example -



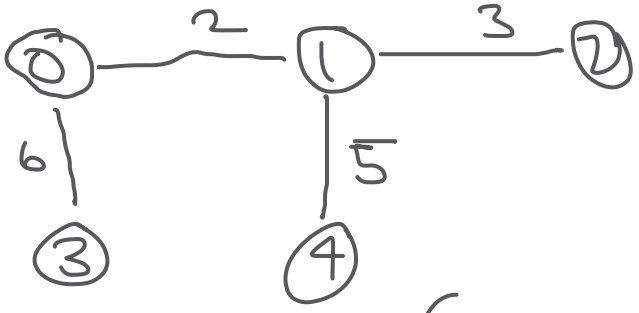
: So

Here
5: nodes (n)
: edges (n-1)

- Minimum Spanning Tree -

→ A Type of spanning Tree where sum of all edge weight is less than the all spanning Tree of same graph.

exa -



⇒ Sum of all edges = (6+2+5+3)
= 16

⇒ for the tree above =
(6+8+5+7)
= 26

→ so we can see 16 < 26, so for other as well.

- lets discuss method by -

① Create 3 DS → key, parent,

MST(visited).

(2) Put a node in key to start.

↳ set key[0] = 0

parent[0] = -1

Now _

take u as 0

mst[u] → true

adj.

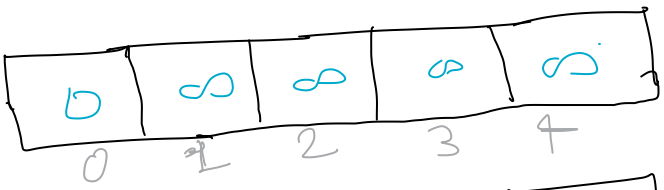
Perform if(mst[i] = false && key[i] < min)

u → i

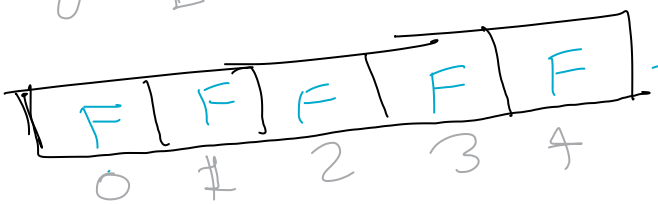
mst[u] = true

adj.

key :

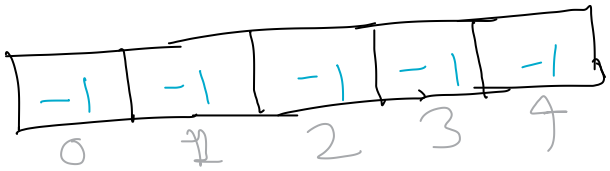


MST :



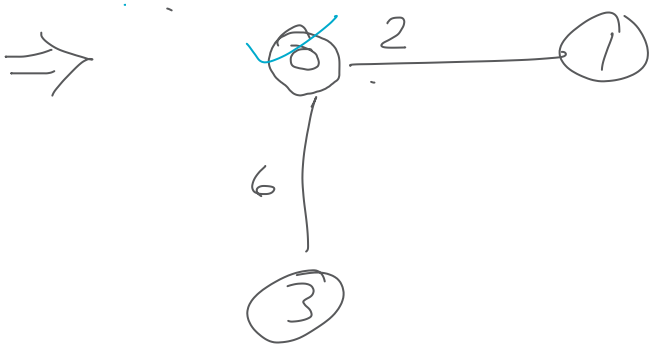
→ unvisited_map

Parent :



So —

① take that key node now find its neighbours, if any one of them is not visited



mark those visited there if

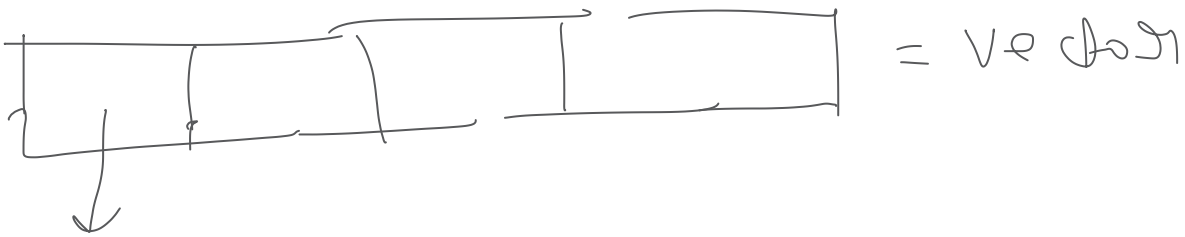
$d(u) + w < d(v)$

$d(v) = d(u) + w$

→ Parent[i] = u.

⇒ vector < pair < pair < int, int >, int >> G.

⇒



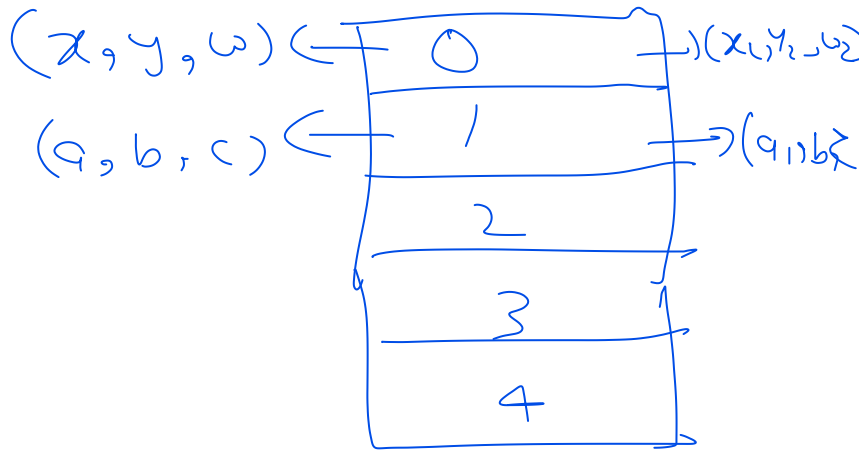
n = vertex,

m = edges;

→ adjlist :

5 1 4 :

N m



u v w

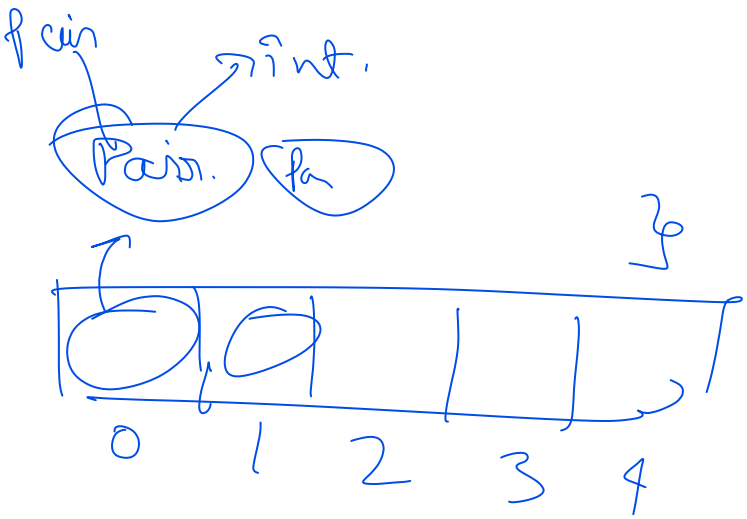
→ 1 2 2 →

→ 1 4 6

→ 2 1 2

```
for(int i=0; i<n;i++)
{
  int u =((g[i]).first).first
  int v=(g[i]).first).sec
  int w = (g[i]).second.
  adj[u].push_back(mp(v,w)
  adj[v].push_back(w,u).
}
```

make_pair



→ vector<int> key(n)

→ vector<bool> visit(n)

→ vector<int> distance(n)

→ for (int i=0; i<=n; i++)

Set key[i] = ∞;

→ visit[i] = 0;

Parent[i] = -1;

}

: It Starting of algo -

→ key[0] = 0;

→ Parent[0] = -1;

→ visit[0] = 1;

→ for(int i=0; i<n; i++)

{

int mini = INT_MAX;

int u;

// find the min node.

for(int v=1; v<n; v++)

{ if(mst[v] == false &&

key[v] < mini) {

u = v;

mini = key[v];

}

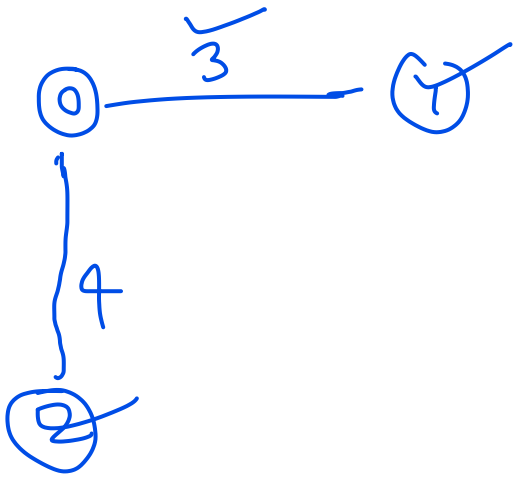
}

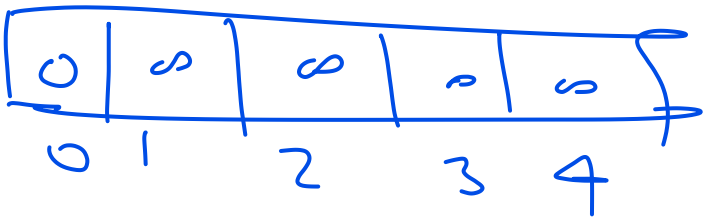
// mark min node as true;

```
mst[u] = true;
// check adjacent nodes
for (auto i : adj[u])
    v = i.first
    w = i.second;
    if(mst[v] == false && w < key[v])
    {
        parent[v] = u;
        key[v] = w;
    }
}
```

```
vector < pair<pair<int,int>,int>> result;
for(int i = 1; i < n; i++)
    result.push_back({{parent[i],i},
                      key[i]});
}
```

Return ans;





mini = ∞

for i = 0

↳ j = 0, (! visited[0] && 0 < ∞)

mini = 0;

u = 0

~~i~~ : adj[u])

if(! visited && w < key[v])

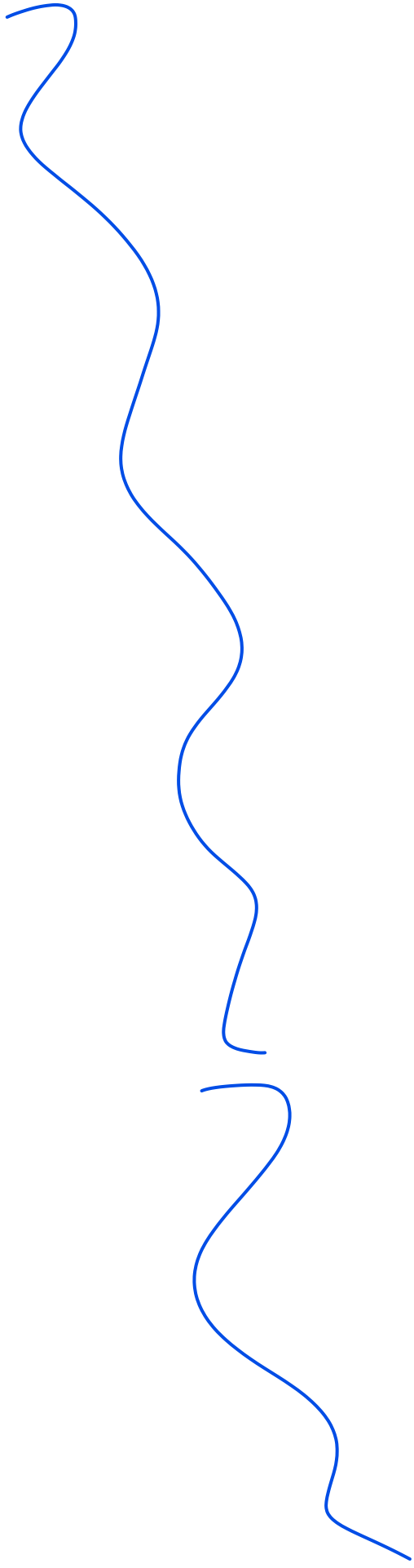
parent[v] = u;

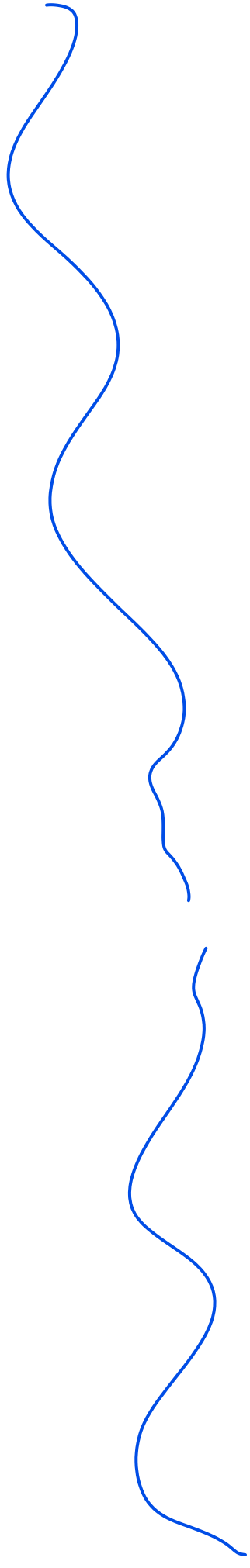
key[v] = weight;

}

}

```
#include <bits/stdc++.h>
vector<pair<pair<int, int>, int>> calculatePrimsMST(int n, int m, vector<pair<pair<int, int>, int>> &g)
{
    // Write your code here.
    unordered_map<int, list<pair<int, int>>> adj;

    // creating adj
    for(int i=0;i<m;i++){
        int u = g[i].first.first;
        int v = g[i].first.second;

        int w = g[i].second;

        adj[u].push_back(make_pair(w,v));
        adj[v].push_back(make_pair(w,u));
    }

    /// starting of algo

    vector<int> key(n);
    vector<bool> mst(n);
    vector<int> parent(n);

    for(int i=0;i<n;i++){
        key[i] = INT_MAX;
        mst[i] = false;
        parent[i] = -1;

    }

    key[0] = 0;
    parent[0] = -1;

    ///finding min node
    // we are taking (n-1), because after adding (n-1) edge, mst will be complete
    for(int i=0;i<n-1;i++){
        int mini = INT_MAX;
        int u;

        for(int j=0; j<n; j++){
            if(mst[j] == false && key[j] < mini){
                mini = key[j];
                u = j;
            }
        }

        /// mark min as true
        mst[u] = true;

        /// Check adjacent nodes
        for( auto it:adj[u]){
        int w = it.first;
        int v = it.second;

        if (mst[v] == false && w < key[v]) {
                parent[v] = u;
                key[v] = w;
            }
        }
    }

    vector<pair<pair<int,int> , int>> ans;
    // ans will not store first node because it's parent is -1 only

for(int i=1; i<n;i++){
```





Lecture 13 : Kruskal's Algorithm

>Kruskal's algorithm is another popular algorithm used to find the Minimum Spanning Tree (MST) of a graph. Unlike Prim's algorithm, which builds the MST by growing a single tree, Kruskal's algorithm builds the MST by adding edges one by one in increasing order of their weight, ensuring no cycles are formed.

Steps of Kruskal's Algorithm

Sort All Edges:
Sort all the edges in the graph by their weight in non-decreasing order.

Initialize Subsets:
Create a subset for each vertex. Each subset contains only that vertex at the beginning, representing a disjoint set.

Pick the Smallest Edge:
Iterate over the sorted edges and pick the smallest edge. Check if adding this edge to the MST would form a cycle:

If it doesn't form a cycle, include this edge in the MST.
If it forms a cycle, discard this edge.
Repeat:
Repeat the above step until there are V-1 edges in the MST, where V is the number of vertices in the graph.

Key Components

Union-Find Data Structure:
To efficiently manage and merge subsets, Kruskal's algorithm uses a Union-Find data structure, also known as Disjoint Set Union (DSU). It supports two main operations:

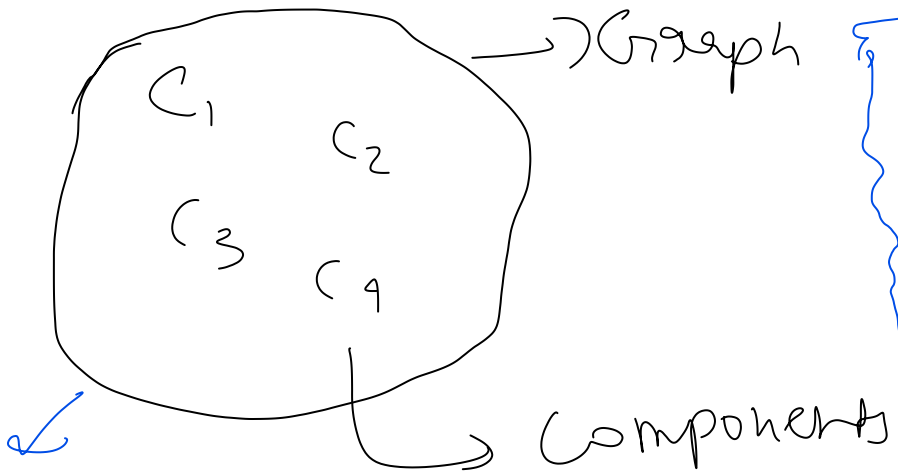
Find: Determine the subset (or set representative) to which a particular element belongs.

Union: Merge two subsets into a single subset.

Cycle Detection:
By using the Union-Find data structure, the algorithm ensures that no cycles are formed when adding an edge to the MST. If two vertices of an edge belong to the same subset, adding this edge will form a cycle.

lets go through the lecture -

① Disjoint Set :



Disconnected.

to know if u & v are connected, we can check it using Disjoint sets.

- 2- important operation →
  - → find Parent() or find set()
  - → Union() or Union Set()

→ lets say we have 5 disconnected components.

→ everynode is a parent of itself.

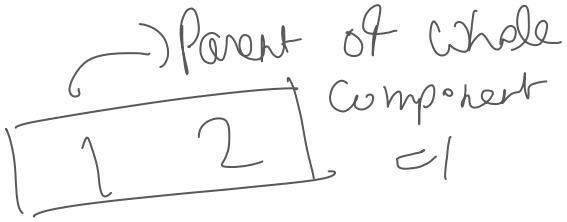
→ find parent(1) → 1

Find parent(2) → 2

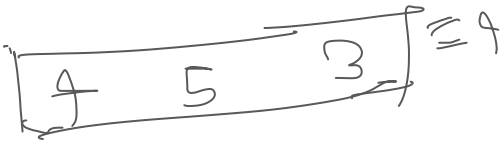
(3) → 3

(4) → 4

(5) → 5

→ Union(1, 2) → 

Union(4,5) → 

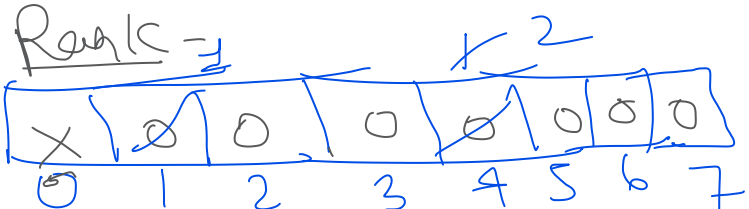
Union (3,5) → 

Union (1,3) →

→ Union by Rank & Path Compression

. Let's say we have 7 components.

→ Union(1,2)



→ lets find Rank for components.

		Rank
1	1 → 1	0
2	2 → 2	0

→ if Rank is same for both, we can connect nodes in any order.

so 1 ← 2 or parent[2] = 1

rank[i] ++;

if ranks equal for both made rank[i]++

now Union(2, 3)

2 → 1 (1 → rank(3) < rank(1))

3 → 3

so parent(3) = 1

1 ← 3

rank[1] = 1

again • Union(4, 5)

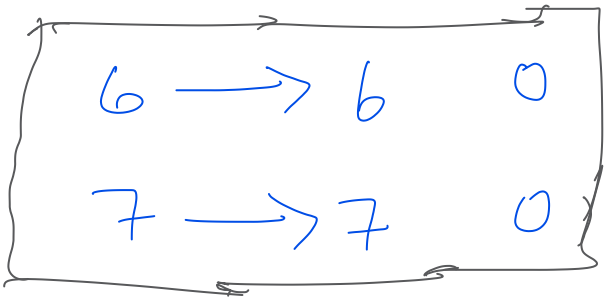
4	4 → 4	0
5	5 → 5	0

so 4 ← 5

rank[4] ++

so rank(4) = 1

now · Union (6, 7)

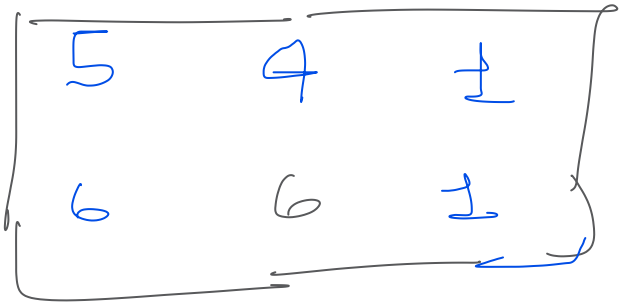


so (6) ← (7) (rank[6]++)

Parent[7] = 6

rank[6] = 1

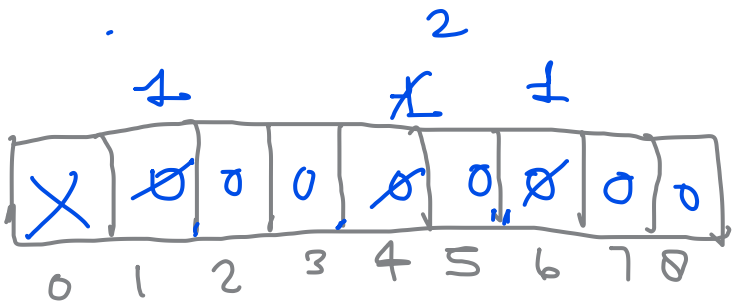
now Union (5, 6)

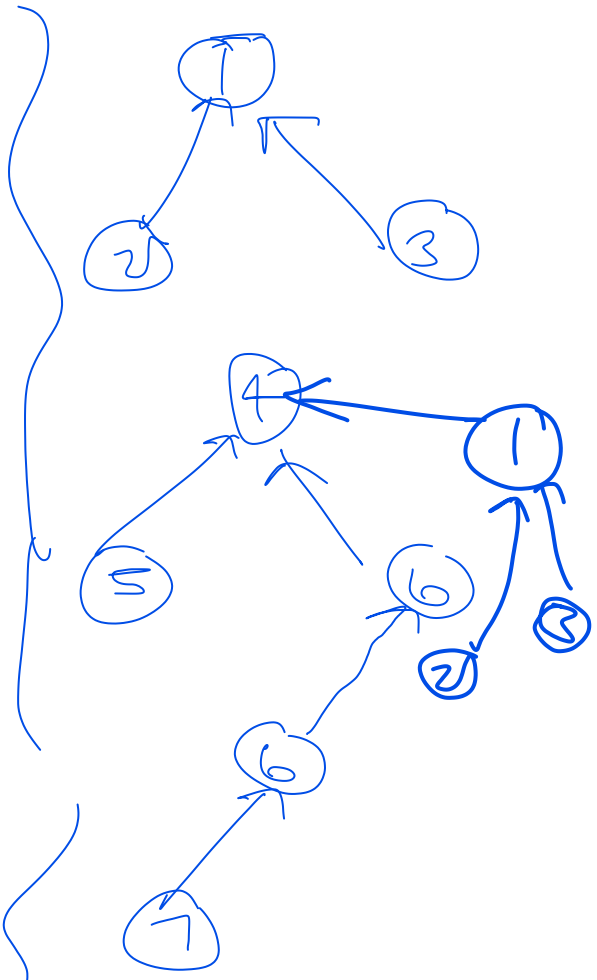


so (4) ← (6)

Parent[6] = 4 , rank[4]++

so rank[4] = 2





- lets take (3, 7)
  Rank

  $3 \longrightarrow 1 \quad 1$

  $7 \longrightarrow 4 \quad 2$

rank(4) > rank(1)

So

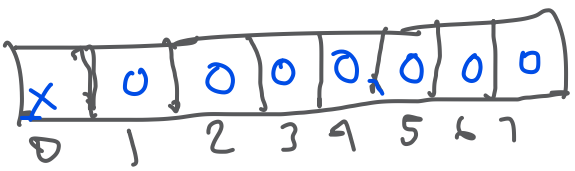
4 ← 1.

Parent[1] = 4

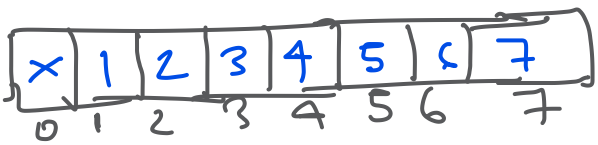
So process will be like-

① make set.

↳



rank



parent.

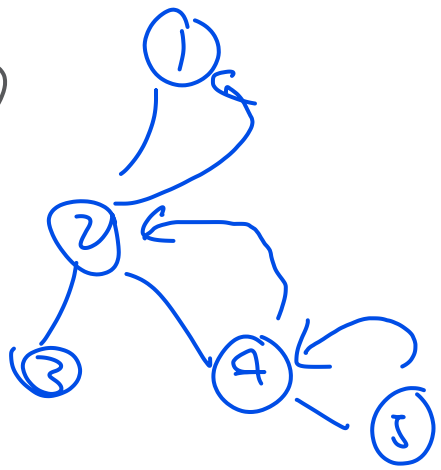
```
So vector<int> parent(n);
   vector<int> rank(n);
   for (int i=0; i<n; i++)
      {
         Parent[i] = i;
         rank[i] = 0;
      }
```

② `int findParent(parent, node) {`

```
if(node == parent[node])
    {
      return node;
    }
return findParent(parent, node);
}
```

Parent[node] =

→



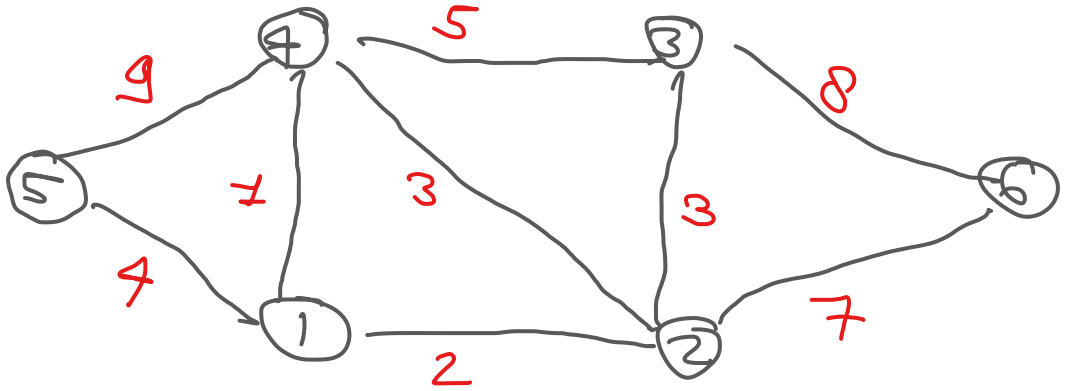
Find parent of 5

Return 1

③ Prepare Union Set - (4, V, & Parent)

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

→ lets discuss kruskal's Algo:



(1) No need of adj list

② Linear DS: we need to store (w, u, v) in an DS then, sort according to weight.

1 – 2 2 ✓
1 – 4 1 –
1 – 5 4 ✓ So
4 – 5 9
4 – 3 5 ✓
2 – 4 3 –
2 – 3 3 ✓
2 – 6 7 ✓
3 – 6 8 ✓

Wt	u	v
1	1	4
2	1	2
3	2	4
3	2	3
4	1	5
5	4	3
7	2	6
8	3	6
9	4	5

Sorted according to the weight.

⇒ Check if(Parent[u] != parent[v])
merge(u, v);

else

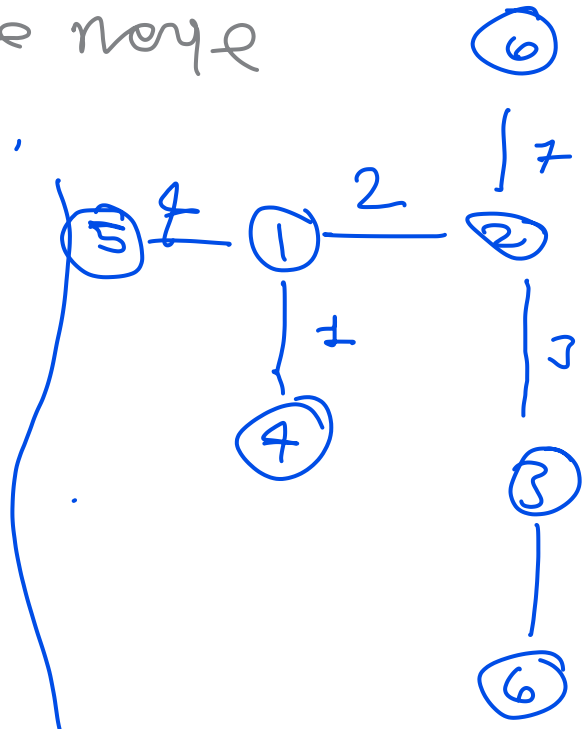
ignore ;

(Initially all Parents are 1's)

So Check for 1 & 4

① 1 → 1
4 → 4 > diff so merge

② 1 → 1
2 → 2 > diff merge



③ 2 → 1
3 → 3 > diff

④ 2 → 1
4 → 1 > same (ignore)

⑤ (1, 5)

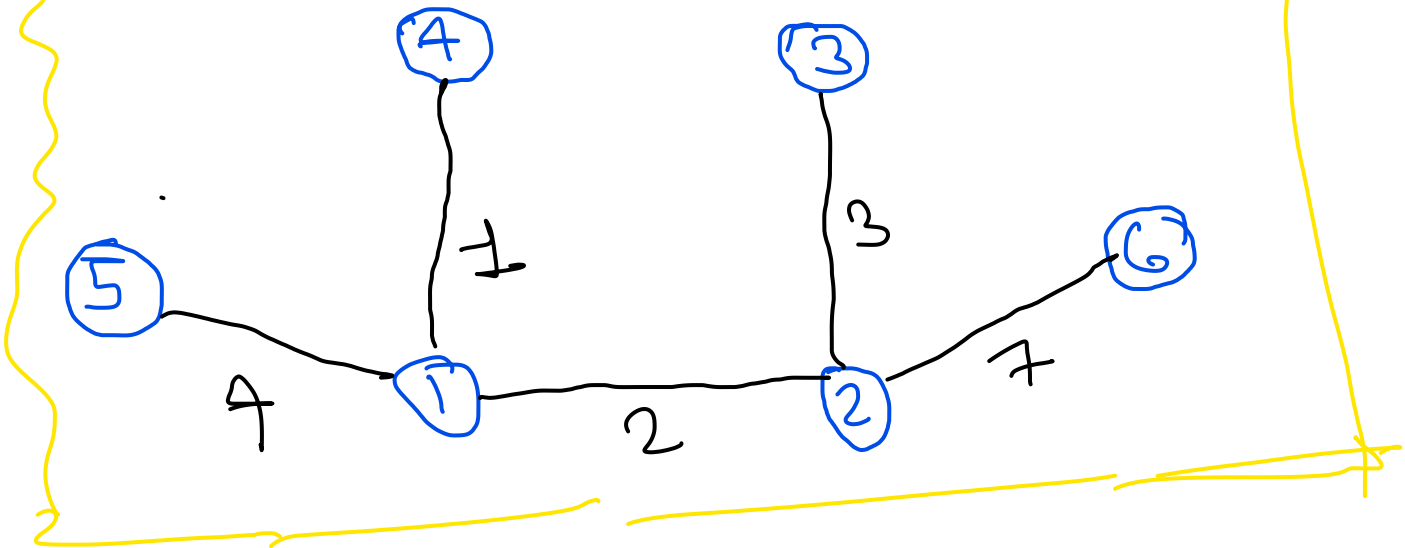
1 is Here but not 5, so diff Components

⑥ (3, 4) → both are in tree so belongs to same, so ignore

⑦ (2, 6) → diff

⑧ (3, 6) → diff

⑨ (4, 5) → same ignore



MST

```
⇒ int mst (vector<vector<int>>& edges, int n)
    {
        sort(edges.begin(), edges.end(), cmp);
        vector<int> parent(n);
        vector<int> rank(n);
        makeSet(parent, rank, n);
    }

→   bool cmp (vector<int> &a, vector<int> &b)
        return (a[2] < b[2]);
    }
    int minweight = 0;
→   for(int i=0; i<edges.size(); i++)
    {
        int u = findParent(parent, edges[i][0]
        int v = findParent(parent, edges[i][1]
        int wt = edges[i][2]
        if (u != v) { minweight += wt;
            unionSet(u, v, parent);
        }
    }
    return weight;
```

}.