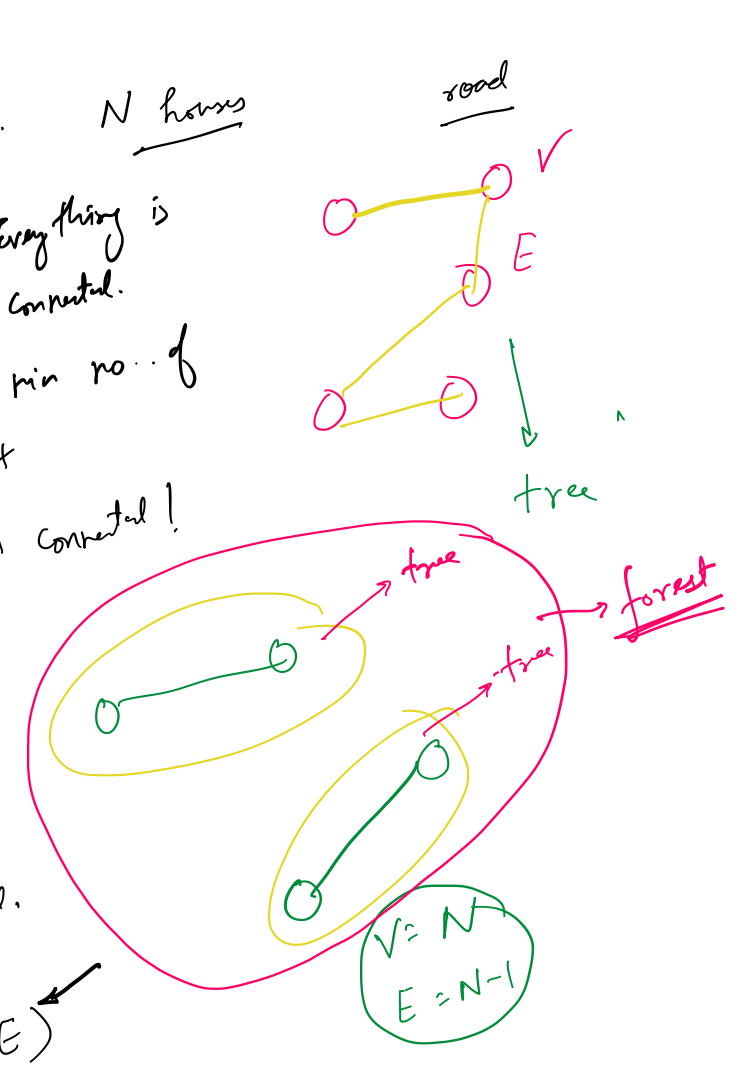
Graphs Follow up Jan

① Deleting e(g). N = Nodes

Initially → Everything is connected.

Remove the min no. of edges s.t. the g is still connected!

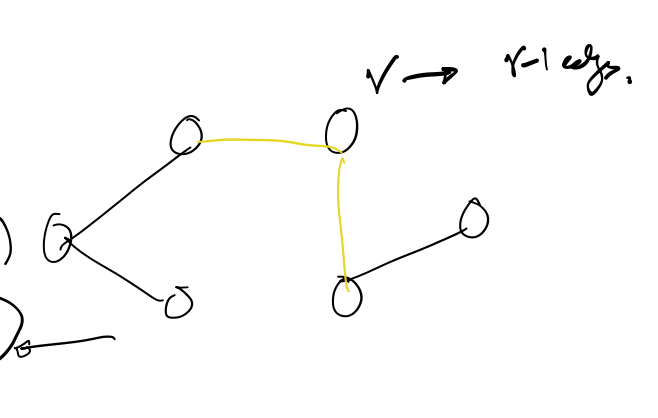
① g is a tree, if remove any edge it becomes disconnected.


tree
forest
V: N
E: N-1

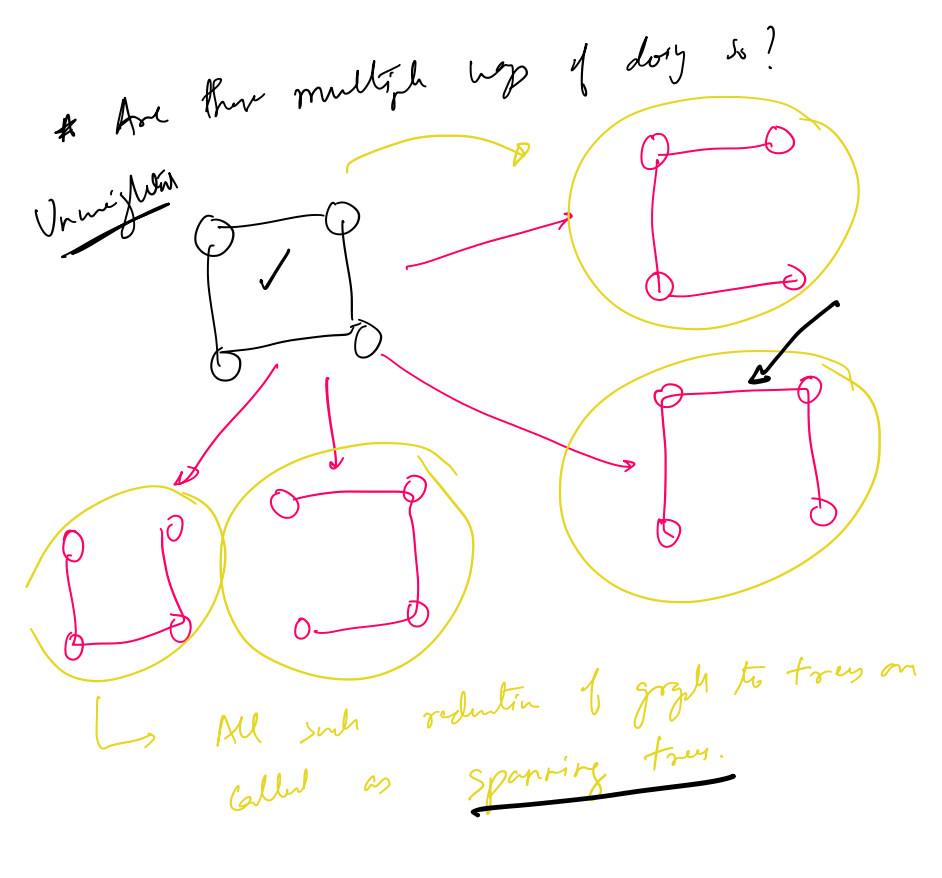
Graph (V, E) : connected.

E' = V-1

edges remove: E - (V-1)

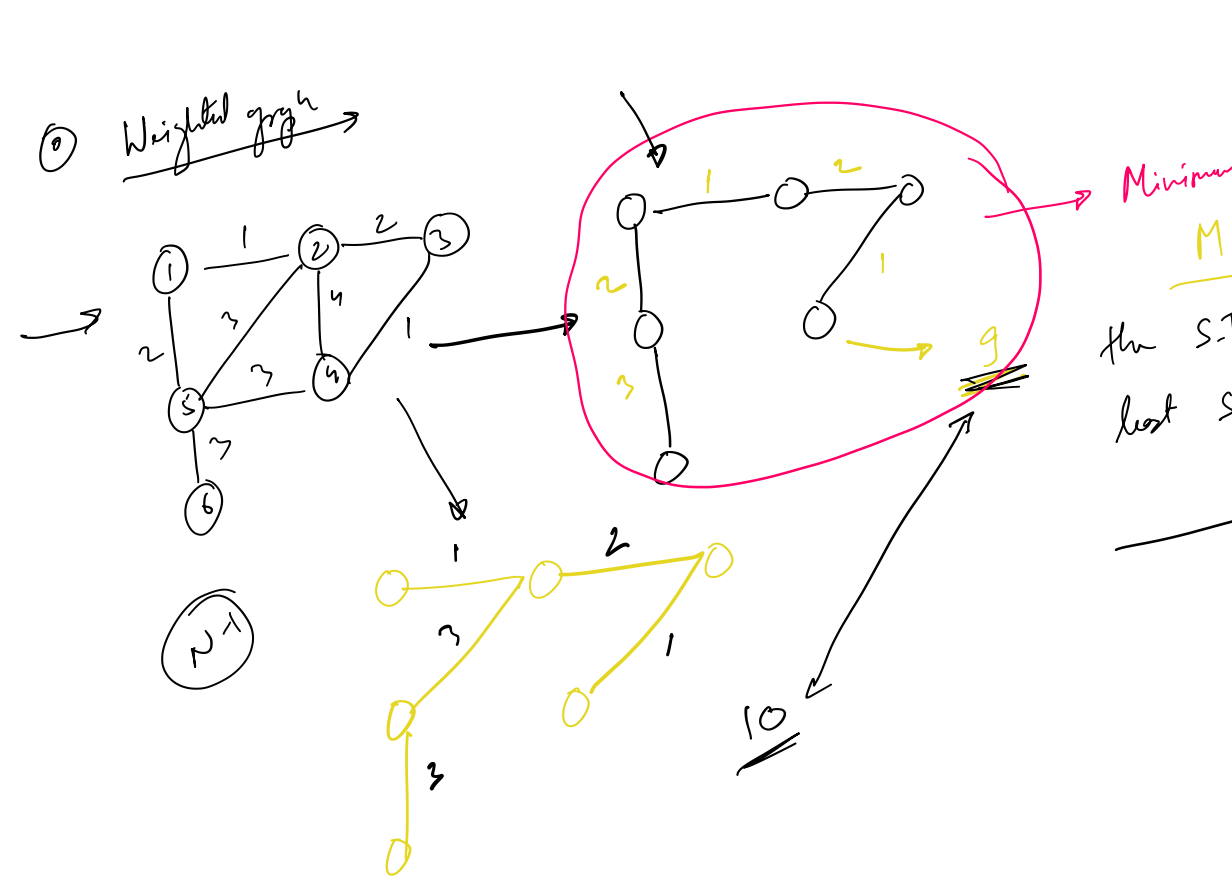


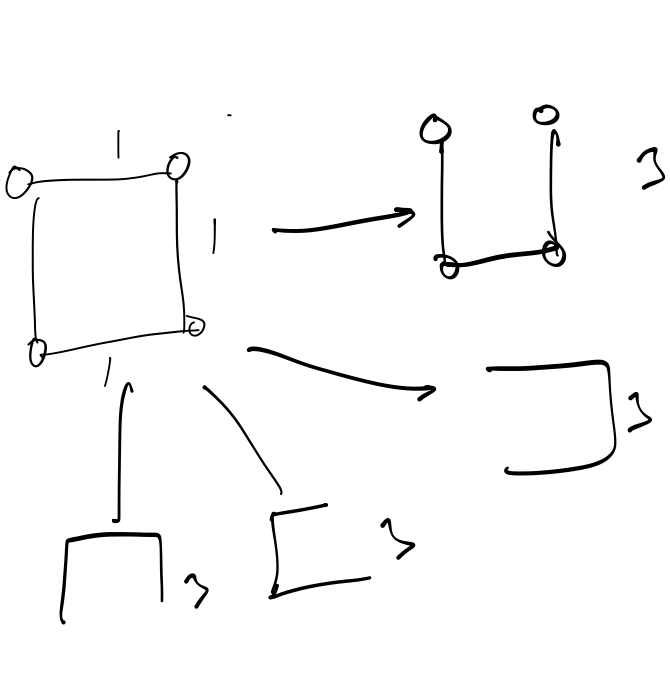
* Are there multiple ways of doing it? Unweighted



All such subsets of graph to form are called as spanning tree.

① Weighted graph


Minimum Spanning Tree
MST
the S.T. having the least sum of weights.

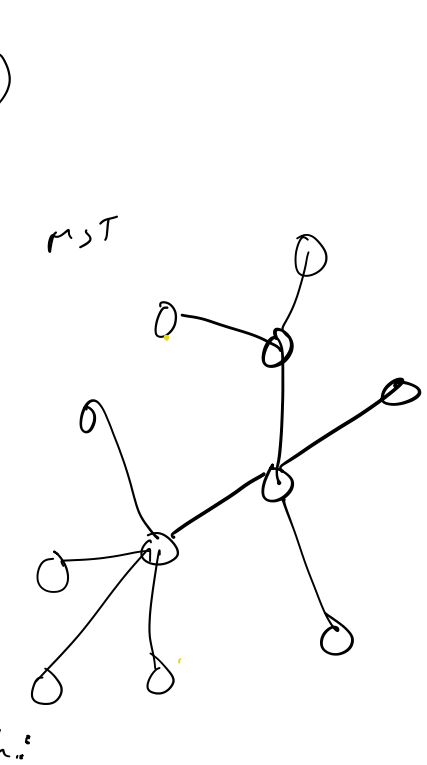


Prop. of MST :

1) If I remove an edge from a MST → forest (2 CC's)

2) If I add another edge to MST → You'll get a cycle

NOTE: MST : edge weight ≥ 0



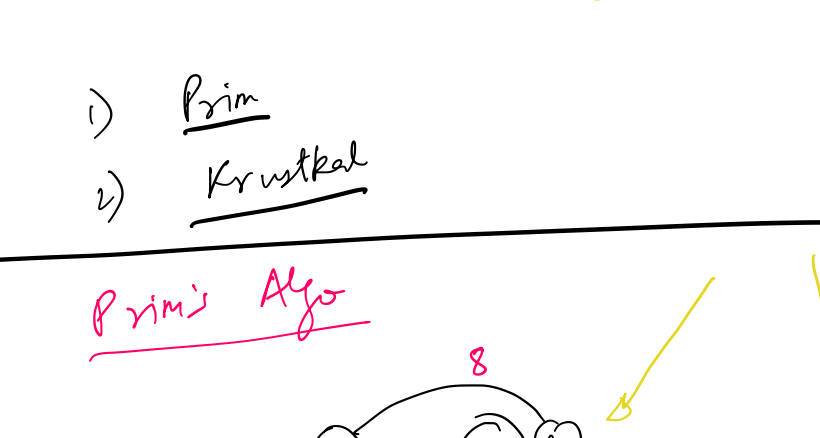
① Weight of MST of Unweighted graph: → V-1

② There could be multiple MSTs.

① Two Algo's for finding MST in a weighted graph!

1) Prim
2) Kruskal

1) Prim's Algo

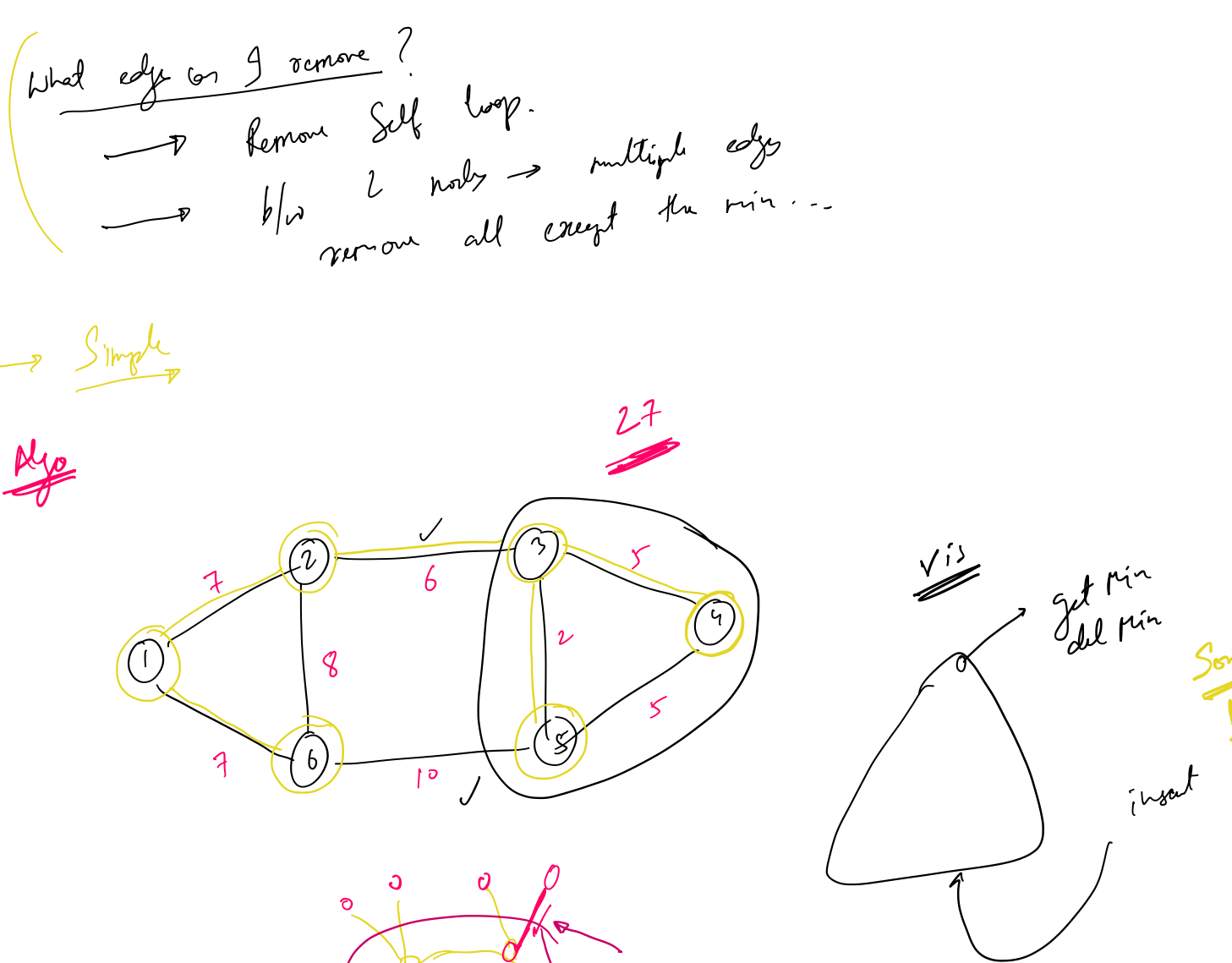


Simple graph:
- No multi edge
- No self-loop

What edges can I remove?
→ Remove self loop.
→ b/w 2 nodes → multiple edges, remove all except the min...

→ Simple

Algo


27
vis
get min
del min
insert
Sorted set
Min heap

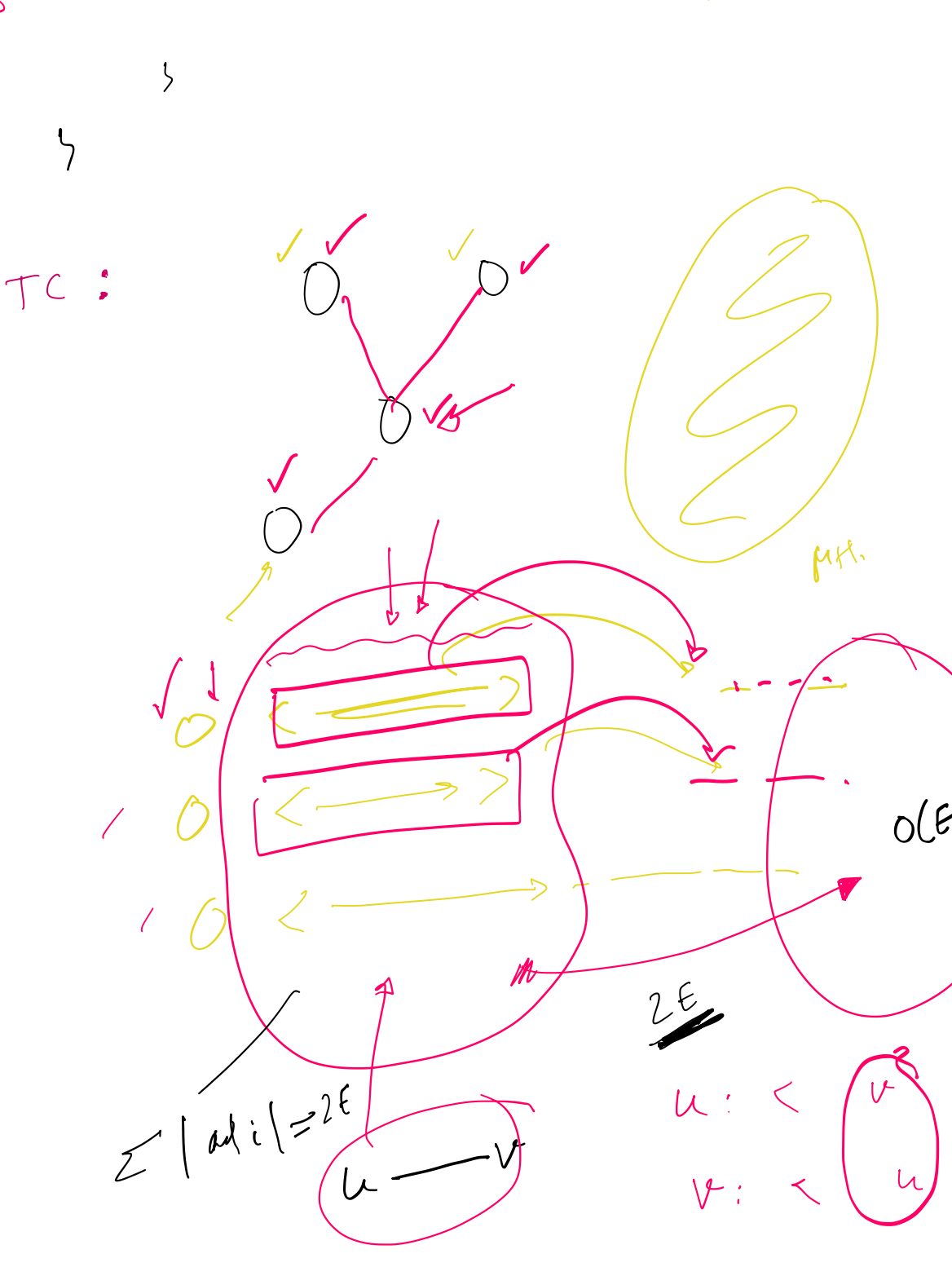
Algo: mh → min heap, vis → set.
u → starting node

```
Create MST(u) {
    vis.add(u);          wt = 0
    add all the edges from u to the mh
    while (mh is not empty) {
        pop the min.
        if (! vis.contains(min.v)) {
            vis.add(min.v);
            for all neighbours of min.v
                add those edges in the min heap.
        }
    }
}
```

check...

<weight, <min, v>>

TC :


Σ |adj i| = 2E
MH
O(E)
2E
u: < v >
v: < u >
get min → O(1)
insert → O(log E)
del min

1 → O(log E)

E → O(E log E)

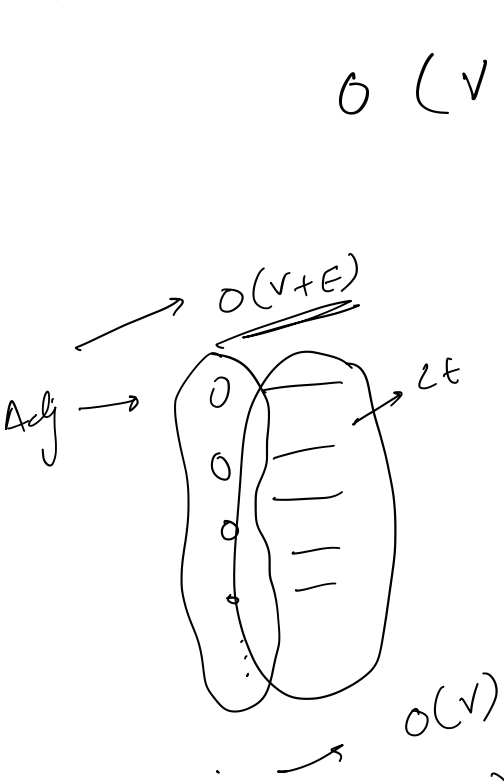
O(V + E log E)

1E → 2

E → 2E

TC: O(V + E log E)

SC: O(V+E)

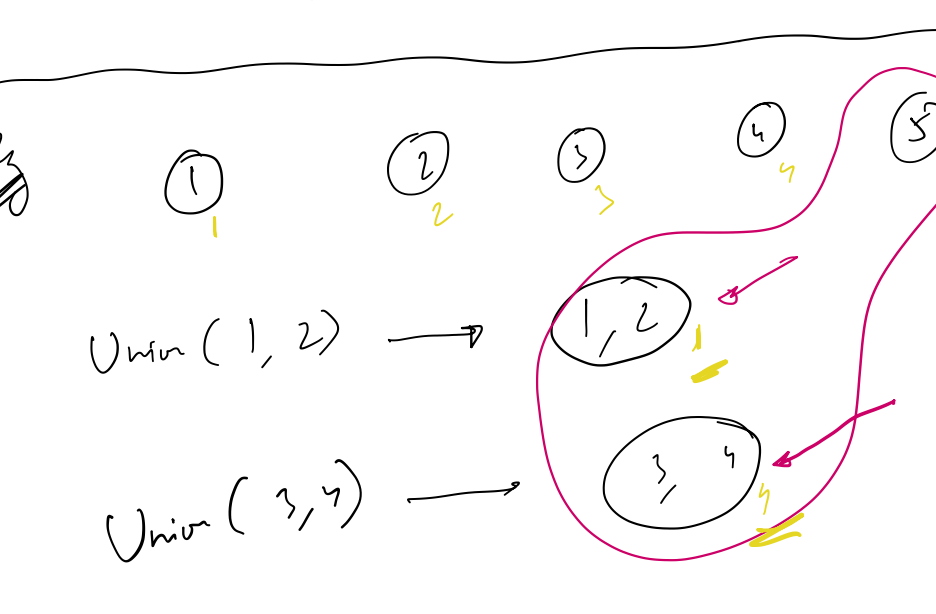


vis → O(V)

mh → O(E).

---

② Disjoint Set Union (DSU)



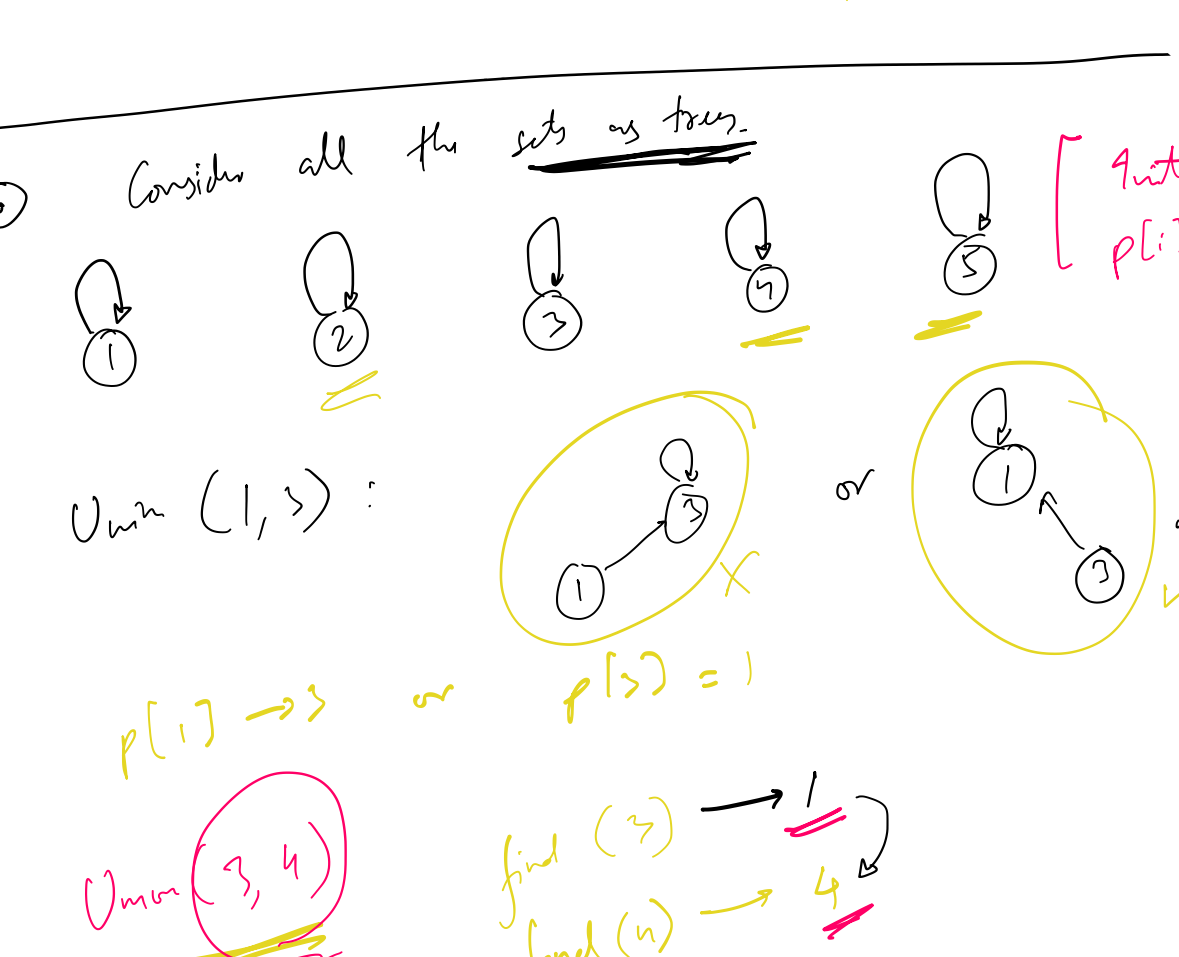
$S_1 \cap S_2 = \phi$

$S_1 \cap S_3 = \{3\}$

S1 & S2 are disjoint sets.

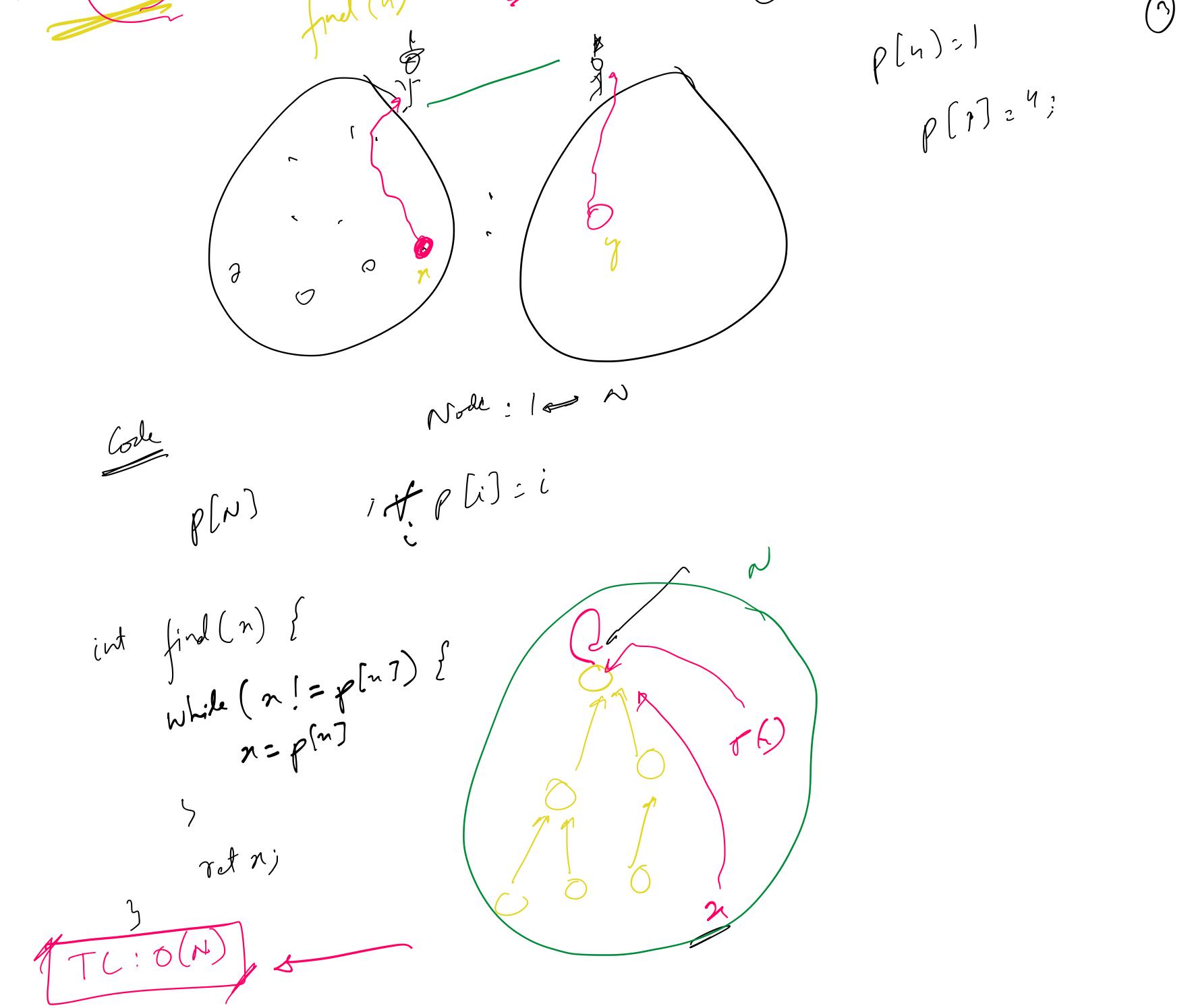
$S_1 \cup S_2 = \{1, 3, 7, ...\}$

Ex:


Union(1, 2) →
Union(3, 4) →
Union(1, 3) →
1 → 1
2 → 1
3 → 4
4 → 4
disjoint sets

① Consider all the sets as trees. [Initially p[i] = i]

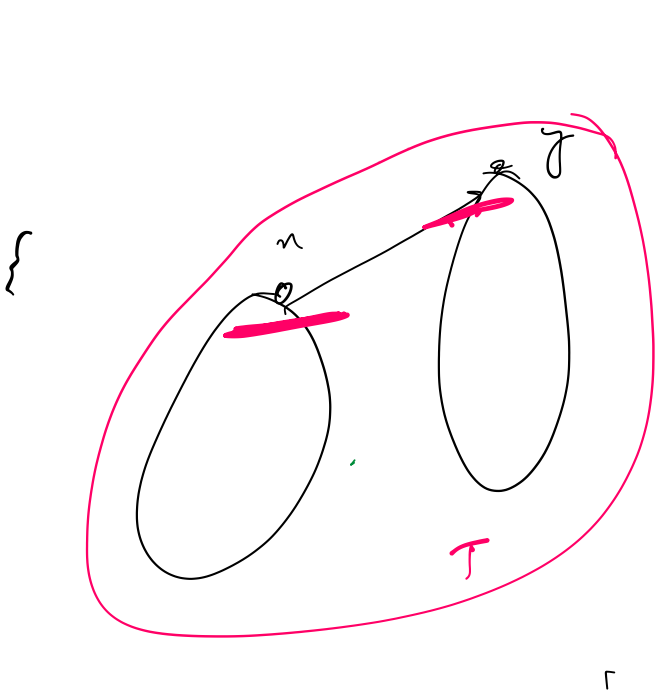
Union(1, 3) :


or
p[1] = 3 or p[3] = 1
Union(3, 4)
find(3) → 1
find(4) → 4
p[4] = 1
p[7] = 4;

Code: p[N]   Node: 1 → N
Initially p[i] = i

```
int find(x) {
    while (x != p[x]) {
        x = p[x];
    }
    ret x;
}
```

TC: O(H)



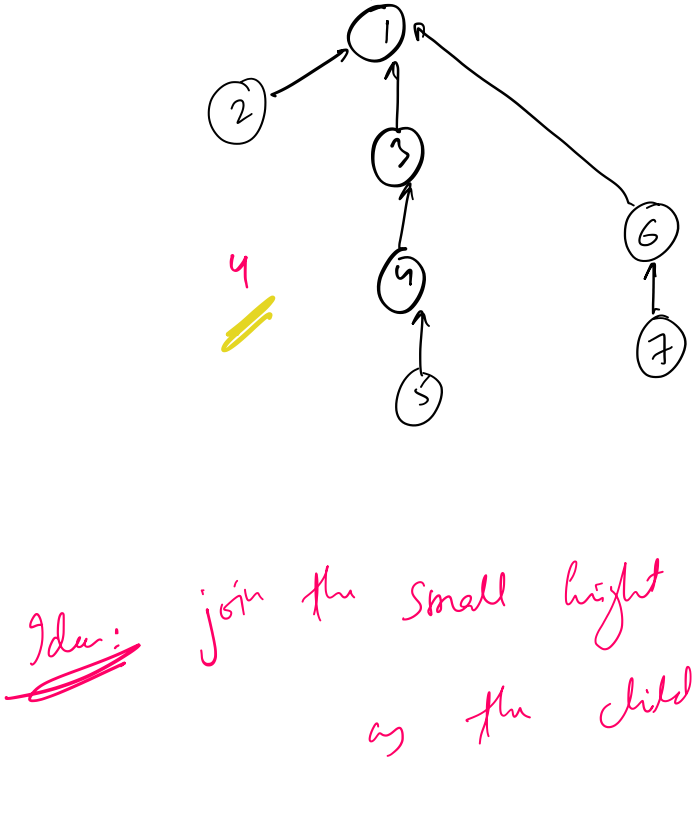
```
bool union(x, y) {
    x = find(x);      O(H)
    y = find(y);      O(H)
    if (x == y)
        ret false;
    p[x] = y;         O(1)
    ret true;
}
```

TC: O(H)

① We can reduce the TC by:
1) Union by height
2) Path Compression.

1) Union by Height

TC depends on the height

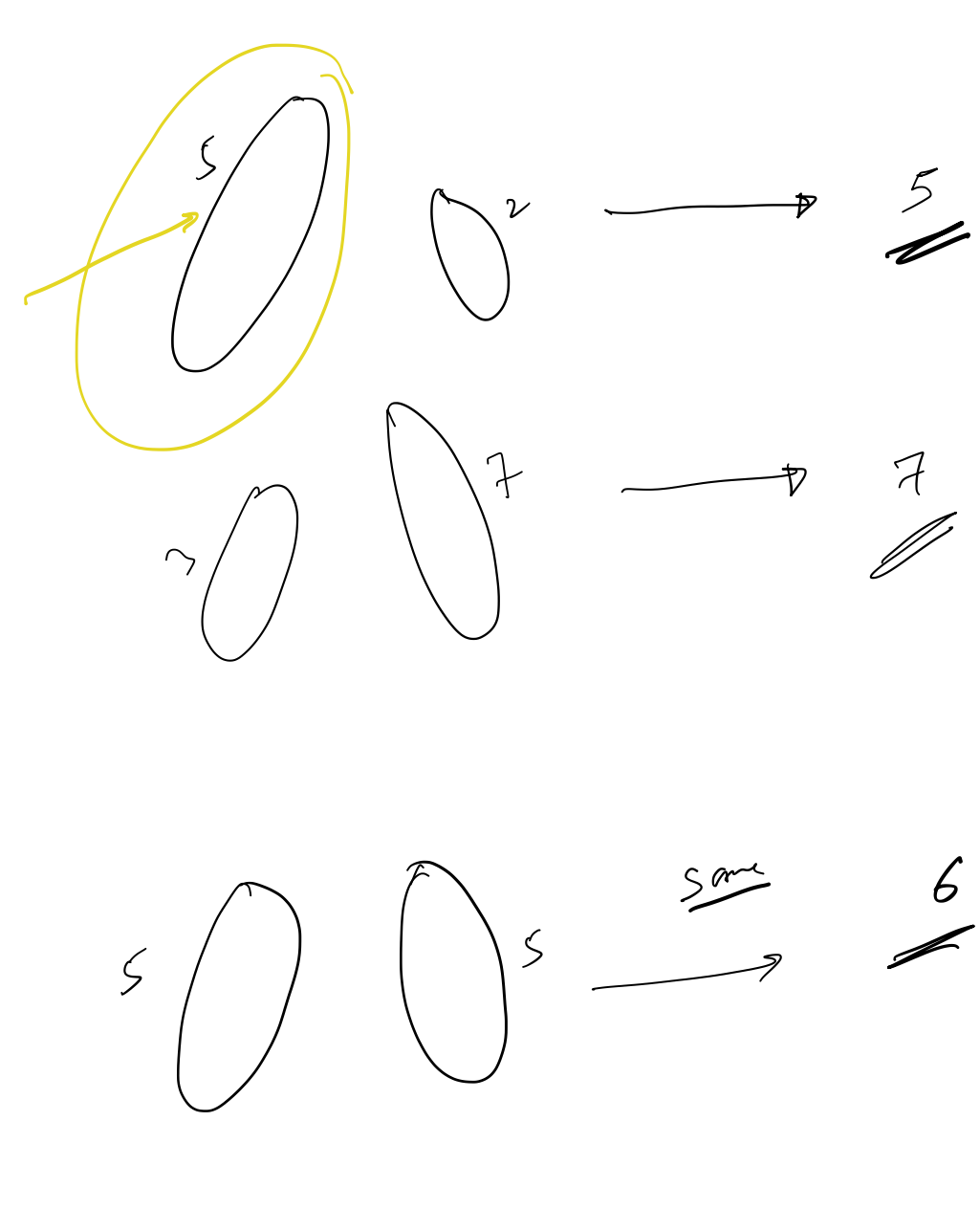


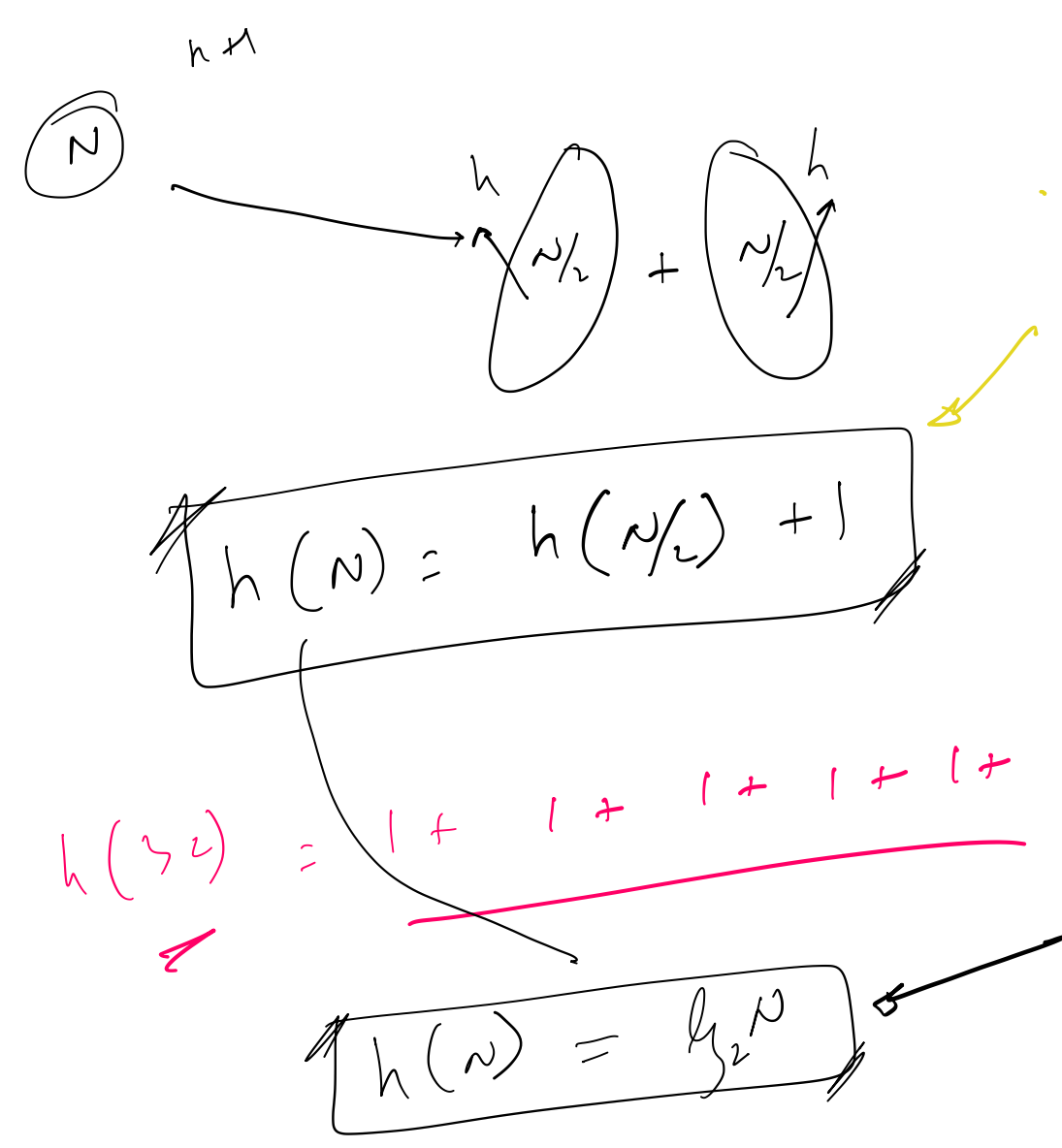
Union (5, 7)

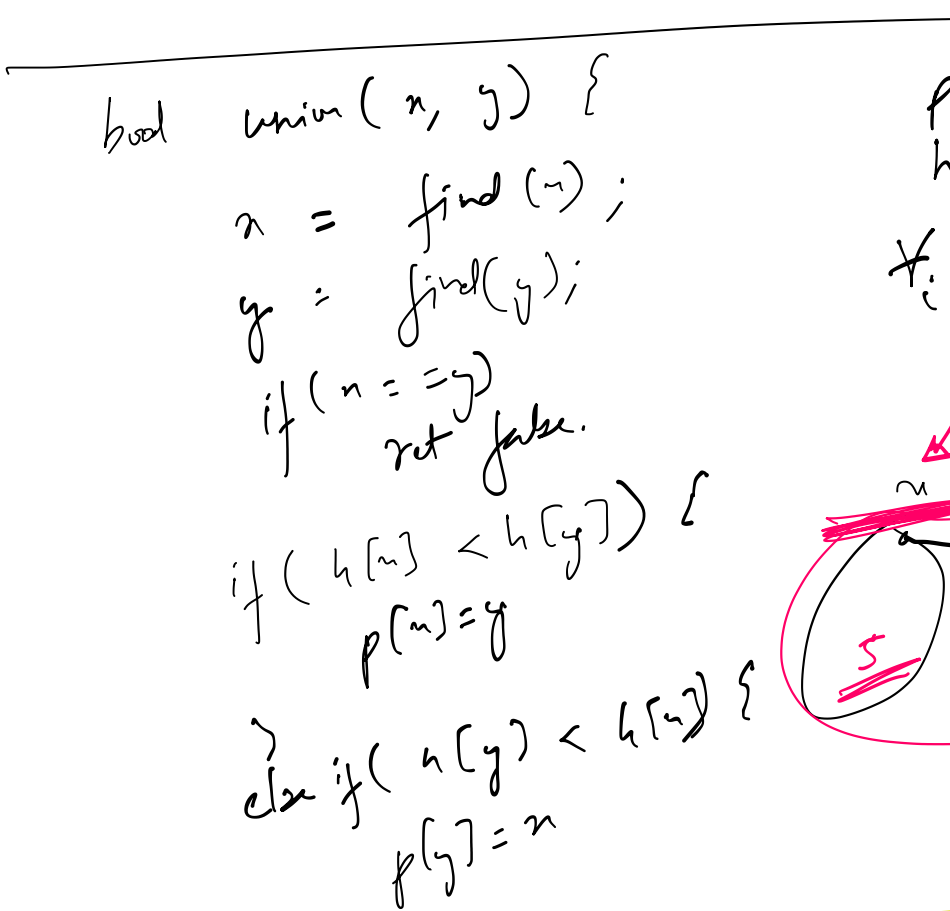
② Choices:
1) p[7] = 5 → H = 5
2) p[5] = 7 → H = 4

Idea: join the small height set as the child of big height set.

① Exp / Obs







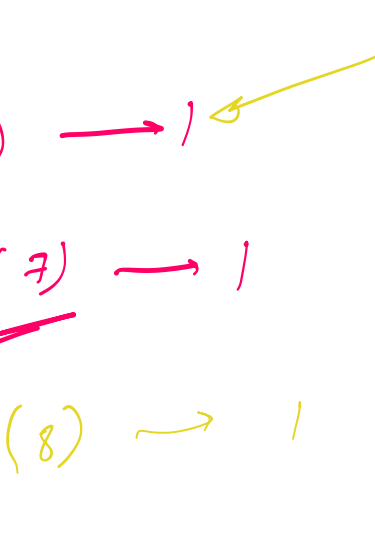
h(N) = h(N/2) + 1

h(32) = 1 + 1 + 1 + 1 + 1 + 1

$h(N) = \log_2 N$

```
bool union(x, y) {
    x = find(x);
    y = find(y);
    if (x == y)
        ret false.
    if (h[x] < h[y]) {
        p[x] = y;
    }
    else if (h[y] < h[x]) {
        p[y] = x;
    }
    else {
        p[y] = x;
        h[x]++;
    }
    ret true;
}
```

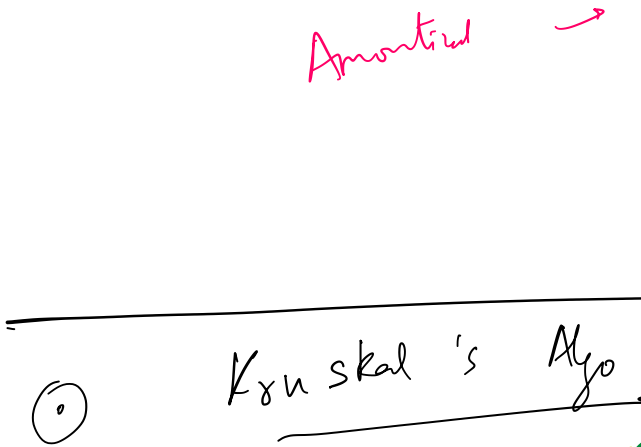
p[ ]
h[ ]
∀ h[i] → 1



TC find: O(log N)

TC Union: O(log N)

2) Path Compression

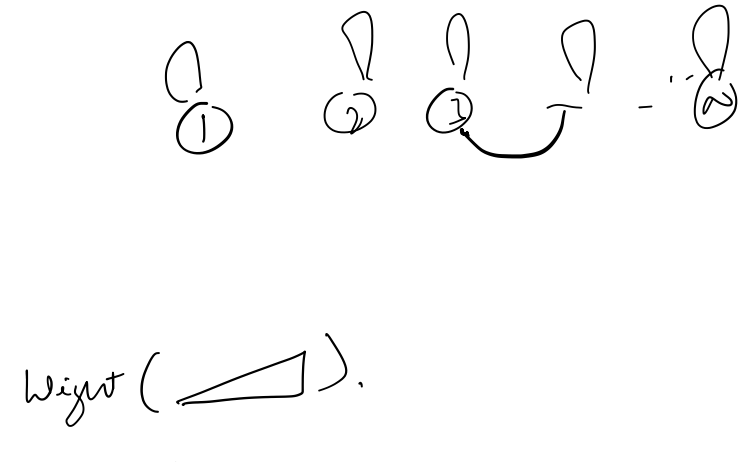


find(9) → 1

f(7) → 1

f(8) → 1

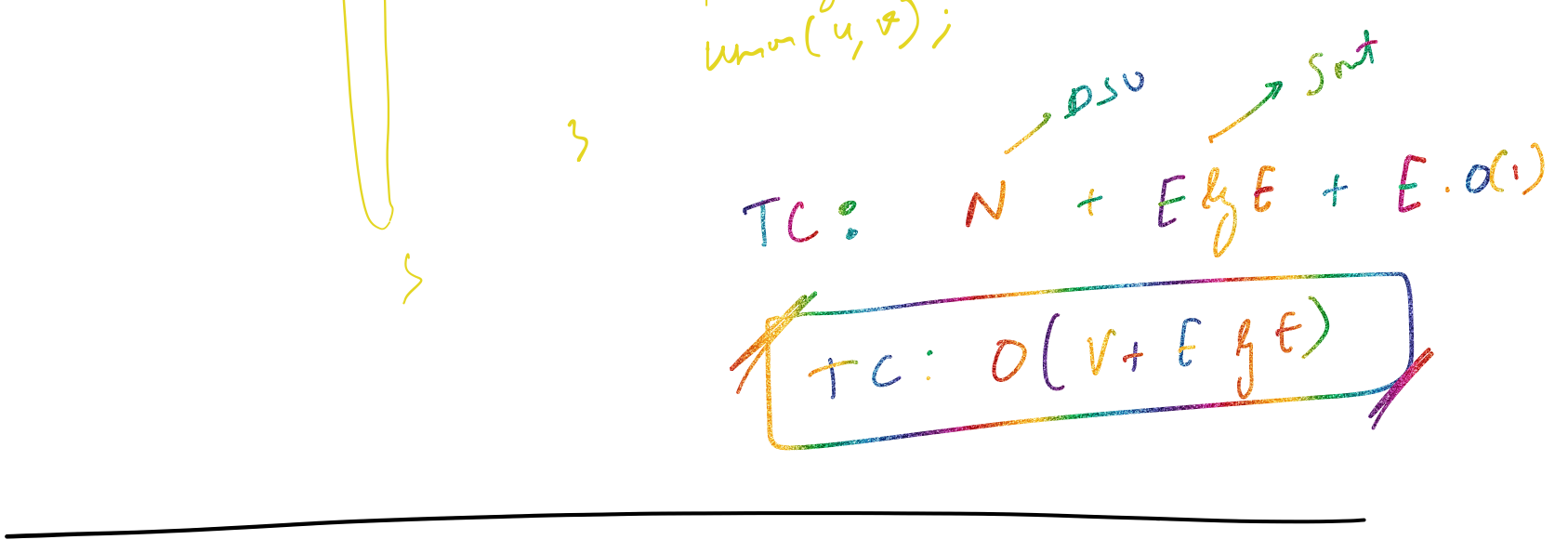
```
int find(x) {
    if (x == p[x])
        ret x;
    p[x] = find(p[x]);
    ret p[x];
}
```



④ TC for find / Union
Amortized → O(1)

path comp + union

① Kruskal's Algo


find
find
≠
add edge

Algo:

```
Create DSU (n)
Sort all edges acc. to weight.
Loop over the sorted list:
    (w, u, v)
    if (find(u) != find(v)) {
        this edge would be a part of MST!
        union(u, v);
    }
```

TC: N + E log E + E · O(1)

TC: O(V + E log E)

① Applications of DSU?

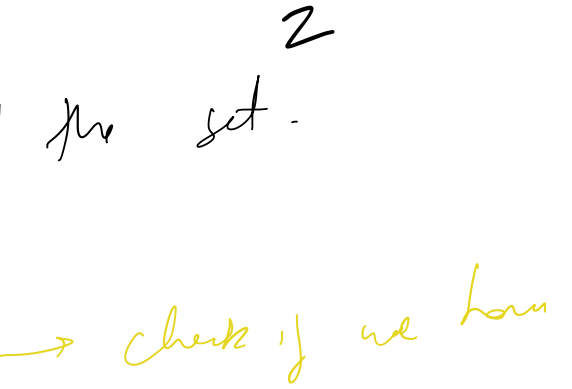
1) Graph → Check if it is connected or not

O(V+E)

∀ edge (u, v): union(u, v)

∀ node: find(i) → Constant!

p(i)



2) How many CC's are there?

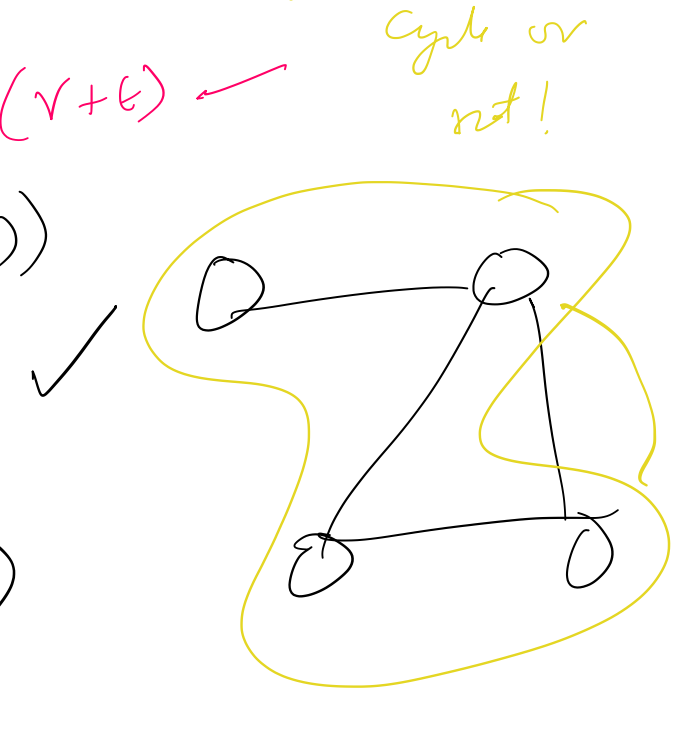
O(V+E)

∀ edge (u, v): union(u, v)

set

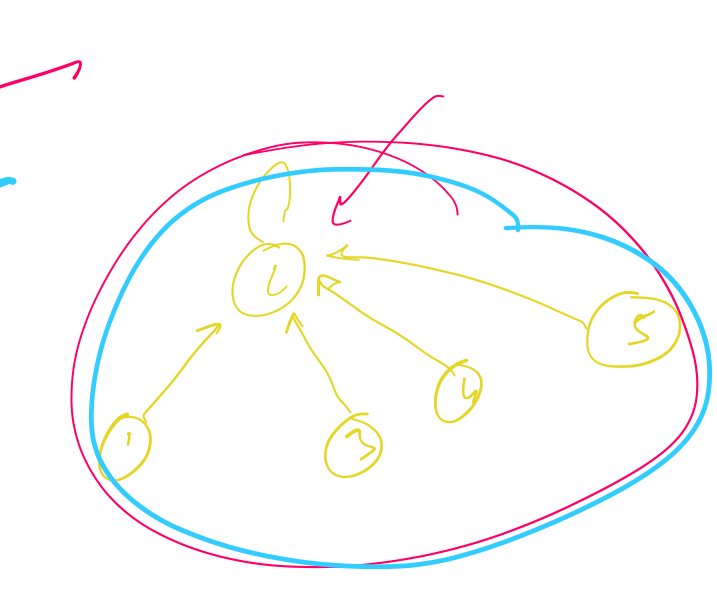
∀ node: set.add(find(node)).

CC's → Sz of the set.



3) Given a graph → Check if we have cycle or not!

O(V+E)

∀ edge (u, v):
if (find(u) == find(v)) → cycle ✓
else union(u, v)