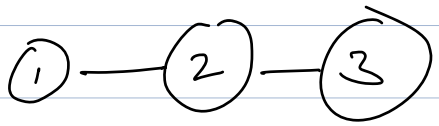
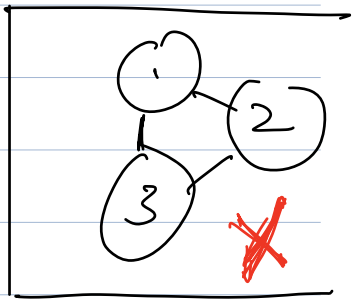
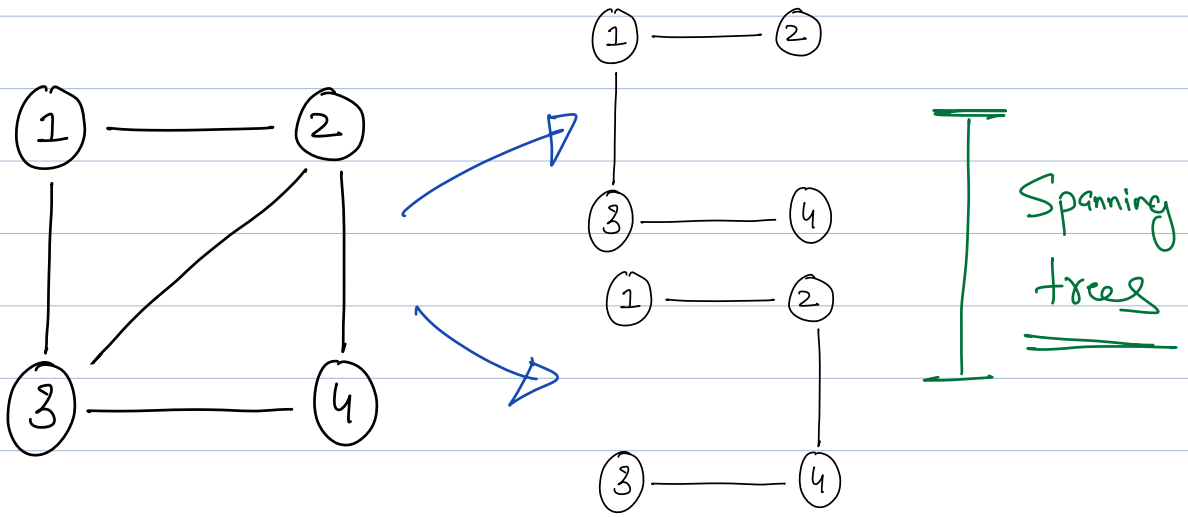
A undirected graph is a tree if it has no cycle and is connected.

eg:





$E = V - 1$

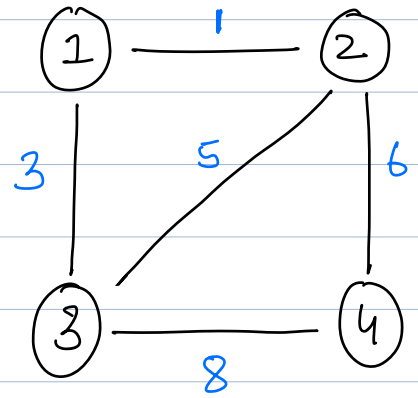


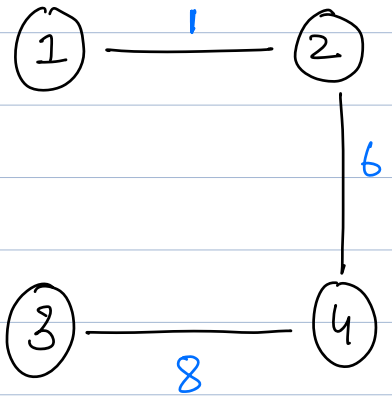
Spanning Tree

A spanning tree T of an undirected graph G is a subgraph that is a tree which includes all the vertices of G.

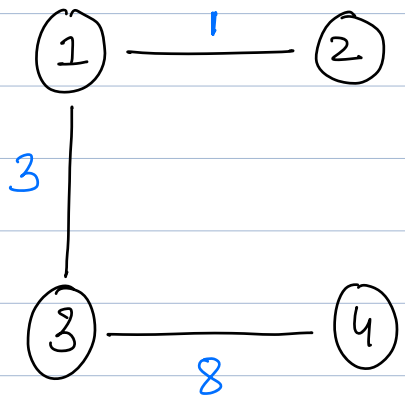
A graph can have several spanning trees.

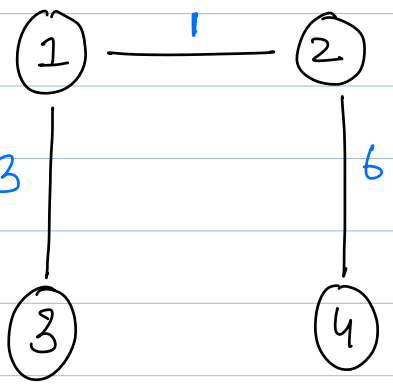
A graph needs to be connected for a spanning tree to exist.





Total Edge Sum ≠ 15

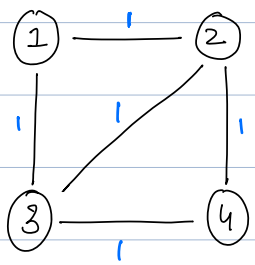


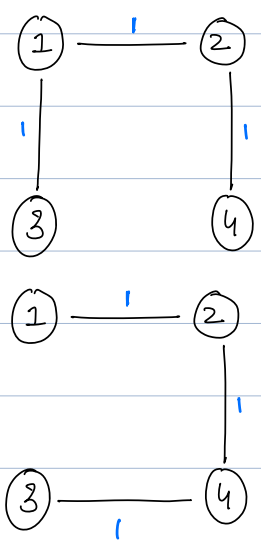


Total Edge Sum ≠ 12

Total Edge Sum ≠ 10

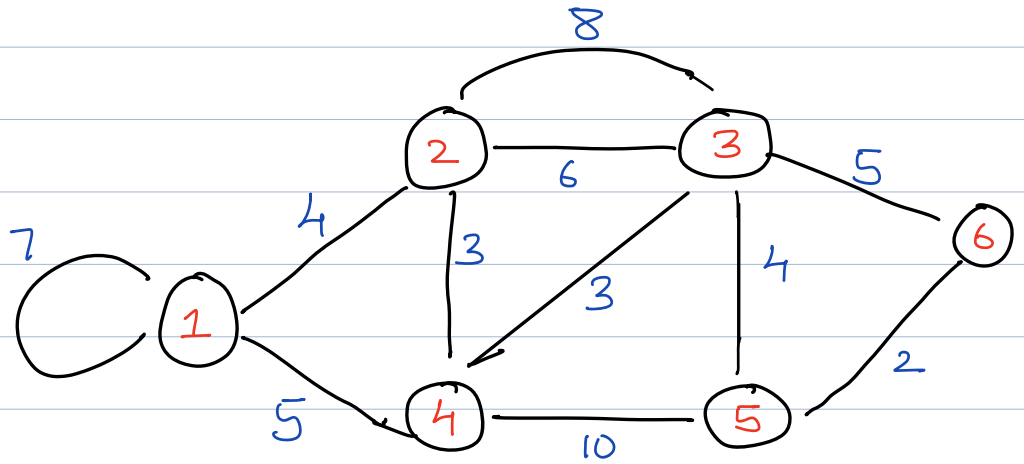
Minimum Spanning Tree (MST)



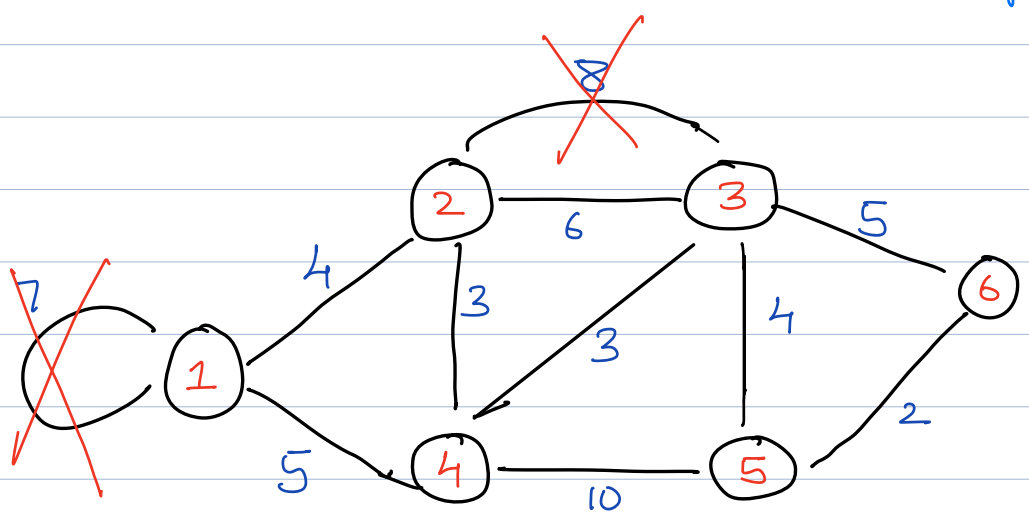


Multiple MSTs are possible

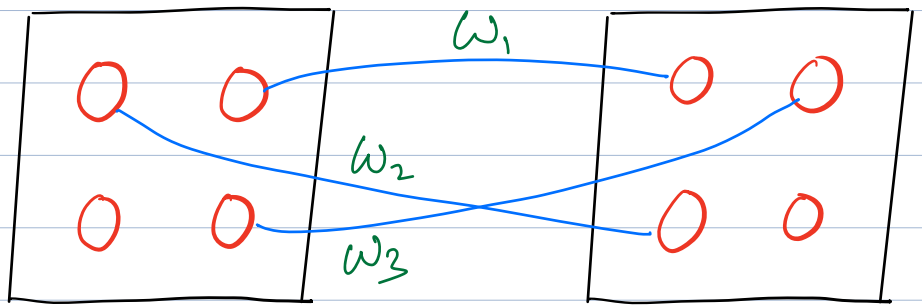
Prim's

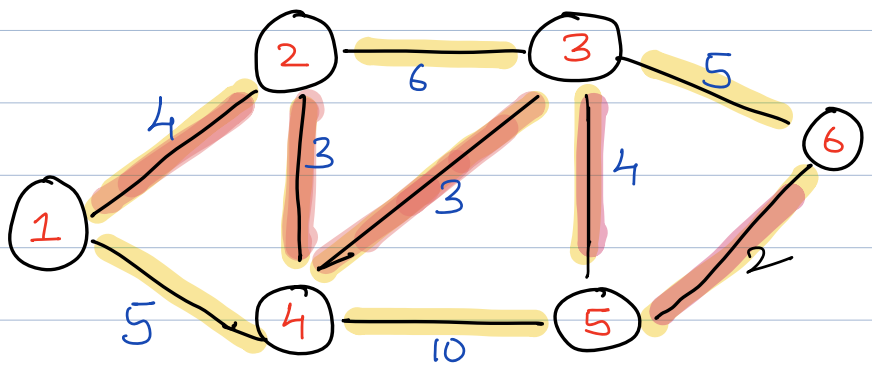


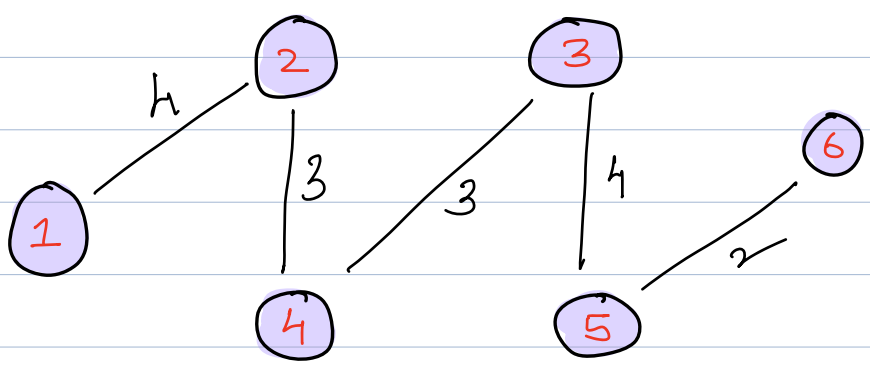
1) Remove Self loops
2) In case of mult-edges keep the one with minimum edge weight



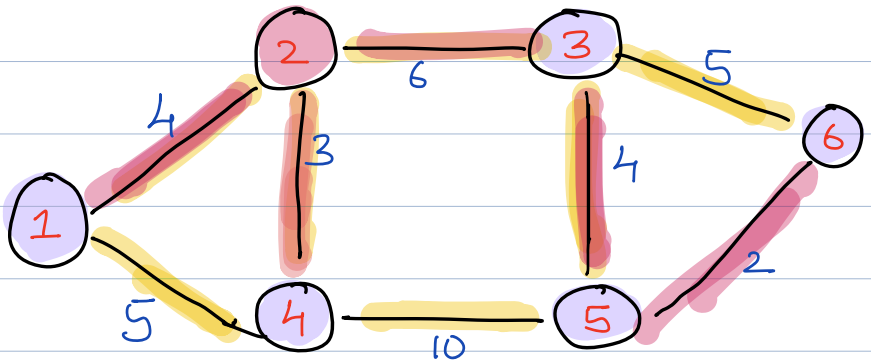
Prim's



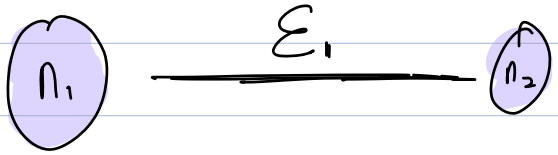


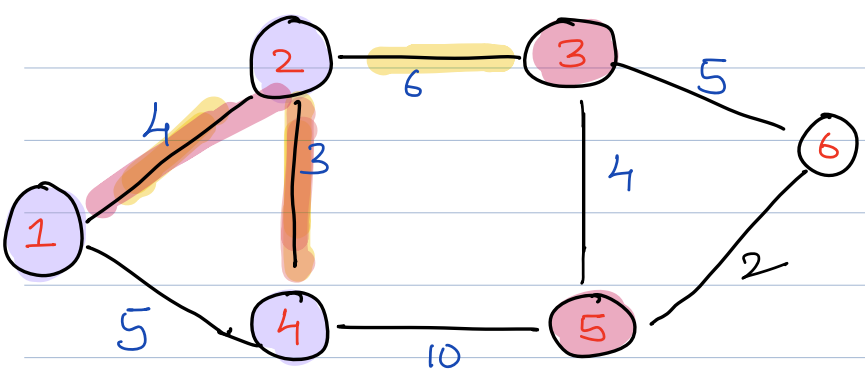


Total Edge Sum ⇒ 16



Edge Sum = 19

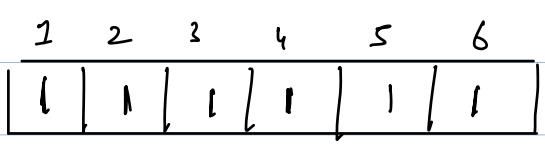


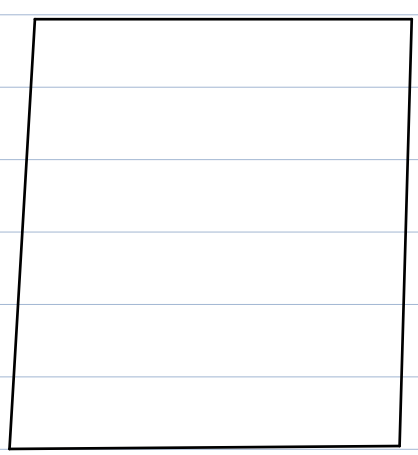


Pair <int, int>

Edge ↵ ↳ Node

Weight





ans = 19

TC: $O(E \log E + V)$

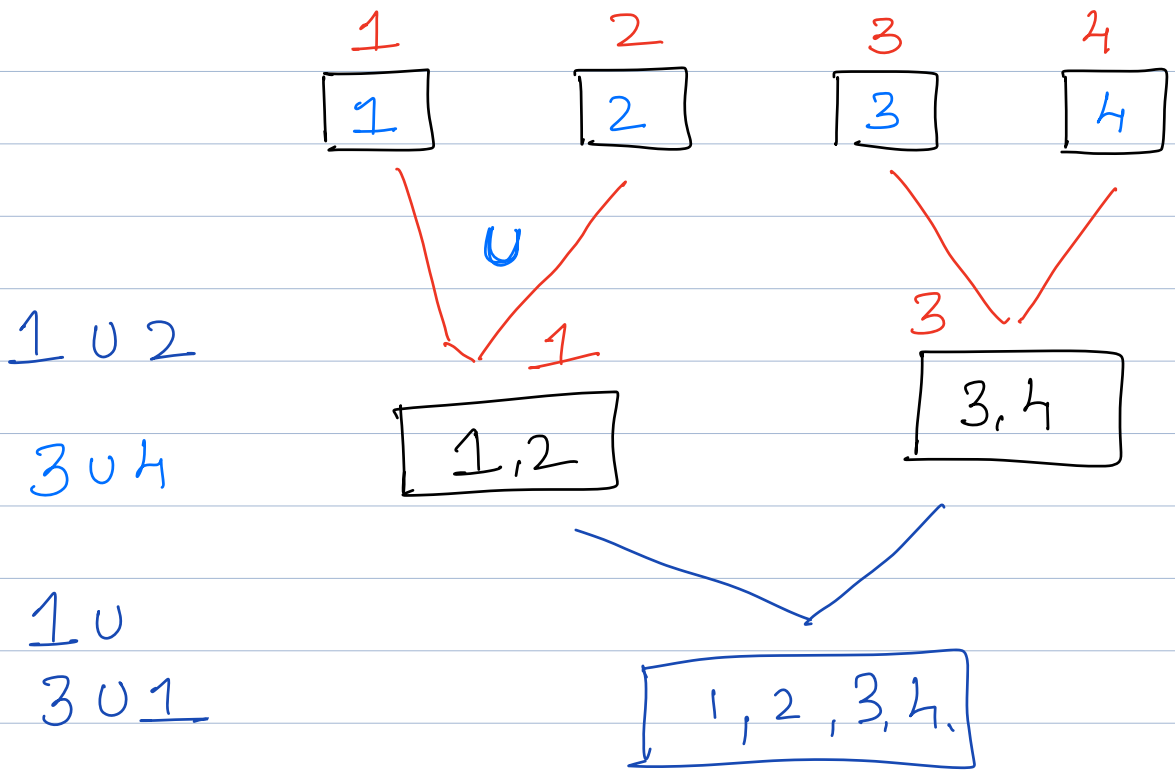
SC: $O(V + E)$

DSU [Disjoint Set Union]

Disjoint Sets ⇒ When sets have nothing in common.

$$S_1 \cap S_2 = \phi$$

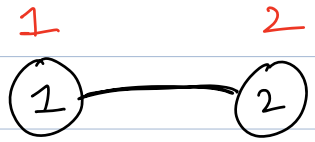
S_1 & S_2 are disjoint Sets.



Parent =

1	2	3	4	5	6
1	2	3	4	5	6

1 ∪ 2



Par [2] = 1
Par [1] = 2

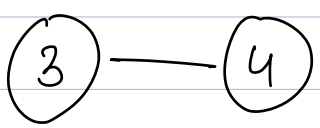
Both Correct

Par [2] = 1

1	2	3	4	5	6
1	1	3	4	5	6

3 ∪ 4.

Par [4] = 3

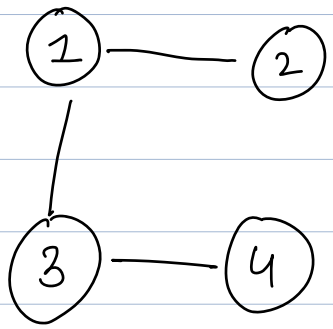


1	2	3	4	5	6
1	1	3	3	5	6

2 U 4

1	2	3	4	5	6
3	1	3	3	5	6

Pat[2] = 4



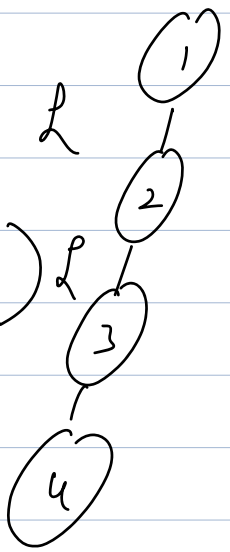
R_1 ⇒ find_Root (2)
R_2 ⇒ find_Root (4)

Pat[R_1] = R_2

OR

Pat[R_2] = R_1

```
int find_root (int node) {
    while ( node != pat[node] ) {
        node = pat[node];
    }
    return node;
}
```



1	2	3	4
1	1	2	3

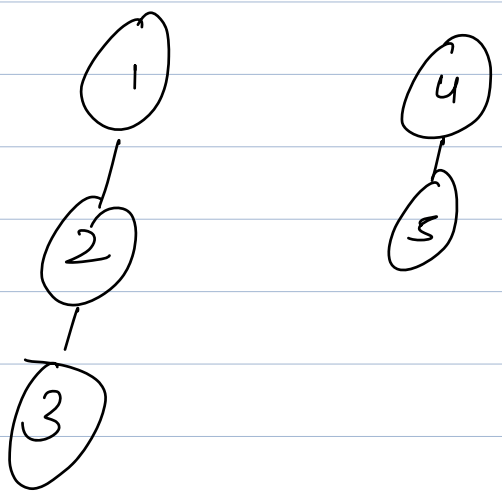
find_root (4)

TC: O(V) SC: O(1)

```
bool union (int x, int y) {
    int r1 = find_root (x);
    int r2 = find_root (y);
    if (r1 == r2)
        return false;
    par [r1] = r2;
    return true;
}
```

TC : O(V)
SC : O(1)

Optimisation 1



par [4] = 1

par [1] = 4

Tree 1 Tree 2

h_1 h_2 $par[R_1] = R_2$

R_1 R_1

$par[R_2] = R_1$

$h_1 \geq h_2$

After union

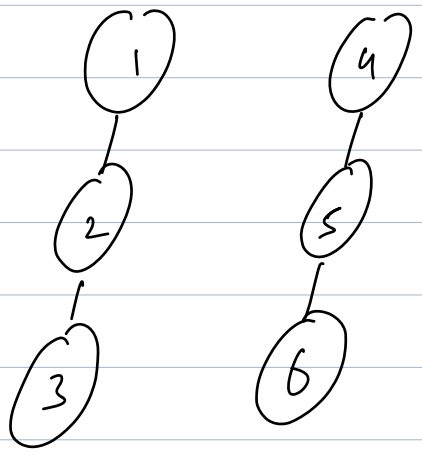
$h \not> h_1$

$h_1 = h_2$

$par[R_1] = R_2$

$par[R_2] = R_1$

$h' \not> h_1 + 1$



Resultant height = $\log(v)$

↓

Find_Root = $O(\log v)$ as we would have

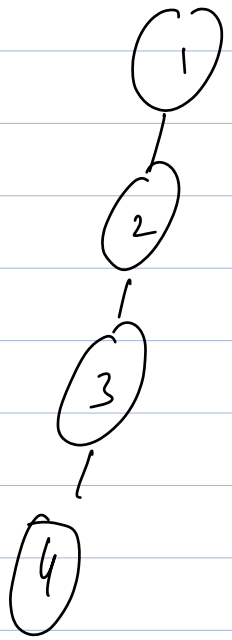
union = $O(\log v)$ balanced tree.

Union by Rank [height]

	1	2	3	4	5	6
h ⇒	1	1	1	1	1	1

```
bool union (int x, int y) {

    int r1 ⇒ find_root(x)
    int r2 ⇒ find_root(y)

    if (r1 == r2)
        return false.

    if ( h[r1] > h[r2] )
        par[r2] = r1;
    elseif ( h[r2] > h[r1] )
        par[r1] = r2;
    else {
        par[r1] = r2
        h[r2]++;
    }

    return true
}
```

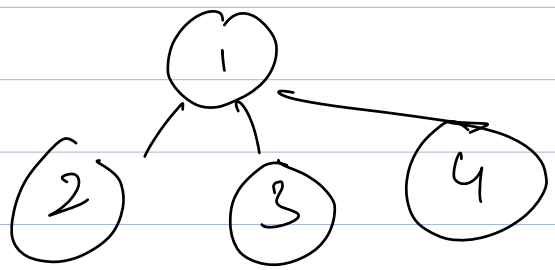
Optimization 2 [Union by path compression]



1	2	3	4
1	1	2	3

find_root(4)

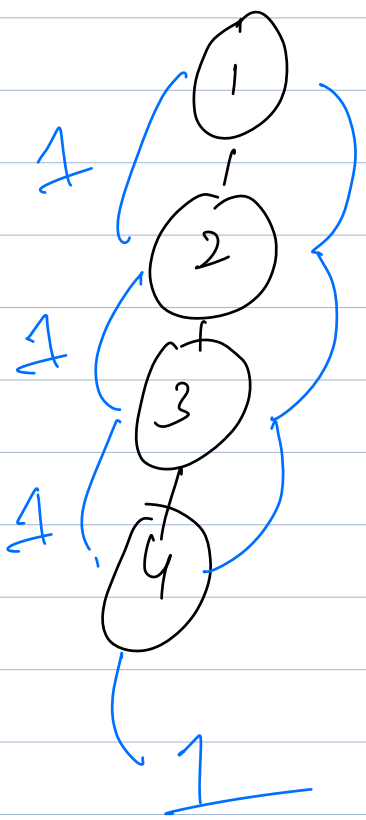
1	2	3	4
1	1	1	1



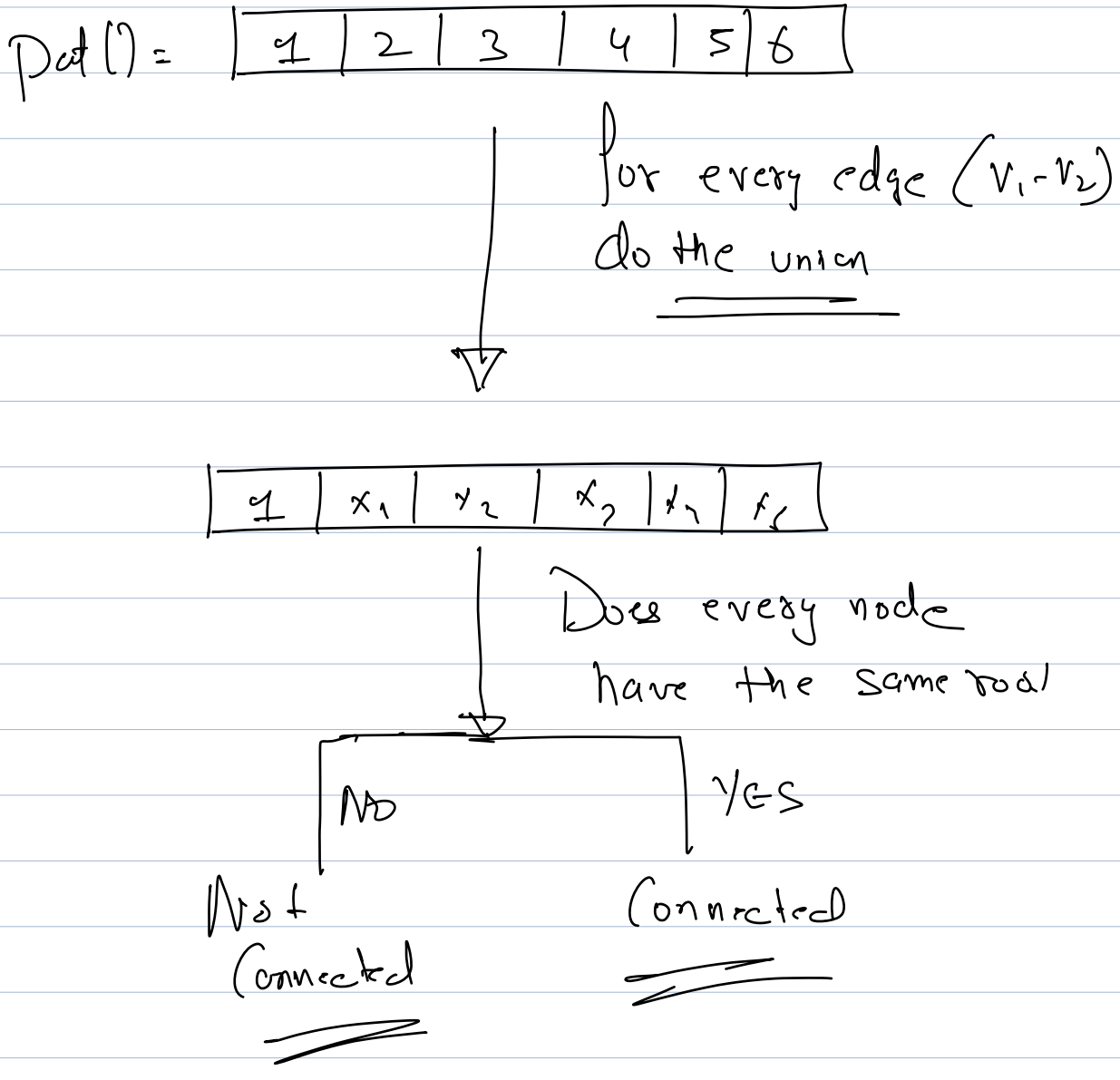
Tc : O(1) amortized

```
int find_root (int node) {
    if (node == par[node])
        return node;
    par[node] = find_root (par[node]);
    return par[node];
}
```

3



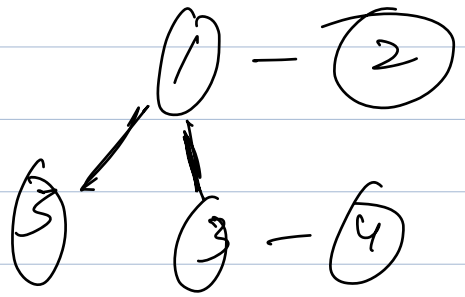
Q1 Given a graph. Check if it is connected.



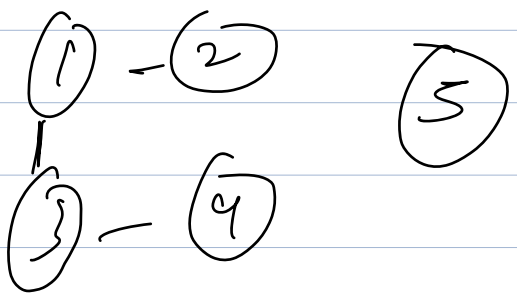
1) Initialise par[] = O(V)
2) Traverse Edges = O(E) x O(1)
3) Check root of every vertex = O(V)

TC: O(V+E)
SC: O(V)

1-2
3-4
1-4
4-5



1-2
3-4
1-4

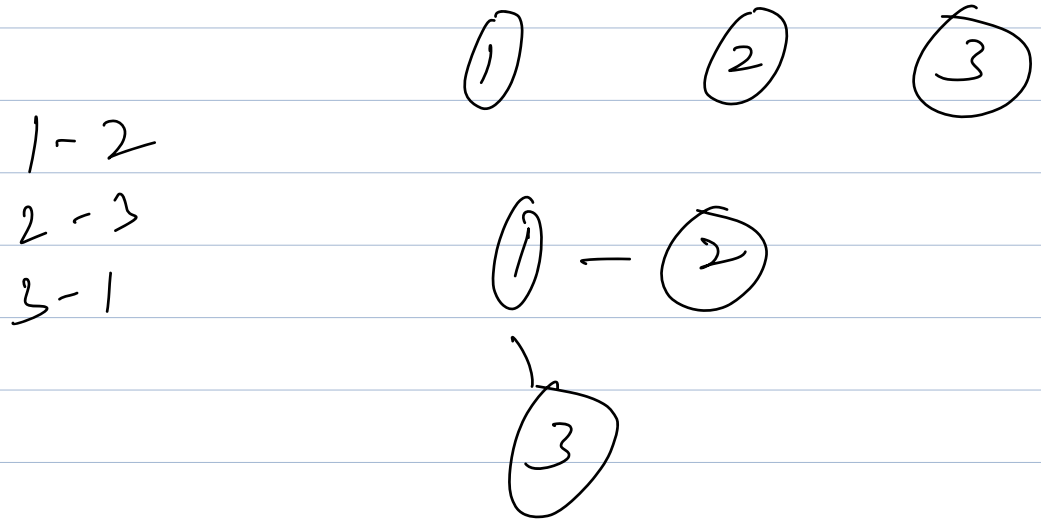


Q2 Given a graph. How many connected components are present?

→ Check the number of different roots after union of all edges.

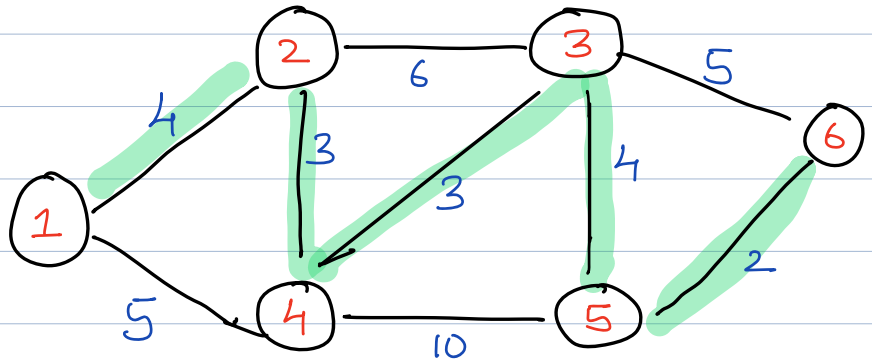
TC: $O(V+E)$ SC: $O(V)$

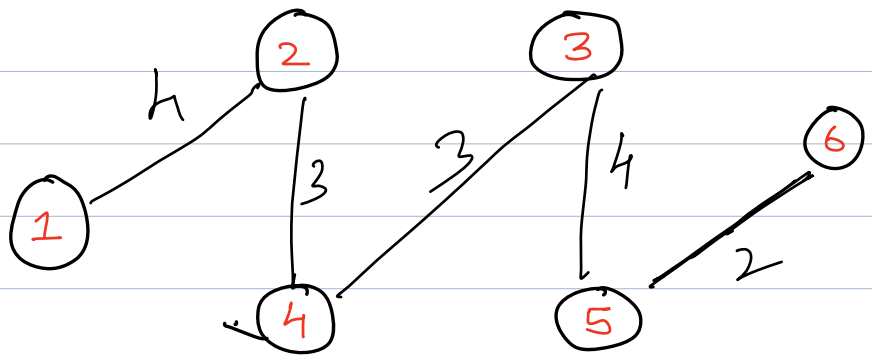
Q3 Given an undirected graph. Check if a cycle exists ?



if (union return false)
cycle exists;

TC: O(V+E) SC: O(V)





⇒ Initialize Parent ⇒ $O(V)$

⇒ Sort Edges. ⇒ $O(E \log E)$

⇒ For all edges do union ⇒ $O(E)$

TC: $O(E \log E + V)$

SC: $O(V)$ ⇒ ignore Sorting

space.

$O(V + \log i)$ → talking

Sorting space into account.