Goal

- Understanding Disjoint Set Union ✓
- Union by Size
- Union by Rank
- Path Compression

optimisations

- Applications of DSU
- Problem Solving

Task:

Given a graph with n nodes and q queries where n, q <= 10^5. The queries are of 2 types:

1. Make an edge between node i and j
2. Given i and j, print YES if they are in the same connected component else no

10 6

unite 1 2

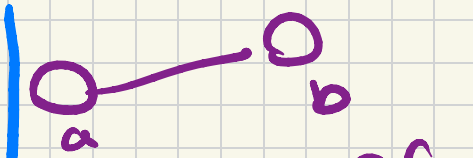
ask 1 2 → Yes

unite 3 4

unite 3 2

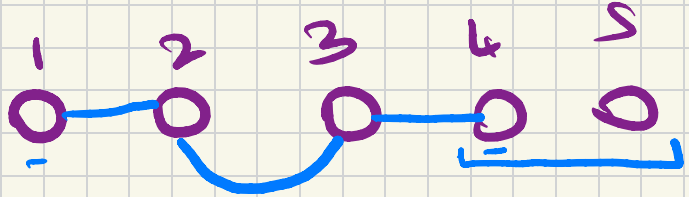
ask 4 1 → Yes

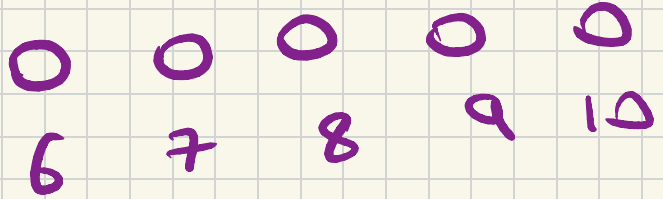
ask 5 4 → No



same cc

No





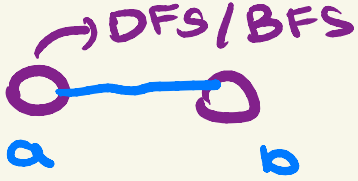
unite 1 2

ask 1 2

unite 3 4

unite 2 3

ask 1 2

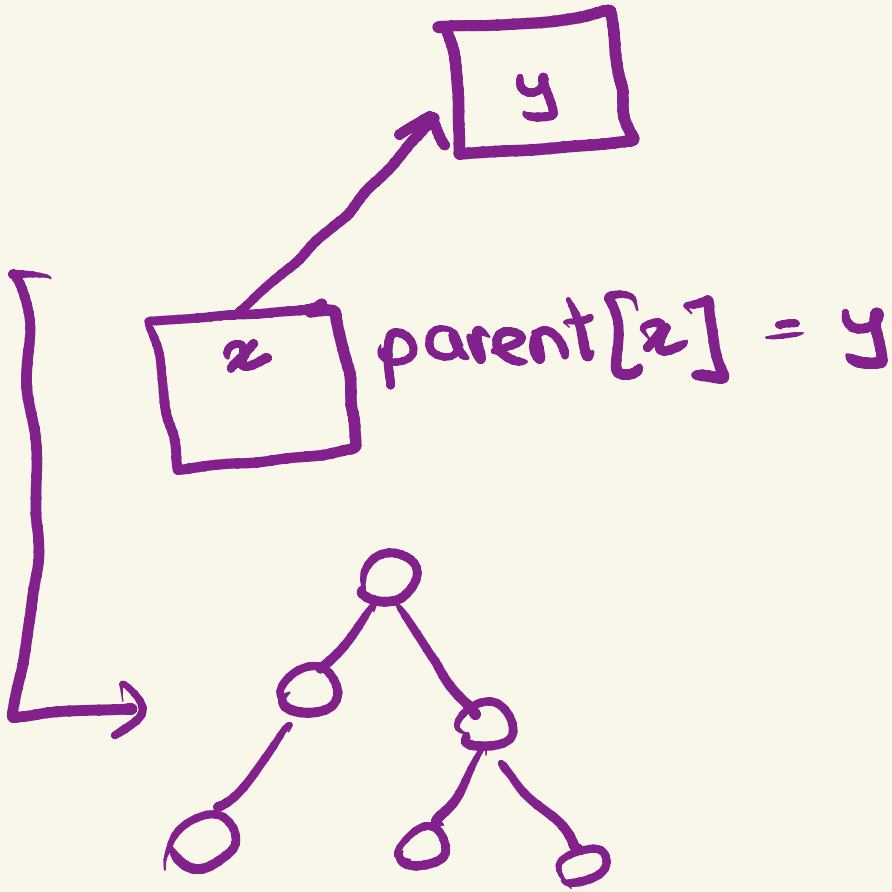


0 0

Naive approach

TLE

unite a b



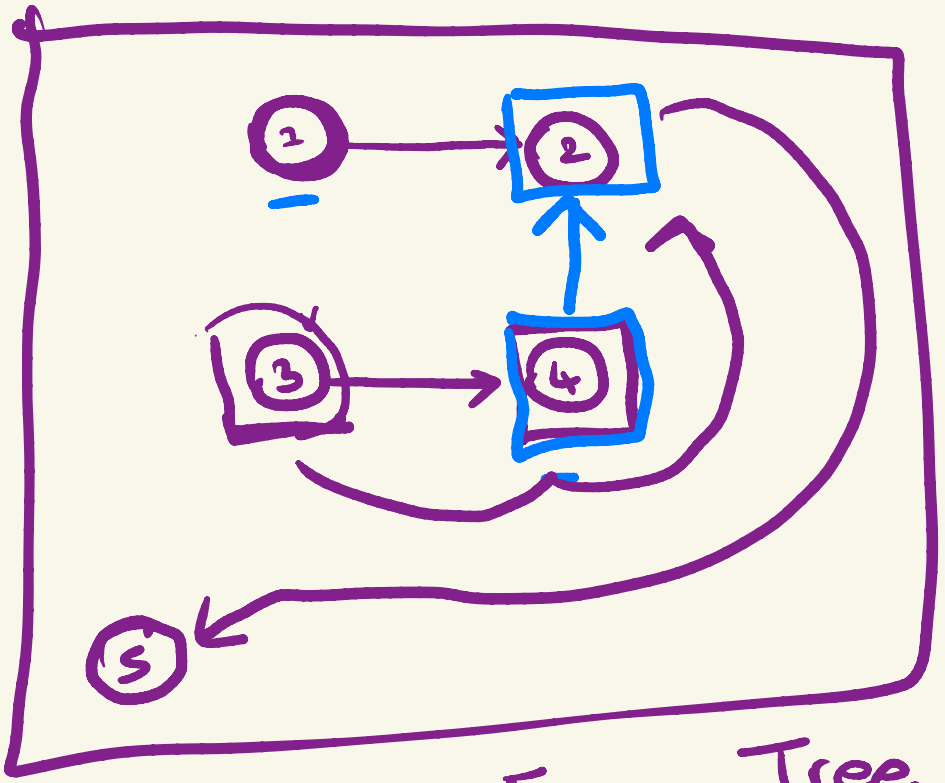
unite 1 2

unite 3 4

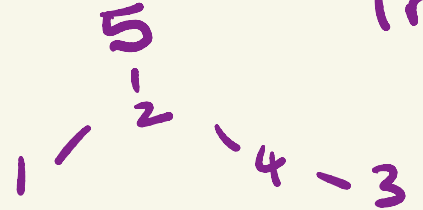
unite 3 2

unite 1 4

unite 3 5



Tree

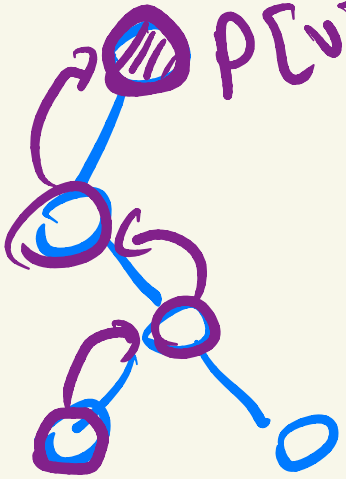


#Why is this a tree structure?

→ We only add edge between un-connected components

1	2	3	4	5
1	2	3	4	5

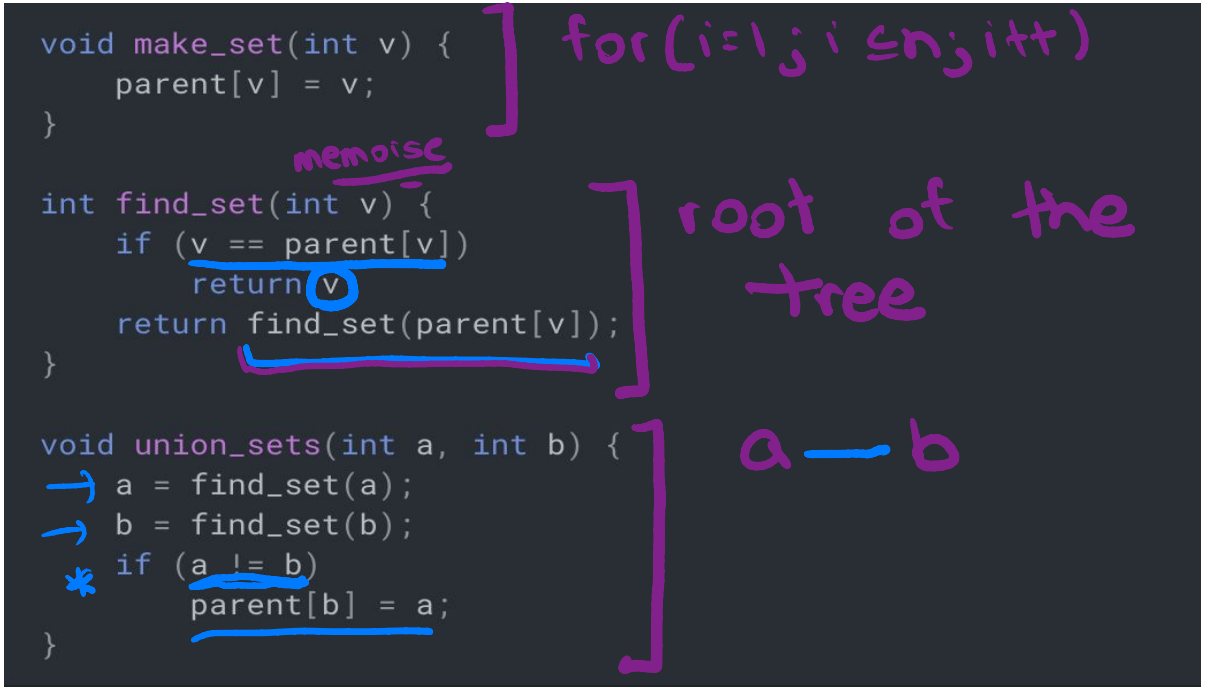
parent



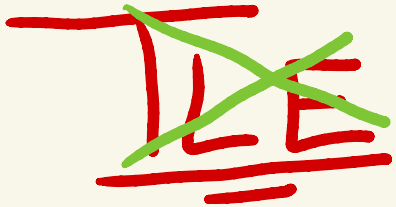
same CC

find_set(a) = find_set(b)

Naive Implementation of DSU

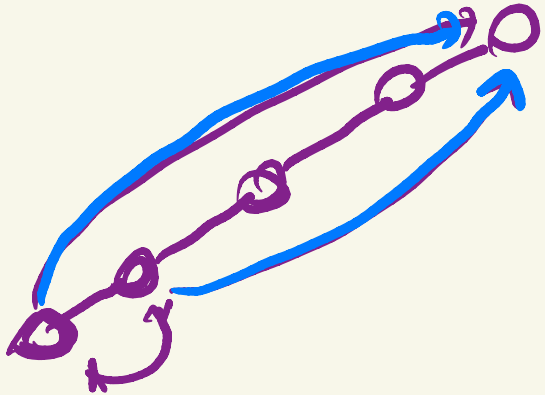


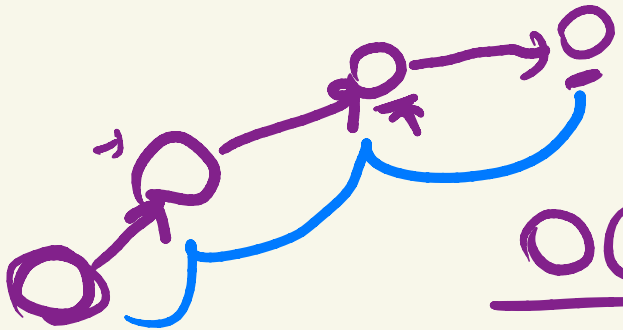
Time complexity: Naive



unite (a, b) ⟶ O(n)

find-set (a, b) → O(n)

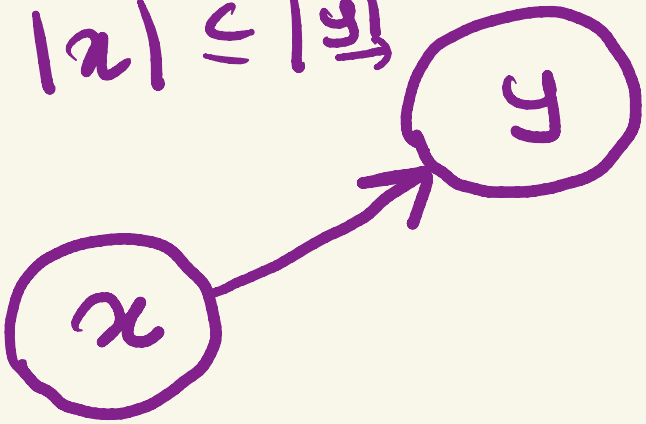




O(n)

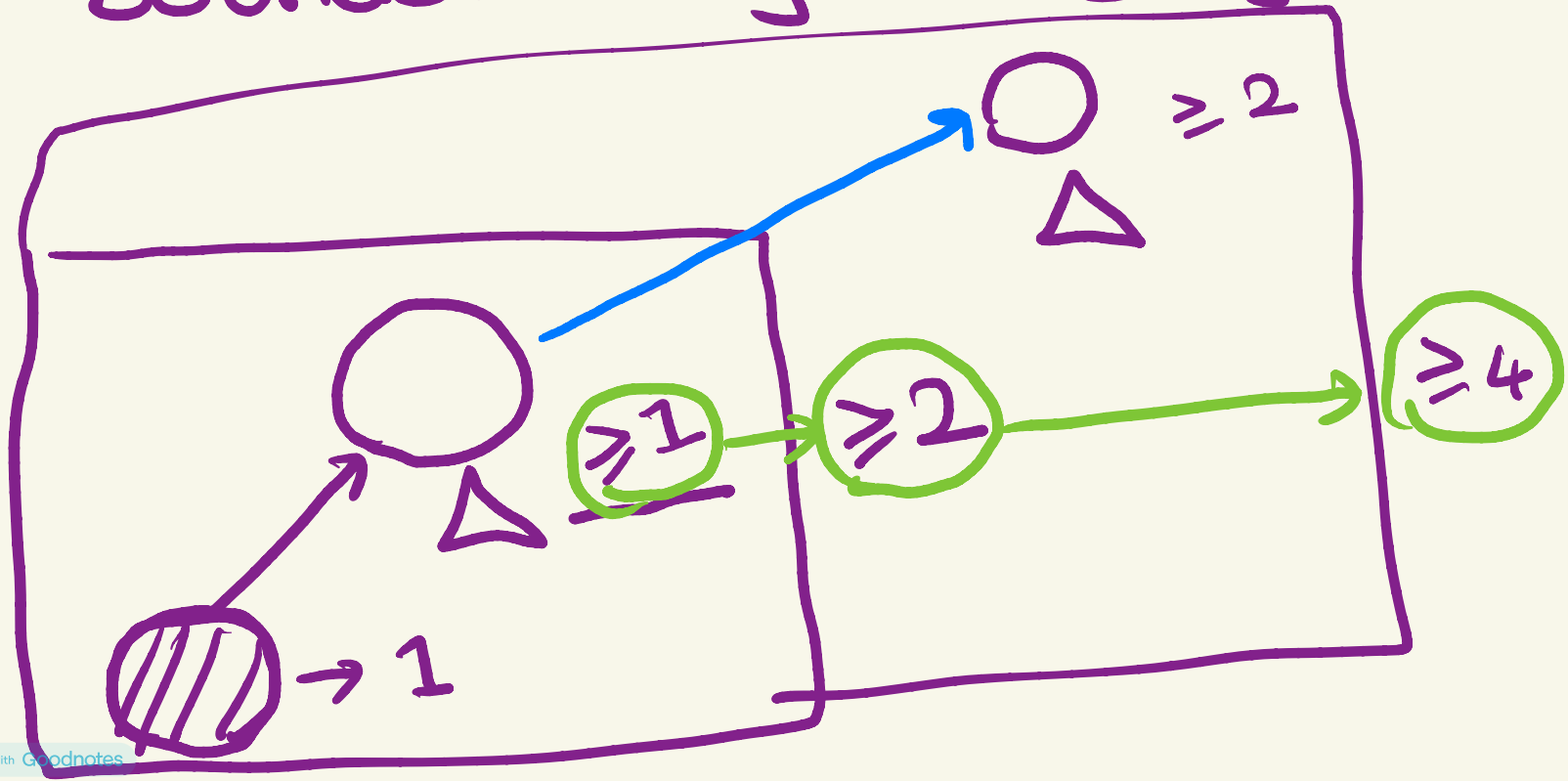
Union by size:

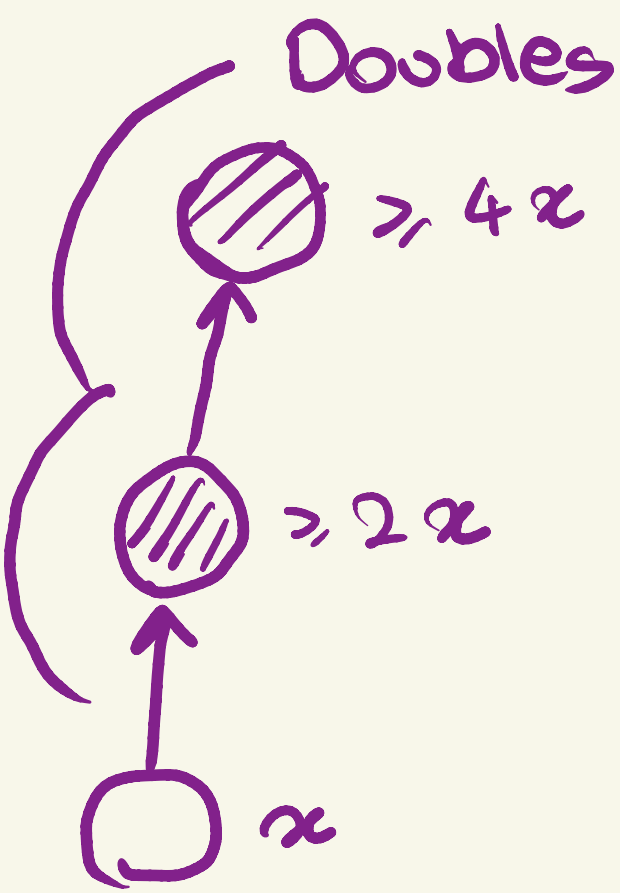
$|x| \leq |y|$

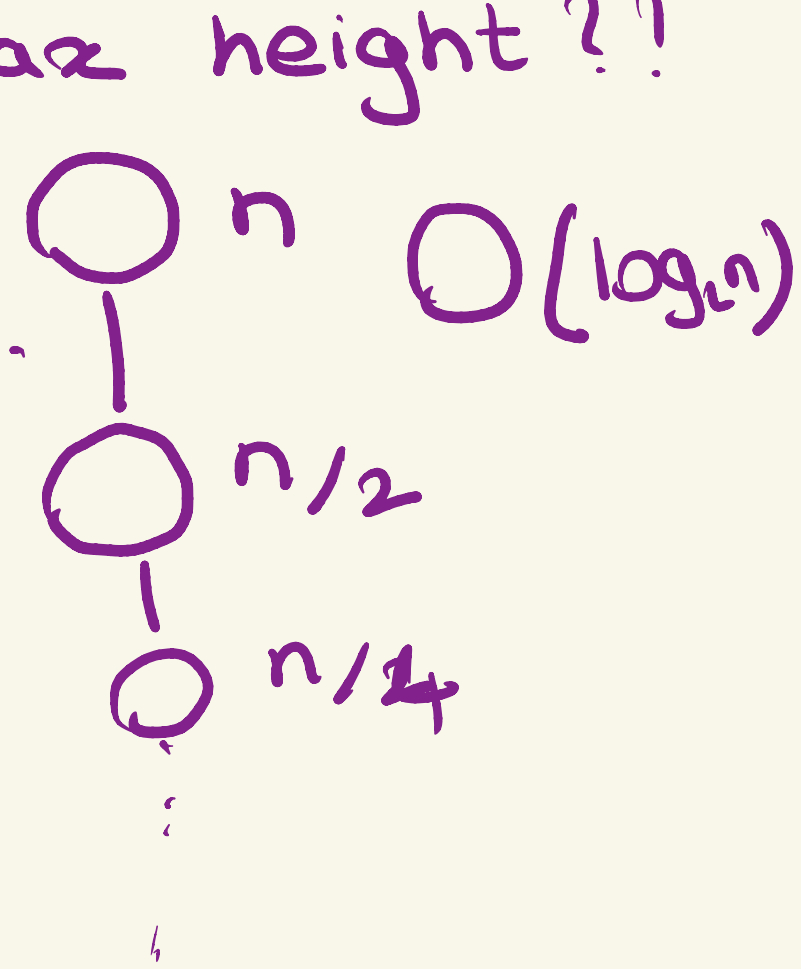


How is this helpful ??

Height of the tree is bounded by $O(\log_2 n)$







Union by Size

find_set(x) $O(\log_2 n)$

unite(a, b) $O(\log_2)$

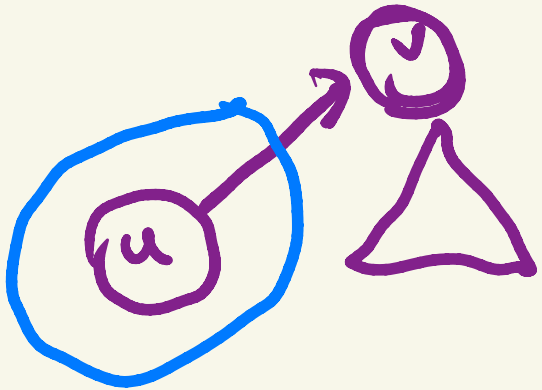
Size

1	1	1	1	1	1

roots of components

roots of components

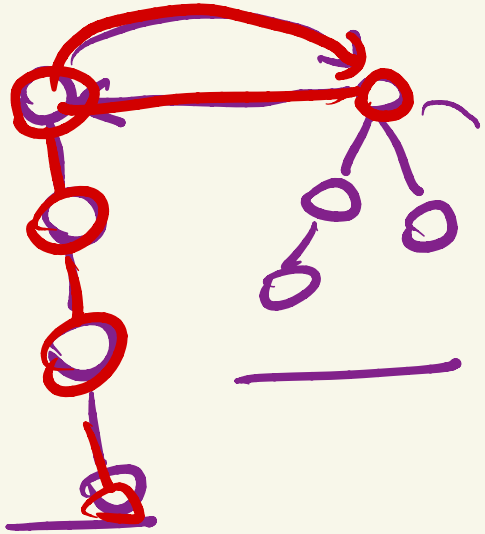
$S_2[v] \; t = S_2[u]$



$O(1)$

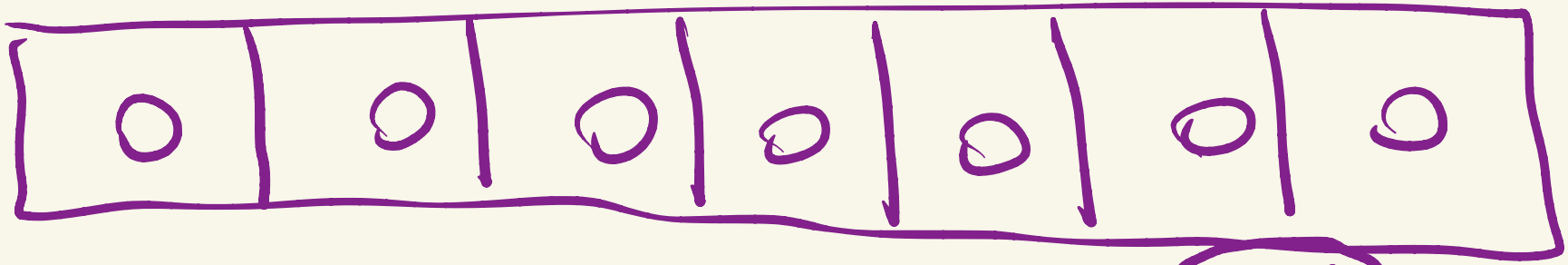
Size $\Longrightarrow$ Height of the tree

$S_2 = 4$



Union by Rank

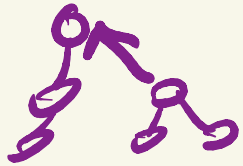
height



$R_u = R_v$

R_v++

$R_u[u] > R[v]$



Union by Size / Rank

- If we want to merge 2 components, we would like to merge the smaller component to the larger component
- This allows us to have at most O(logn) height of our DSU Tree
- Proof:

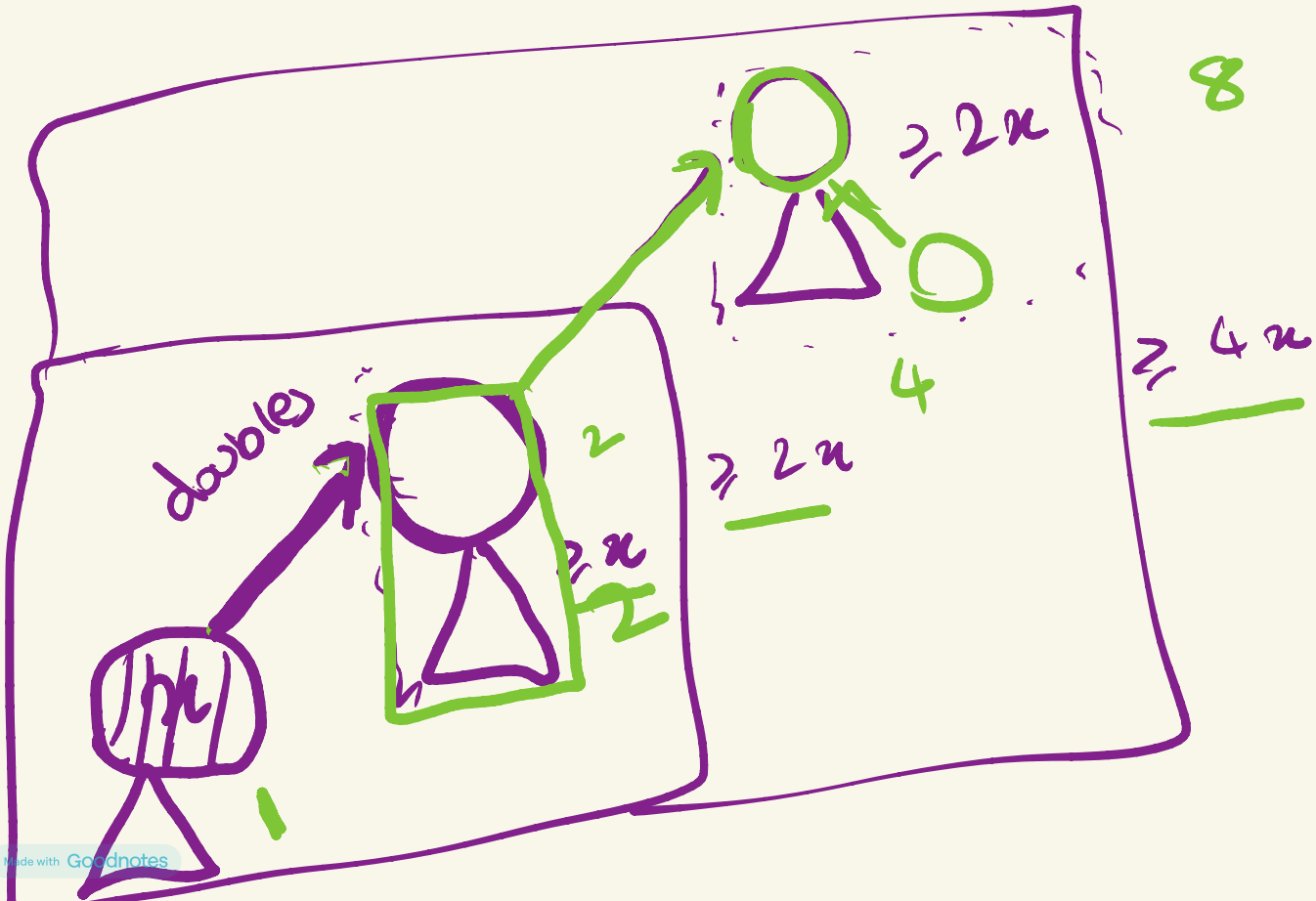
  Consider a component of size x, it would merge into a component of size x. Now it’s size is 2x. This component would merge into a component of size 2x. Post merging the size becomes 4x.

  Basically after each merge, the size doubles. Hence ~logn merges

Union by Size Implementation

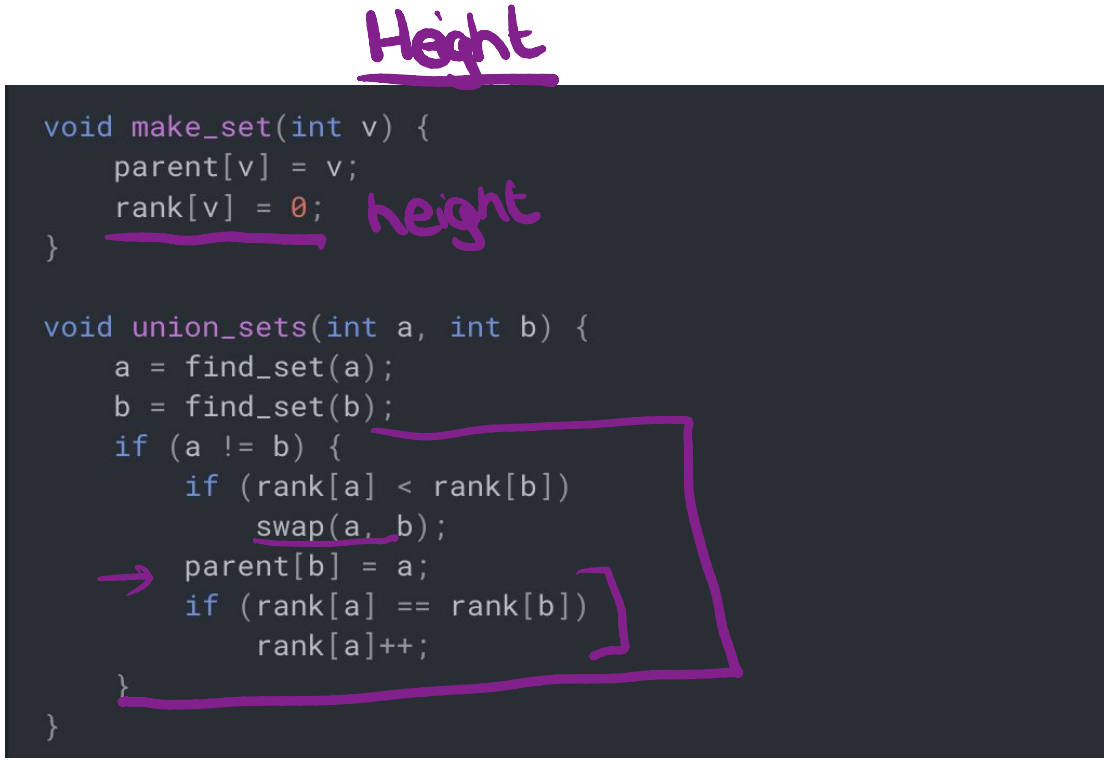
```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b])
            swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
```


doubles
2
1
8
4
≥ 2n
≥ 2n
≥ 4n

$O(\log_2 n)$

Union by Rank Implementation



1 2

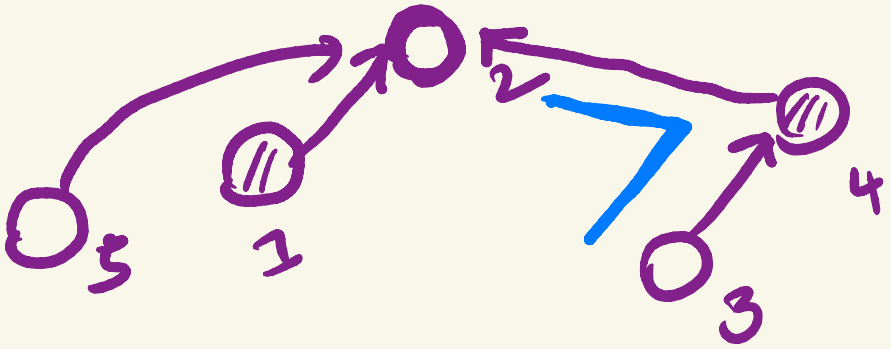
3 4

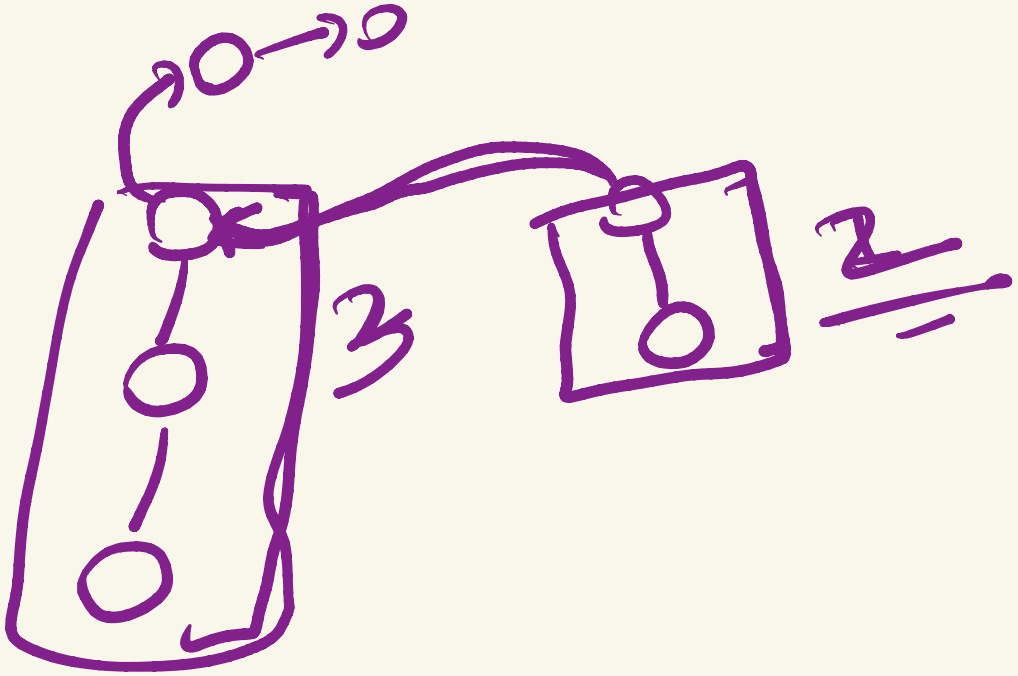
4 1

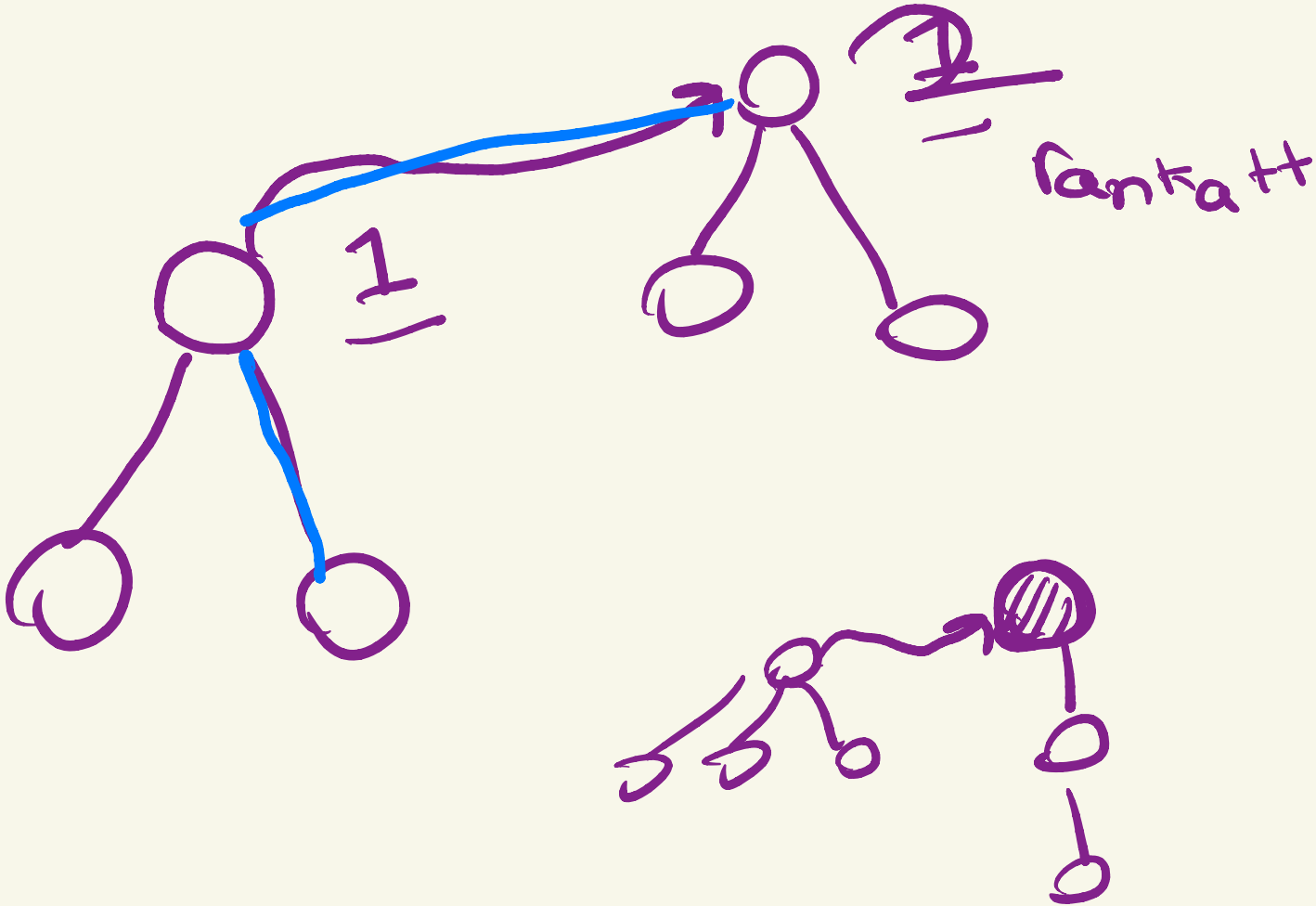
5 4

Rank

0	2	0	1	0
1	2	3	4	5





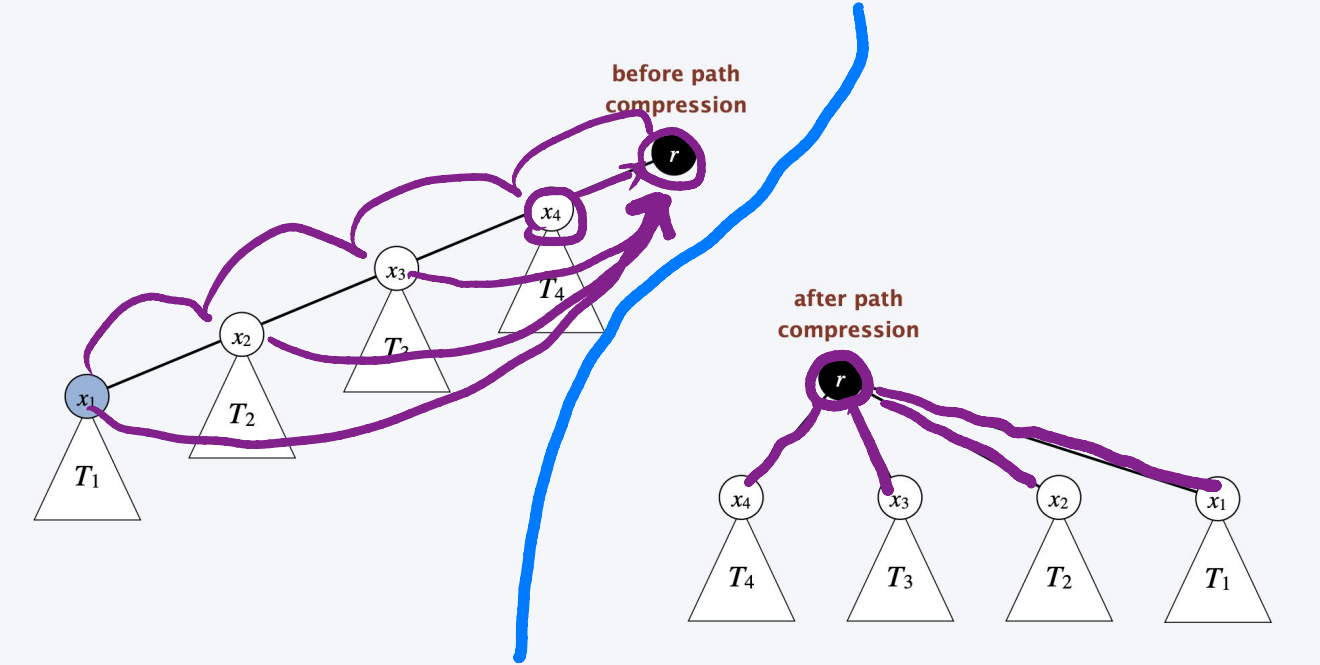


Path Compression

- In path compression, we “memoise” the find function.
- Essentially when finding the root of a particular node x, we update the root of all parents of node x to the root.
- This alone optimises our time complexity to O(logn) on average.
- Proof: http://e-maxx.ru/bookz/files/dsu/Worst-Case%20Analysis%20of%20Set%20Union%20Algorithms.%20Tarjan,%20Leeuwen.pdf


```
find_set(x) {
    if p_x = x    return x
    return p_x = find_set(p_x)
}
```

Illustration of Path Compression

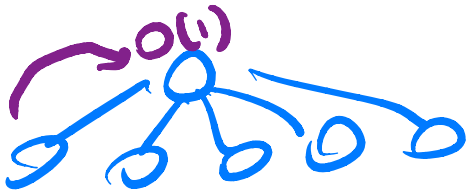


Path Compression Implementation

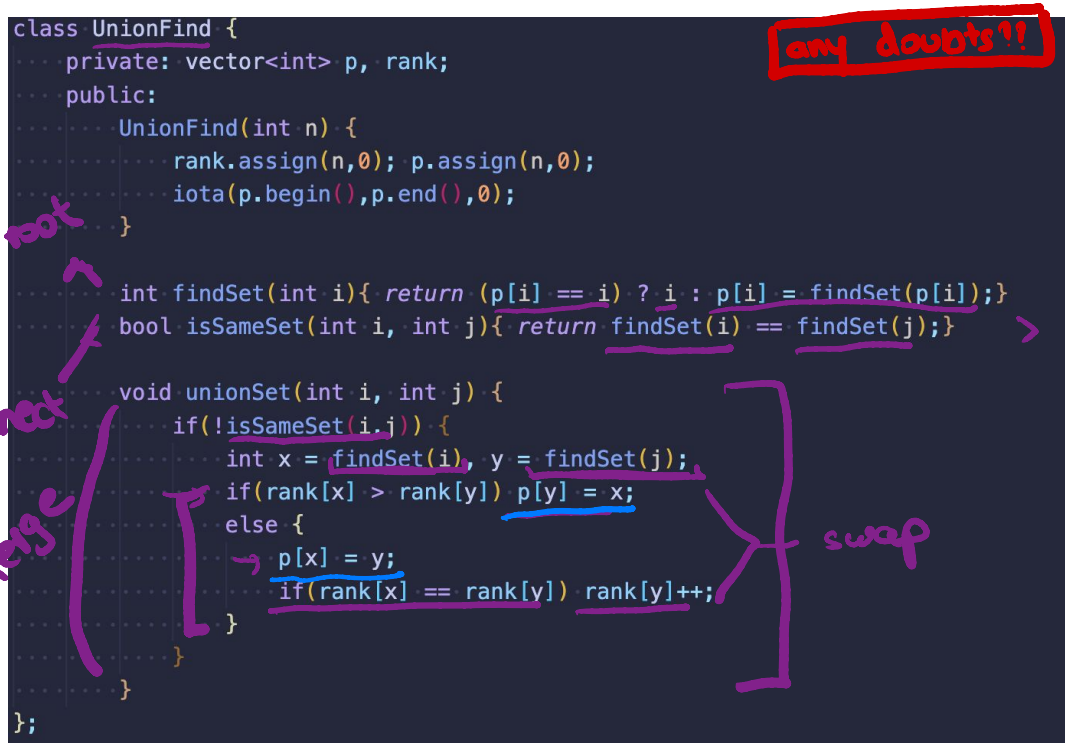
```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}
```

Using Both Optimizations Together

- Using both optimizations together, i.e.
  A. Union by Rank / Size
  B. Path Compression
  optimizes our time complexity to O(α(n)) per query
- Proof:
  https://codeforces.com/blog/entry/98275



Implementation of DSU (Optimized)




```
class UnionFind {
    private: vector<int> p, rank;
    public:
        UnionFind(int n) {
            rank.assign(n,0); p.assign(n,0);
            iota(p.begin(),p.end(),0);
        }

        int findSet(int i){ return (p[i] == i) ? i : p[i] = findSet(p[i]);}
        bool isSameSet(int i, int j){ return findSet(i) == findSet(j);}

        void unionSet(int i, int j) {
            if(!isSameSet(i,j)) {
                int x = findSet(i), y = findSet(j);
                if(rank[x] > rank[y]) p[y] = x;
                else {
                    p[x] = y;
                    if(rank[x] == rank[y]) rank[y]++;
                }
            }
        }
};
```

1. Union Find DSU(n); Blackbox

2. DSU.findSet(a,b)
   DSU.isSameSet(a,b)
   DSU.unionSet(a,b)

$O(\alpha(n)) \approx O(1)$

Applications of DSU

- ✓ Quick Merging of 2 Connected Components in a Graph
- ✓ Find Number of Connected Components
- ✓ Store data about each connected component → Size
- * Using DSU in Kruskal's Algorithm to Find Minimum Spanning Tree

DSU

→ (a+b+c+d) = (a+b) + (c+d)

* Size $S_{2x} + S_{2y}$
* Sum $x + y$
* Min $\min(x,y)$
* Max $\max(x,y)$
* Gcd $\gcd(x,y)$
* Bitwise And
* Bitwise Or
* Bitwise Xor

DSU

n q

unite a b

number of CC in the new graph

1 [1 2] → 4

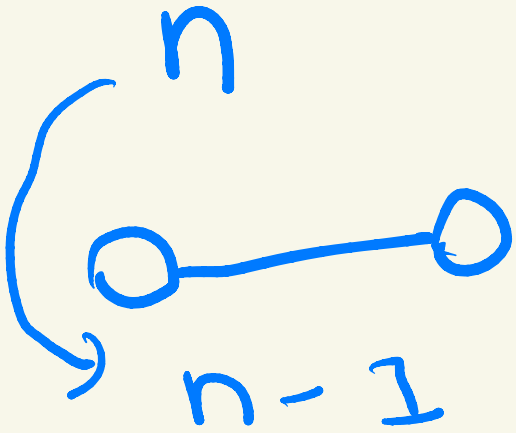
2 [3 4] → 3

3 [4 1] → 2

4 [2 3] → 2

5 [5 1] → 1

How to approach this!!



Problem Solving

https://codeforces.com/problemset/problem/151/D

D. Quantity of Strings

time limit per test: 2 seconds
memory limit per test: 256 megabytes

Just in case somebody missed it: this winter is totally cold in Nvodsk! It is so cold that one gets funny thoughts. For example, let's say there are strings with the length exactly n, based on the alphabet of size m. Any its substring with length equal to k is a palindrome. How many such strings exist? Your task is to find their quantity modulo 1000000007 $(10^9 + 7)$. Be careful and don't miss a string or two!

Let us remind you that a string is a palindrome if it can be read the same way in either direction, from the left to the right and from the right to the left.

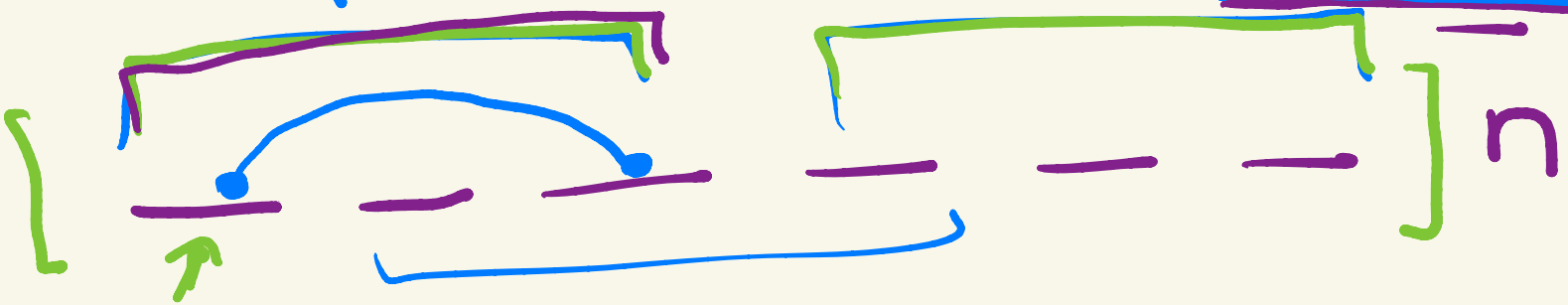
palindrome

Big giveaway

$k = 3$

n^2

$n, m, k \leq 2000$

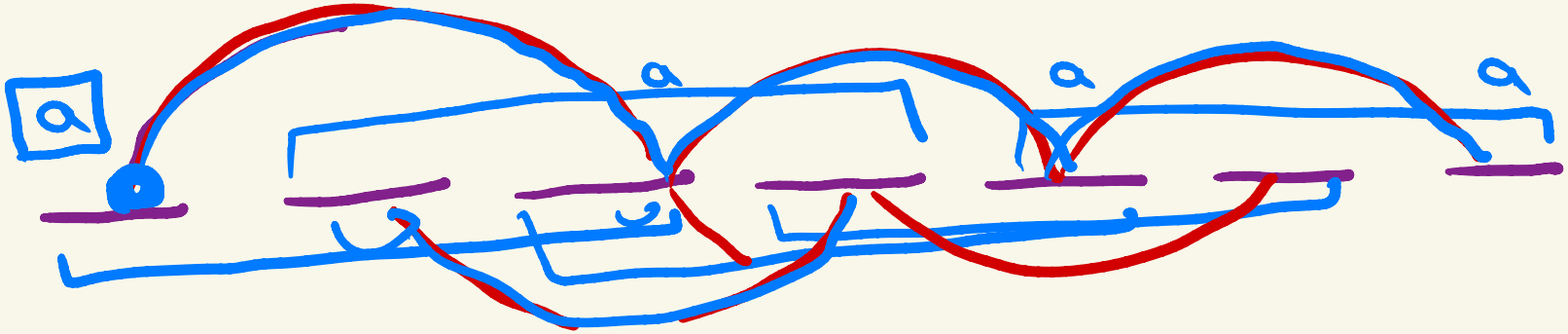


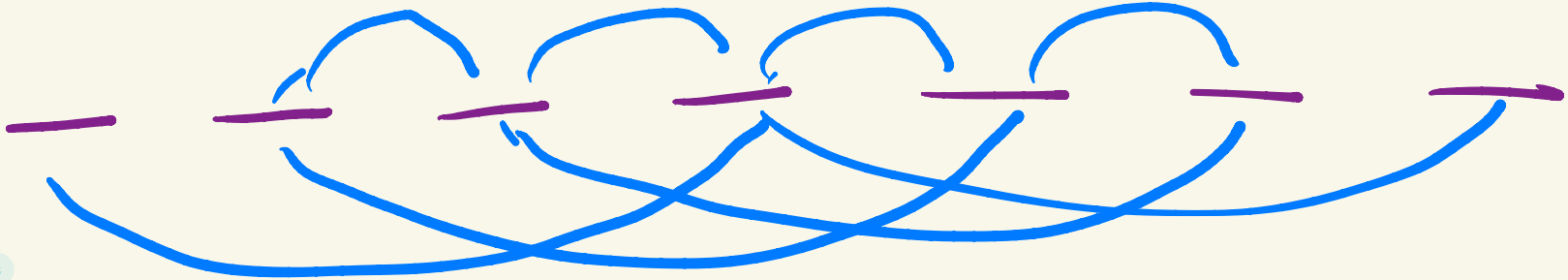
alphabet → 26

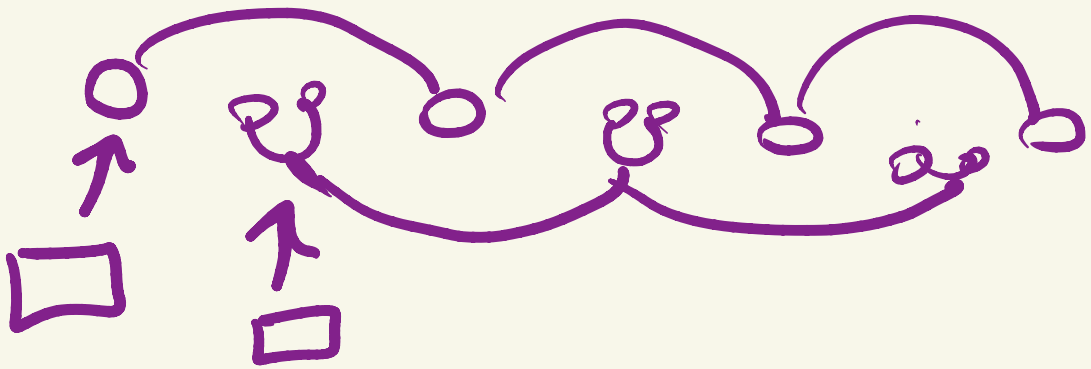
↳ m character

How many such strings exist?!

K = 3







[1-m] [1-m]

m choices

choices → number of connected comp.

$m^{\text{no. of CC}}$

Implementation

```
int n, m, k; cin >> n >> m >> k;

UnionFind DSU(n);

for(int i = 0; i <= n - k; i++) {
    int left = i, right = i + k - 1;
    while(left < right) {
        DSU.unionSet(left, right);
        left++, right--;
    }
}

int leaders = 0;

for(int i = 0; i < n; i++) leaders += (DSU.findSet(i) == i);

cout << power(m, leaders) << "\n";
```

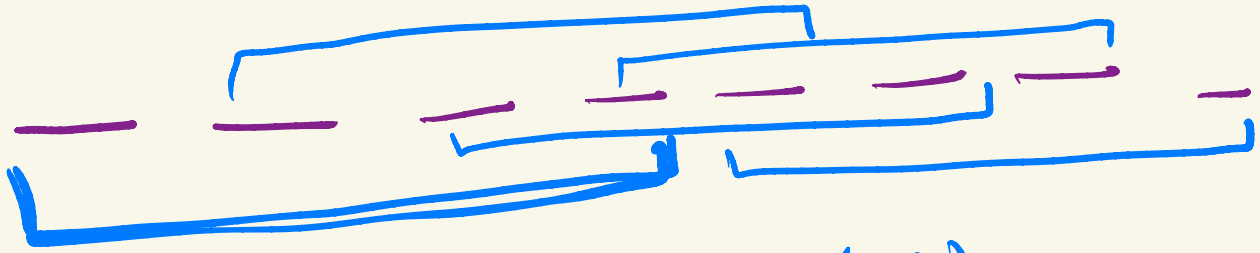
O(n²)

P(i) = P

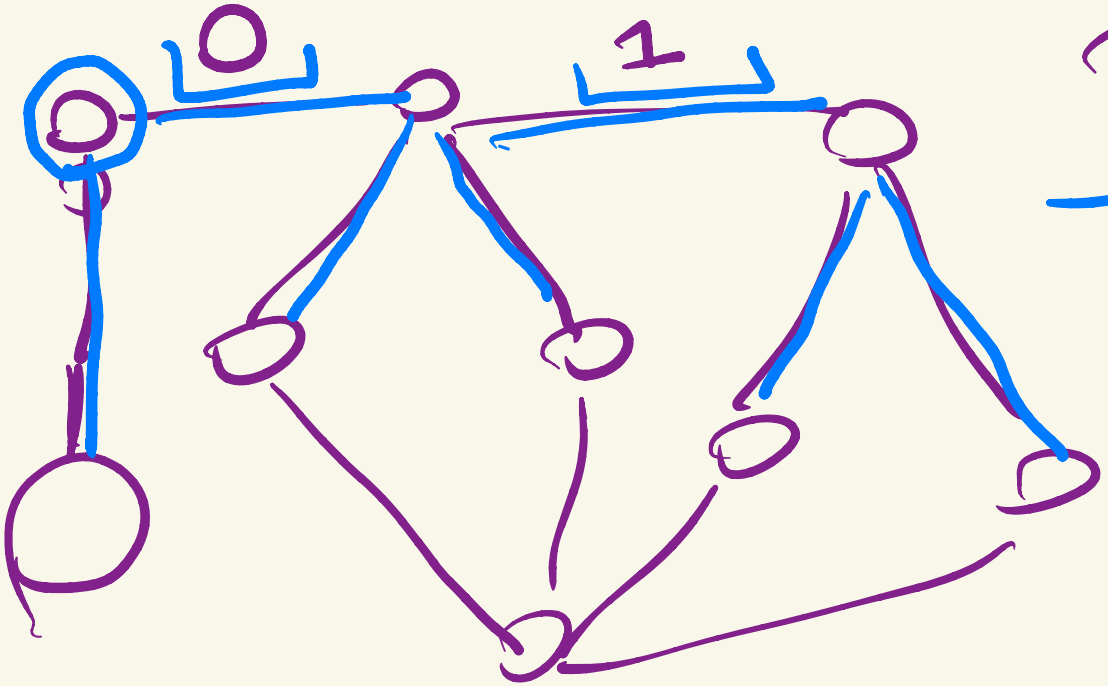
DFS

$k = n/2$

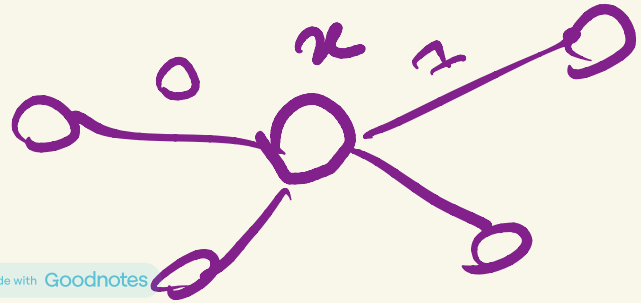
$n/2$ subst → $n/2$



$O(n \times n) \rightarrow O(n^2)$

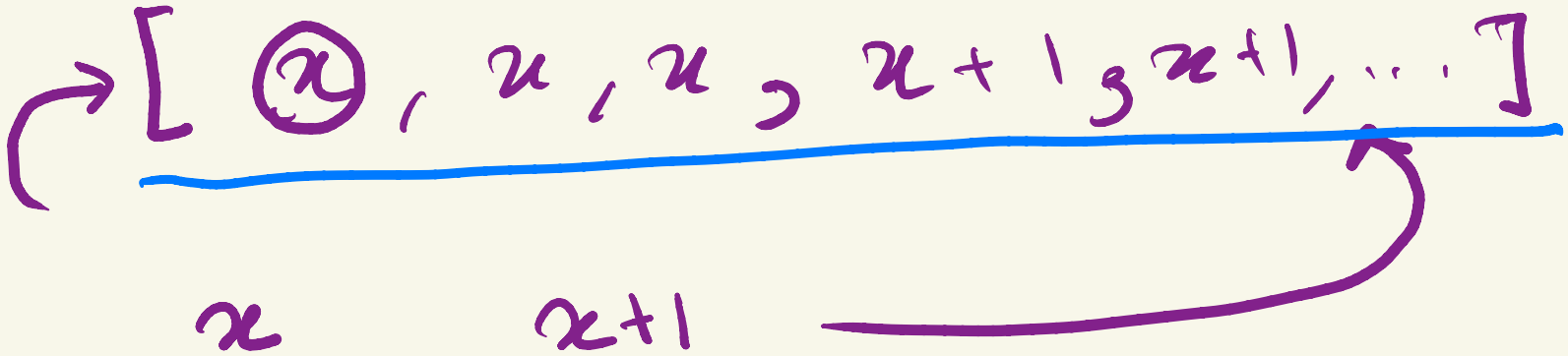


Dijkstra



x

$x+1$



$O(n+m)$