Disjoint Set union :

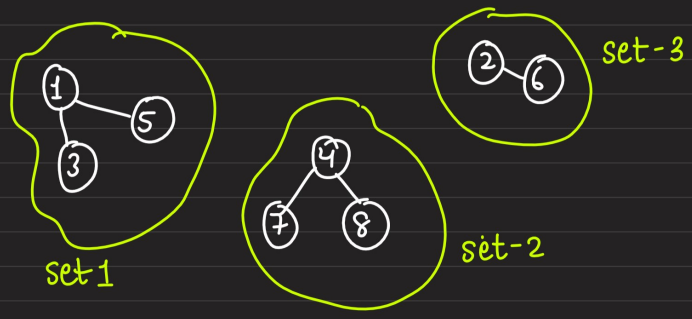
1, 2, 3 are disjoint sets.

We want to support

1) Union
2) Check same { (x,y) in same set or not }
3) Find (x) ↳ set in which x is present.

We want to do these operations in sub-linear time.

Analogy with graph :



Find(x) → find comp. no. of x

check-same(x,y) → if (x,y) is in same component or not

not easy { Above 2 queries can be handled in O(1) complexity. But taking union ≡ adding edge b/w connected components

Union → O(N)

Check-same / Find → O(1)

} using graphs.

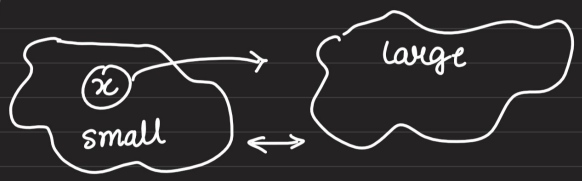
Idea 2 : Maintain sets / DS

Small-to-large trick.

Always merge smaller set into larger set. Deletion & add in set log(N).

Now, time complexity over all the queries is $O(\log^2 N)$

logic :

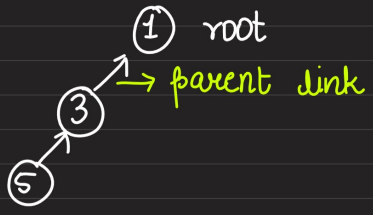


size of set in which x lies after joining doubles.

So, total no. of moves that can be there is log N and each costs log N. So, for every query, time taken = $O(\log^2 N)$

Q. Can we do better than this ? → YES

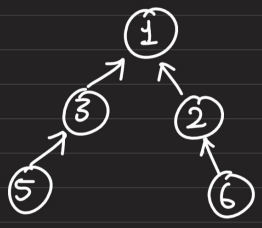
Idea 3 : Union Find :



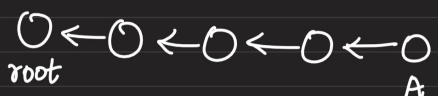
5 belongs to set same as 1.

3 belongs to set same as 1.

Union (5, 6) ≡



Is union Randomly Good?

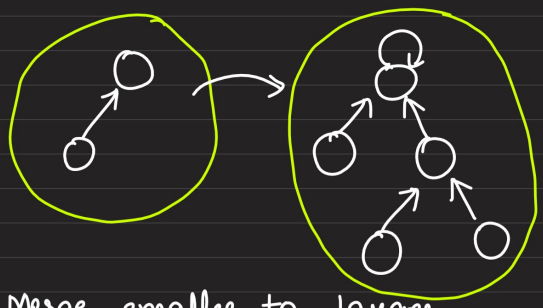


When we call find for elem. A then it will follow parent links to reach the root.

O(N) complexity

Union became easy, but find complexity became worse.

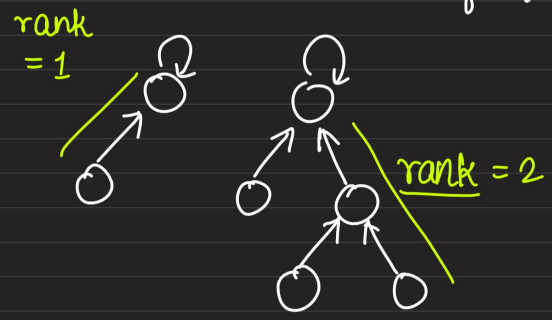
Improving



Merge smaller to larger.

Improvement - 1 : [Rank - Compression]

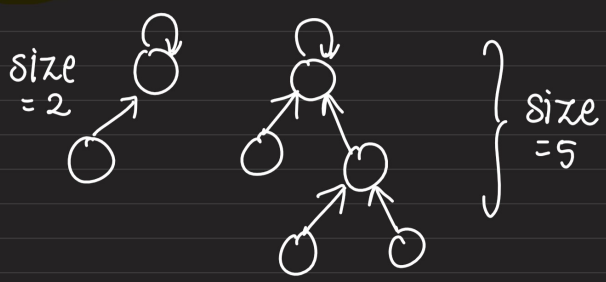
↳ longest among all the chains from root to node.



Now, time complexity : $O(\log N)$

even after all the merges the rank can be almost $\log(N)$.

Improvement 1 : (Size - Compression)



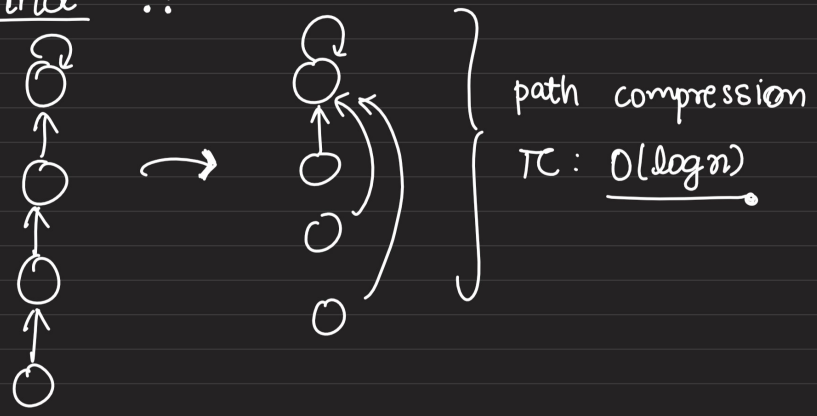
This will be also $\log(N)$.

Using size is better as it also tells the size of set!

Union - Done ✓

Improvement 2: [Path Compression]

Find - ??



Improvement 3: [Doing both Size & Path compr.]

↳ Time complexity further reduces to $O(\log^* N)$

$$\log^* N = \begin{cases} 0, & N = 1 \\ 1 + \log^*(\log N), & N \neq 0 \end{cases}$$

↳ very small

eg: $\log^* 2^{6400} \approx 5-6$ } O(1)

Implementation using set :

```
struct option_1{
    int elementIn[100100];
    vector<int> customSet[100100];
    void init(){
        for(int i=1;i<=100000;i++)
        {
            elementIn[i] = i;
            customSet[i].push_back(i);
        }
    }
    int find(int x){
        return elementIn[x];
    }
    int unite(int x,int y){ //merge sety into setx
        lli setx  = elementIn[x];
        lli sety = elementIn[y];
        if(customSet[setx].size()<customSet[sety].size()){
            swap(setx,sety);
        }
        for(auto it: customSet[sety]){
            elementIn[it] = setx;
            customSet[setx].push_back(it);
        }
        customSet[sety].clear();
    }
};
```

Assumuption :
Atmax there will be 10^5 no. of sets.

and value of element in the set is between $0 - 10^5$.

Initially,

Integer 'i' is in the i^{th} set.

So, elementIn[i] = i ;

i^{th} Integer in i^{th} set.

and customSet[i] contains only integer i

Now, finding in which set element belongs to is O(1).

Unite

Suppose given sets are
set 1 → <1, 3, 5>
set 2 → <2, 11>
set 3 → <4, 7, 9>

We will first create set 1/2/3 using unite.

unite (1,3) → unite set 3 to set 1 & then clear set 3.
So,
customset[1].push_back(3)
elementIn[3] = 1;

unite (1, 5) → customset[1].pushback(5)
elementIn[5] = 1;
} O(1)

Similarly we create set 2 and set 3.

Implementation using Union find :

```
//Option 2 -> union find
struct union_find{
    int par[100100];
    int rank[100100];
    void init(){
        for(int i=1;i<=100000;i++)
        {
            par[i] = i;
            rank[i]= 0;
        }
    }
    int find(int x)
    {
        if(par[x]==x) return x;
        else return (par[x] = find(par[x])); //path compression
    }

    //rank compression
    int unite(int x,int y){ //merge sety into setx
        int rootx = find(x);
        int rooty = find(y);
        if(rank[rootx]<rank[rooty]) swap(x,y);
        if(rank[rootx]==rank[rooty]) rank[rootx]++;

        par[rooty] = rootx;
    }
};
```

Here, we keep parent and rank.

Initially parent[i] = i;
rank[i] = 0;

← updating the parents as well

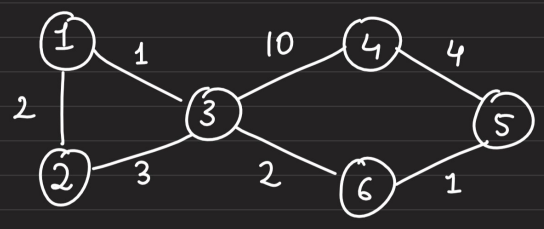
→ merging smaller set to larger.

Minimum Spanning Tree :-
(of a graph)

N nodes and M edges } weighted / connected

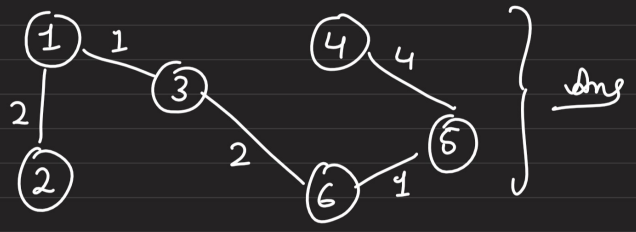
↳ create tree of N nodes by taking (N-1) out of M edges such that weight of tree is minimum.

eg: G(V,E)



Find MST.

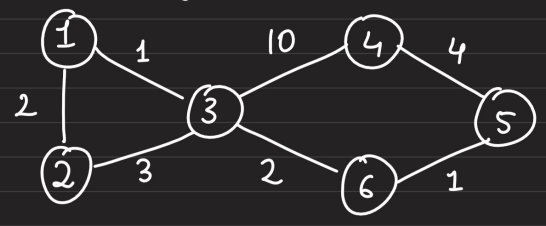
Soln:



Ans

MST
- Kruskal's Algorithm } enough to solve all the problems.
- Prim's algorithm

Kruskal's Algorithm

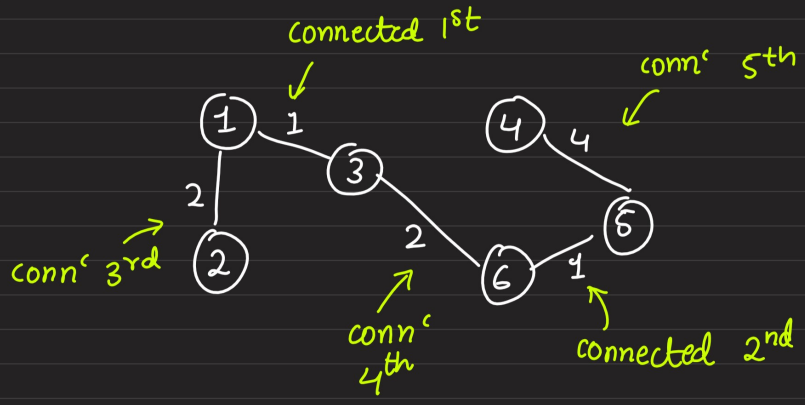


Greedy: Take edges of smaller cost.

sorted edge-list

1, 3 → 1
5, 6 → 1
1, 2 → 2
3, 6 → 2
2, 3 → 3
4, 5 → 4
3, 4 → 10

{ Connect edges of small cost but skip if you get a cycle.



NOTE: Edge 2,3 → 3 got skipped.

How to skip ??

↳ We can use union-find here.

Find(x), Merge(x, y)

↳ If (find(a) == find(b)) → don't connect them.

Sort = $O(E \log E)$

Iteration on all edges and merging $V-1$ edges
$O(E+V)$

Total = $O(E \log E)$ }

Implementation :

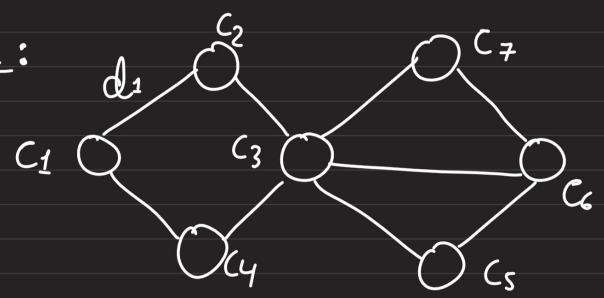
```
lli n,m;
cin>>n>>m;
vector<pair<int,ii>> edgeList; //weight, {u,v}
union_find uf;
uf.init();

int u,v,w;
for(int i=0;i<m;i++)
{
    cin>>u>>v>>w;
    edgeList.push_back({w,{u,v}});
}
sort(edgeList.begin(),edgeList.end());

lli mst_cost = 0;
lli cnt = 0;
for(auto edge :edgeList)
{
    int u = edge.S.F;
    int v= edge.S.S;
    int wt = edge.F;
    if(uf.find(u)!=uf.find(v))
    {
        cnt++;
        mst_cost += wt;
        uf.unite(u,v);
    }
}
if(cnt==n-1)
{
    cout<<mst_cost<<'\n';
}else{
    cout<<"IMPOSSIBLE";
}
```

Problem 1 : Cost of Maximum spanning tree.

App 1.
- reverse_sort edge_list and follow the same greedy approach.
- start with largest weighted edge and go to small.

App. 2
- Modify graph → change weight to -wt.
- the find cost of mst & ans = -cost;

Problem 2 :



○ → nodes are factories.

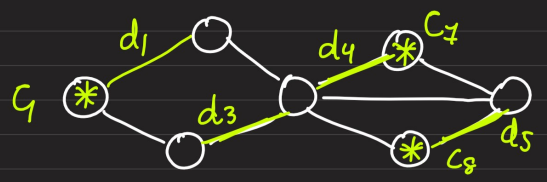
— → edges are roads.

Initially, none of the roads / factories are working.

Cost of activating i^{th} factory & i^{th} road is C_i & d_i resp.

If a factory is opened, it can start producing goods and if roads are activated, goods can be transferred b/w nodes connecting them.

eg:



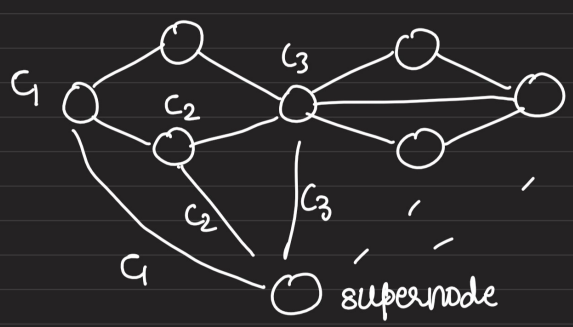
} green - coloured nodes / edges are activated!

Here cost = $d_1 + d_3 + d_4 + d_5 + C_1 + C_7 + C_8$

Find minimum cost needed to make goods reach all of the plants.

$N, M \leq 10^5$ } constraint.
$C_i, d_i \leq 10^9$ }

sol^n: It is like multisource MST.



} connect all the nodes with super-node using N edges of weight C_i

Now, our answer is MST of the modified graph.

Weighted Union find :- (Adv. Idea 1)

(Level - 2)

The problem

N variables x_1 x_2 x_3 ..... x_N

KB → Knowlegde base

Q → queries

↳ 1 a b c → Add this to knowledge base ($x_a - x_b = c$) or Inconsisten

↳ 2 a b → Print $x_a - x_b$

N = 5 , Q = 7

1	2	3	5	$x_2 - x_3 = 5$	} $x_2 - x_5 = 7$
1	3	5	2	$x_3 - x_5 = 2$	
1	2	4	3	$x_2 - x_4 = 3$	
1	2	5	8	$x_2 - x_5 = 8$ → not added as it is inconsisten. Print (IC)	

1 2 5 7 $x_2 - x_5 = 7$ → do nothing.

2 5 4 $x_5 - x_4$ → print (-4)

We know $x_2 - x_5 = 7$, $x_2 - x_4 = 3$ } $x_5 - x_4 = -4$

2 1 5 → $x_1 - x_5$ → can't be determined.

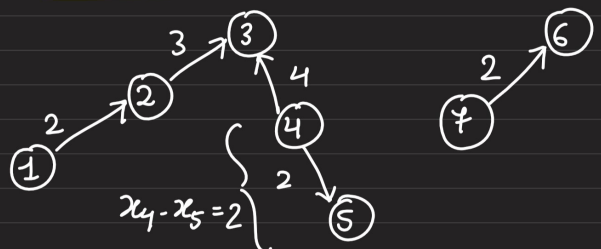
What does information tells us ?

$x_a - x_b = c$ add ??

$\Rightarrow x_a = x_b + c$

} Representation

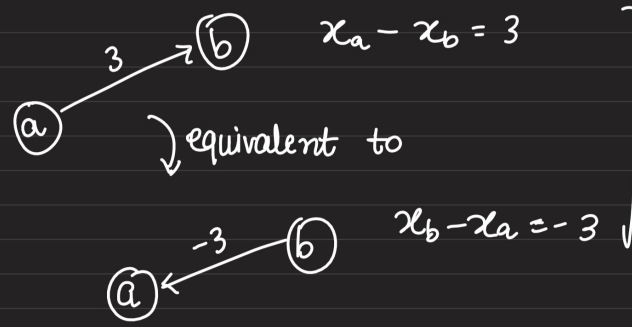
How does this help?



no unique parent now, so a bit different than DSU.

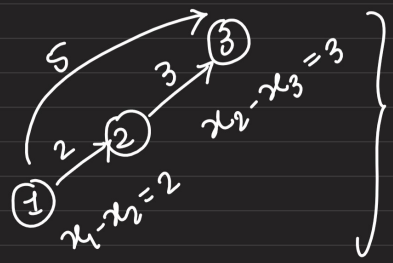
Observation :

1.



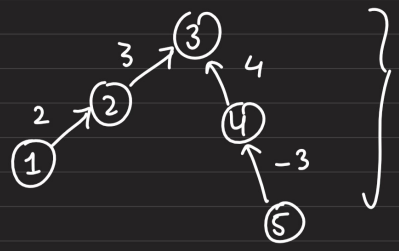
} this can help us to maintain single parent.

2.



} kind of weighted path reduction.

3.



We can find $x_i - x_j$ b/w any $i, j \in$ set

$x_i - x_{root} =$ known1

$x_j - x_{root} =$ known2

$\Rightarrow x_i - x_j =$ known 1 − known 2

Designing DSU:

a) designing find

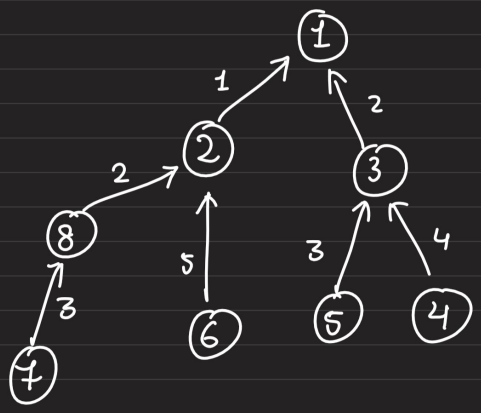
Find $(x) \rightarrow \langle p, c \rangle$

p: parent, c: cost

Find (6) = ⟨1,6⟩

Find (4) = ⟨1,6⟩

Find (1) = ⟨1,0⟩



If we want $x_a - x_b$

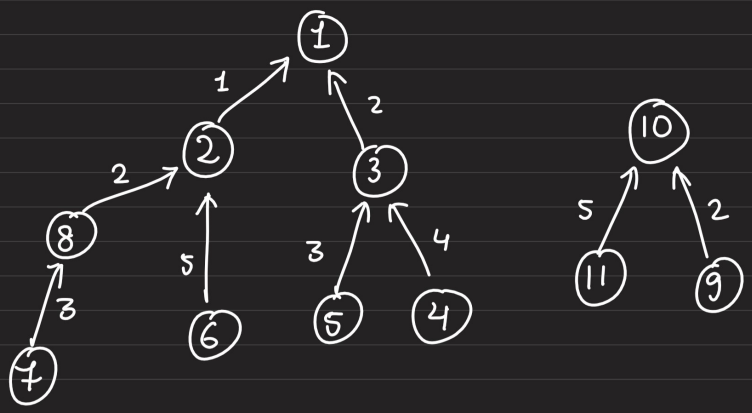
find (a) = (parA, c_a)

find (b) = (parB, c_b)

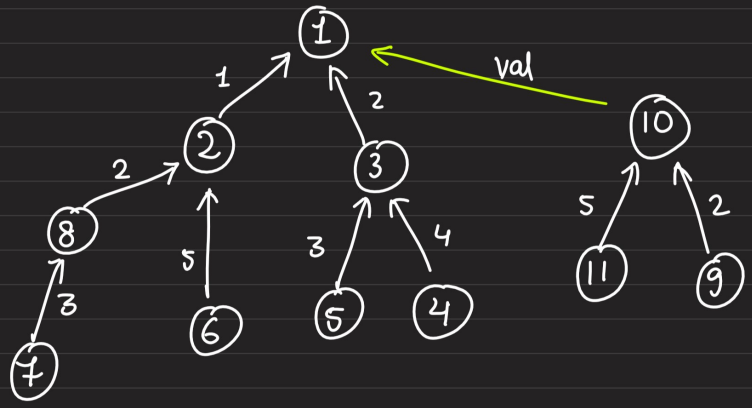
if (parA ≠ parB) can't be determined

if (parA = parB) return $(c_a - c_b)$;

b) design union



union (7, 9, 5)



$x_7 - x_1 = 6$

$x_9 - x_{10} = 2$

$x_{10} - x_1 = val$

$x_7 - x_9 = 5$

} Solve them

Alites

$x_7 - x_1 = 6$

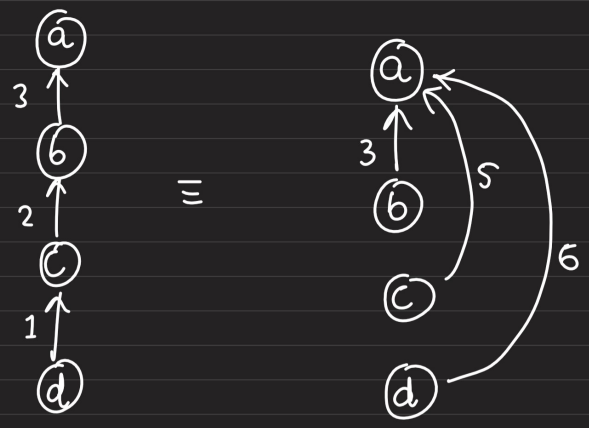
$x_9 - x_1 = 2 + val$

$x_7 - x_9 = 5 = 4 - val$

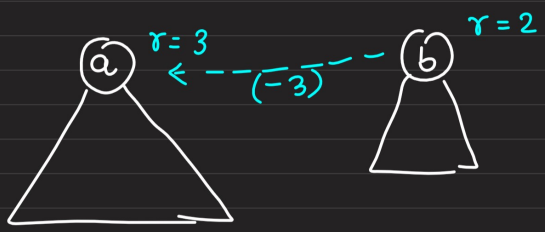
$\therefore$ val = -1.

{ Instead of merging 2 nodes, we merge their leaders with some value.

Path Compression



Rank Compression



$x_a - x_b = 3$

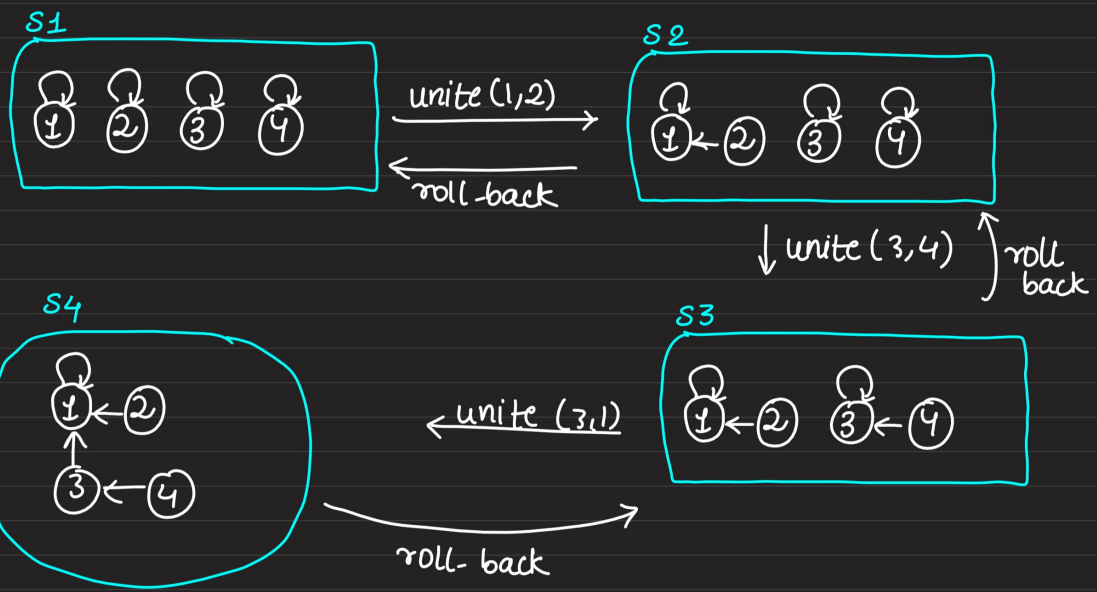
{ Instead of this, we will do $x_b - x_a = -3$

Implementation :

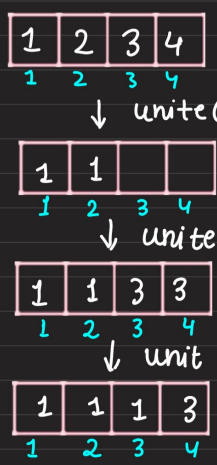
```
struct weighted_union_find
{
    vector<lli> par;
    vector<lli> rank;
    vector<lli> val;
    void init(lli n)
    {
        par.resize(n+1);
        rank.resize(n+1);
        val.resize(n+1);
        for(lli i=0;i<=n;i++)
        {
            par[i] = i;
            rank[i] = 0;
            val[i] = 0;
        }
    }
    ii find(lli x)
    {
        if(x==par[x]) return {x,0};
        else
        {
            ii temp = find(par[x]);
            par[x] = temp.F;
            val[x] += temp.S;
            return {par[x],val[x]};
        }
    }
    void unite(lli a,lli b,lli w) //merge rootb to roota
    {
        lli roota = find(a).F;
        lli vala = find(a).S;
        lli rootb = find(b).F;
        lli valb =  find(b).S;
        lli x = w + vala - valb;

        if(rank[roota]==rank[rootb]) rank[roota]++;

        val[rootb] = x;
        par[rootb] = roota;
    }
};
```

Rollback Idea on Union-Find: (Adv Idea 2)

UF
- Find (x)
- unite (x, y)
- roll-back ()



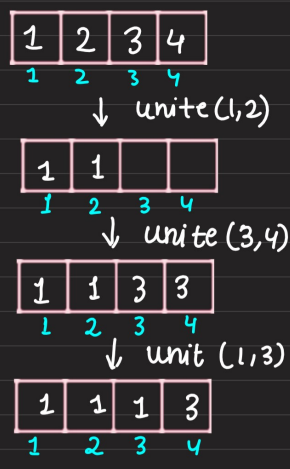
Conversion of parent array

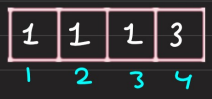


Naive approach:

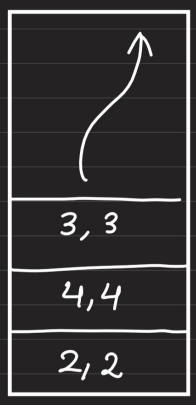
Saving all parent array $O(N \cdot Q)$

How to model ?





} final state



Changes stack

4's intial value was 4

2's initial value was 2

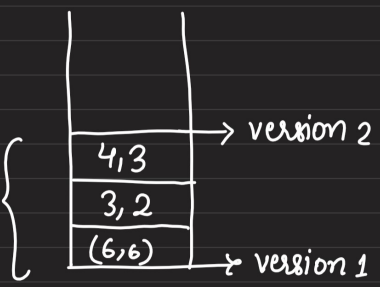
Take the top element revert parent and pop();

Issue with path compression :

unite(4,5)

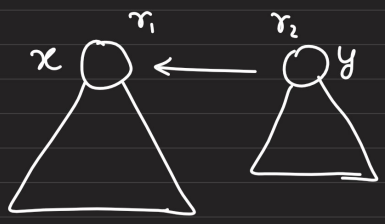
find(4) → do path compression

{ can be O(N)



So, don't use path compression.

Only use Rank compression.

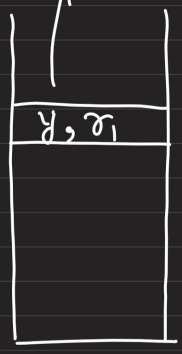


// assuming $r_1 \geq r_2$

```
if(r1 == r2) r1++;
par[y] = x
```

(∵ it must have been leader)

par[y] = y



Initially:

rank	r_1	r_2
par	x	y

→

	r_2
x	x

x y

par[y] = x

To rollback:

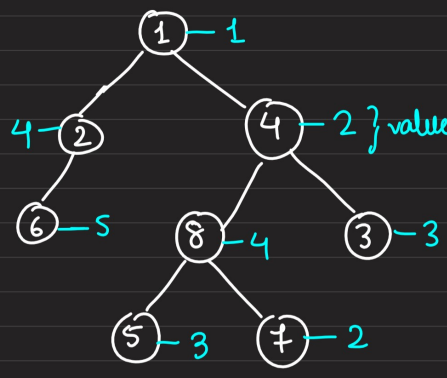
```
int x = par[y];
rank(x) = r1;
par[y] = y;
```

previous state

O(1) rollback

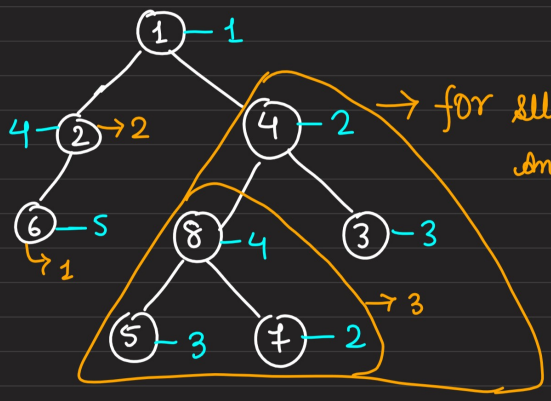
DSU on tree : (Adv Idea 3)

Problem) For each node, find no. of distinct values in the subtree



$N \leq 10^5$

Sol:

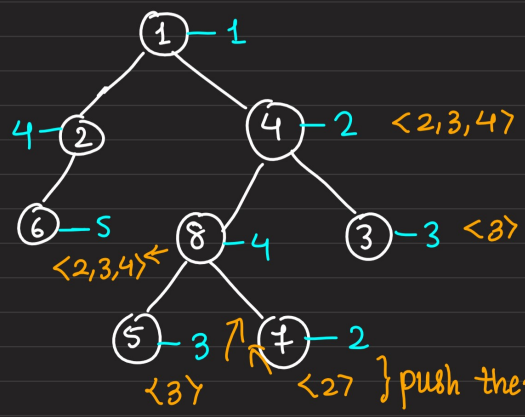


→ for subtree at node 4
Ans = 3 {2, 3, 4}

— → value of node

— → Ans for subtree at that node.

Idea



We will have N-1 merges. Create sets starting at leaf level and merge them going upwards to the root.

set merging ⇒ DSU

Implementation :

```cpp
vector<lli> val;
vector<vector<lli>> g;
vector<set<int>> cols;
vector<lli> ans;

//Small to large technique of merging
int merge(int a,int b) //merge set b into a
{
    if(cols[a].size()<cols[b].size()) swap(a,b);
    for(auto it: cols[b])
    {
        cols[a].insert(it);
    }
    cols[b].clear();
    return a;
}

int dfs(int nn, int pp)
{
    int cur_set = val[nn];
    for(auto v:g[nn])
    {
        if(v!=pp)
        {
            cur_set = merge(cur_set,dfs(v,nn));
        }
    }
    ans[nn] = cols[cur_set].size();
    return cur_set;
}
```

```cpp
void solve()
{
    lli n;cin>>n;
    g.resize(n+1);
    val.resize(n+1);
    cols.resize(n+1);
    ans.resize(n+1);

    for(int i=1;i<=n;i++)
    {
        lli x;
        cin>>x;
        cols[i].insert(x);
        val[i] = i;
    }
    int a,b;
    for(int i=0;i<n-1;i++)
    {
        cin>>a>>b;
        g[a].push_back(b);
        g[b].push_back(a);
    }
    int root; cin>>root;
    int result = dfs(root,0);
    for(int i=1;i<=n;i++)
    {
        cout<<i<<' '<<ans[i]<<'\n';
    }
}
```