Disjoint Set Union

Problem

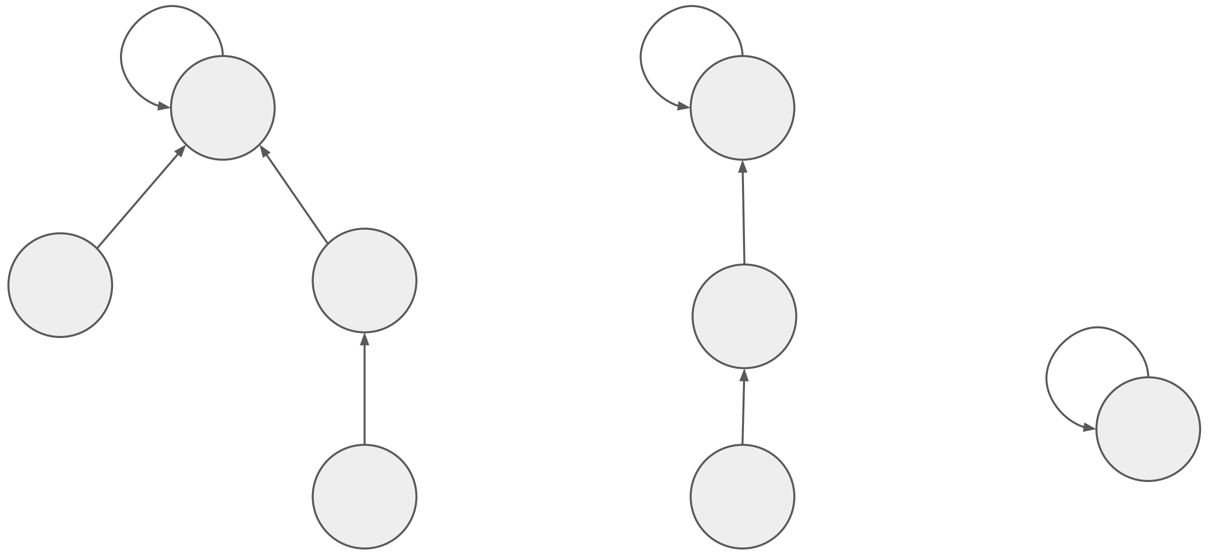
Maintain a partitioning of nodes into disjoint (non-overlapping) sets.

N nodes and support M operations, which can each be:

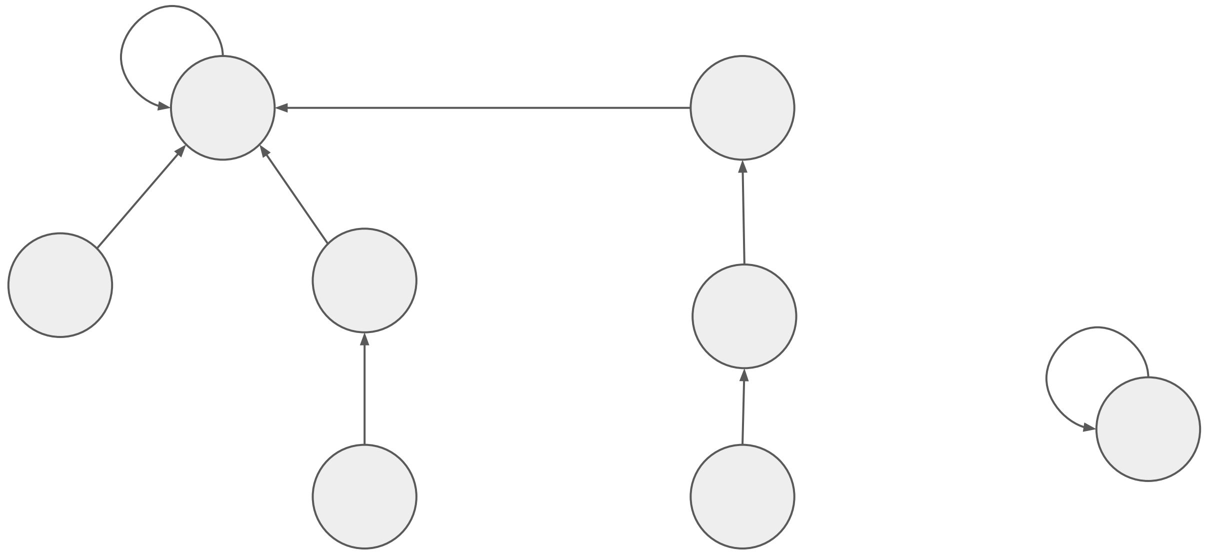
merge(a, b): merge sets of nodes a and b

find(a): find the representative element of the set that a is in

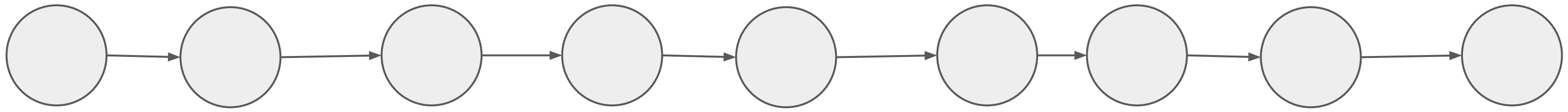
Basic approach



Basic approach



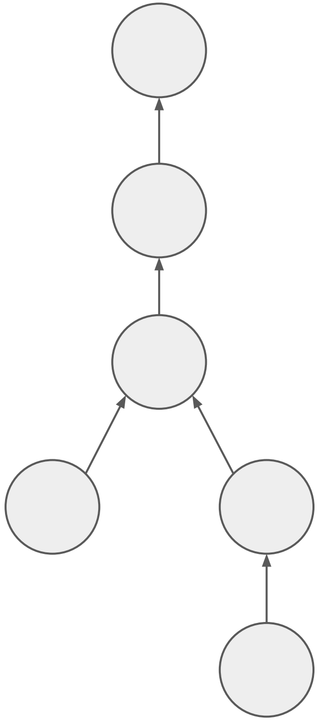
Worst case complexity



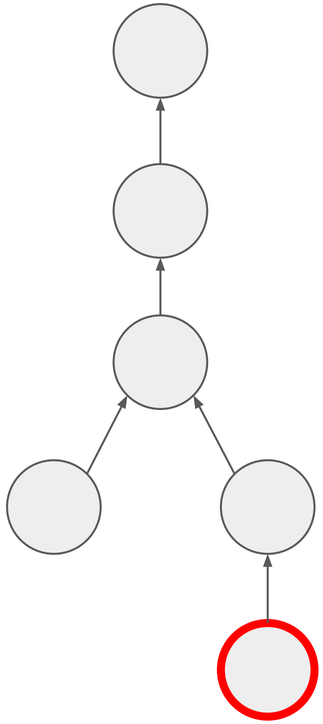
Optimizations

Path compression
Merge by size
Merge by rank (height)

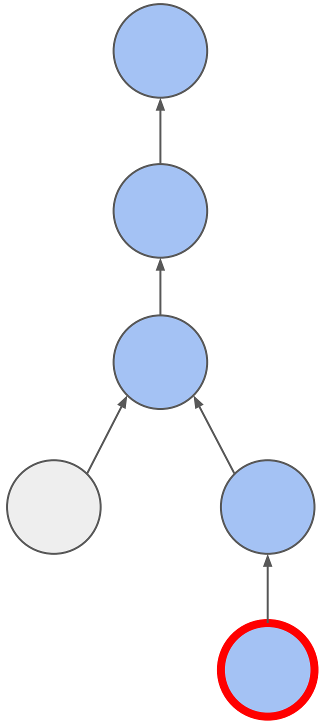
Path Compression



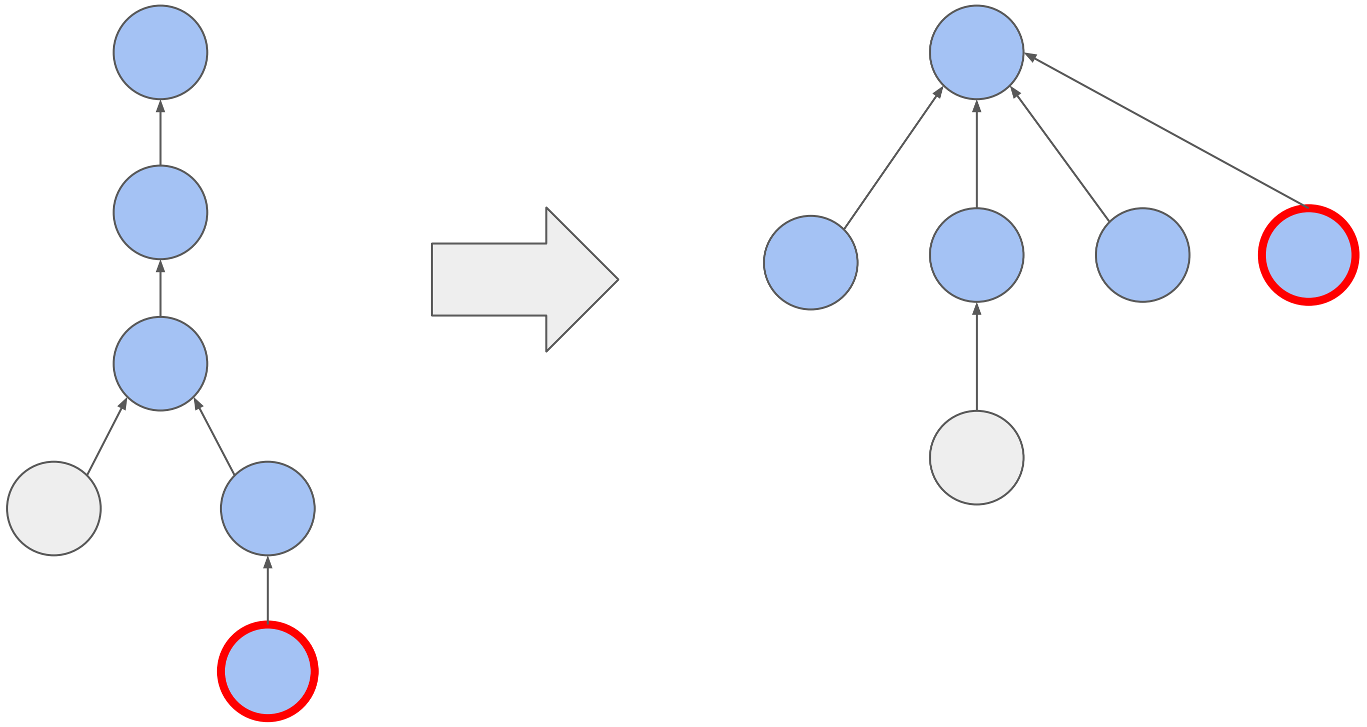
Path Compression



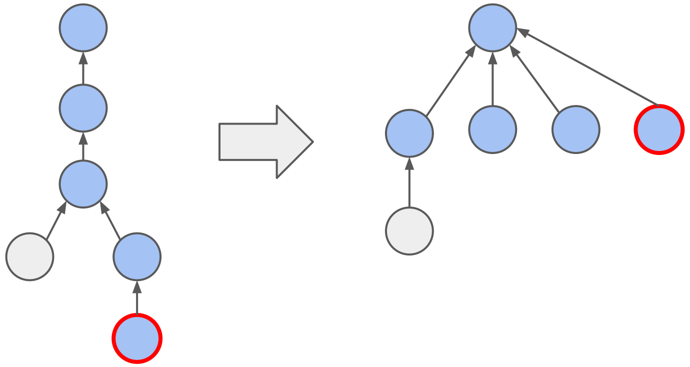
Path Compression



Path Compression



Path Compression



Complexity for M operations:

O(MlogN)

Note that a single operation could still take O(N) time.

Proof:

Analysis too hard

If you want analysis, try to understand https://dl.acm.org/doi/pdf/10.1145/62.2160 (Tarjan, van Leeuwen). Theorem 4 is the relevant one.

Implementation

C++

```cpp
const int MN = 1e6 + 5;
int n;
int dsu[MN];

void init(){
    for(int i = 0; i < n; i++) dsu[i] = i;
}

int find(int a){
    if(dsu[a] == a) return a;
    dsu[a] = find(dsu[a]);
    return dsu[a];
}

void merge(int a, int b){
    a = find(a);
    b = find(b);
    if(a != b) dsu[b] = a;
}
```

Python

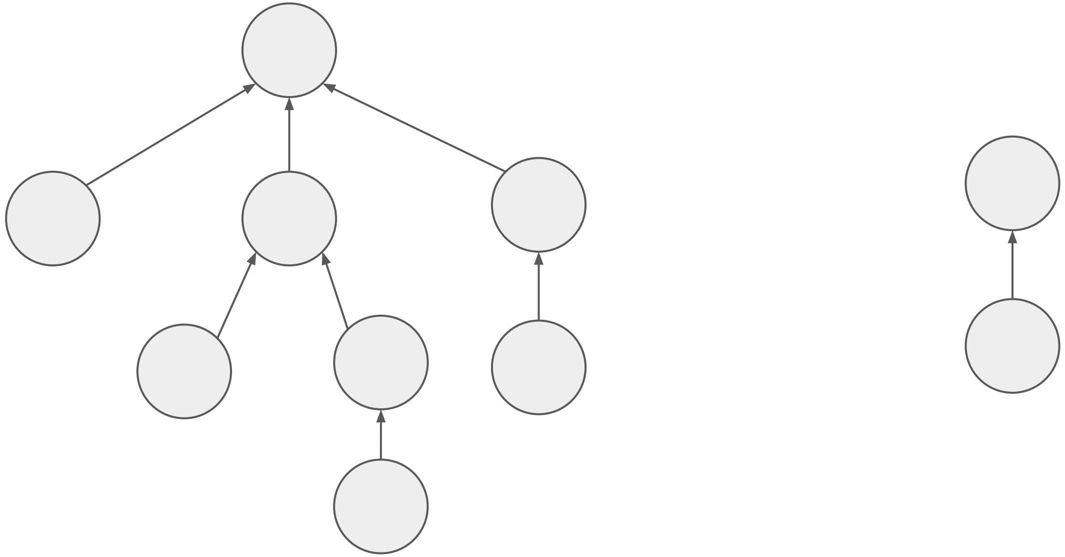
```python
dsu = [i for i in range(n)]

sys.setrecursionlimit(int(1e6)+5)
def find(a):
    if dsu[a] == a: return a
    dsu[a] = find(dsu[a])
    return dsu[a]

def merge(a, b):
    a = find(a)
    b = find(b)
    if a != b: dsu[a] = b
```

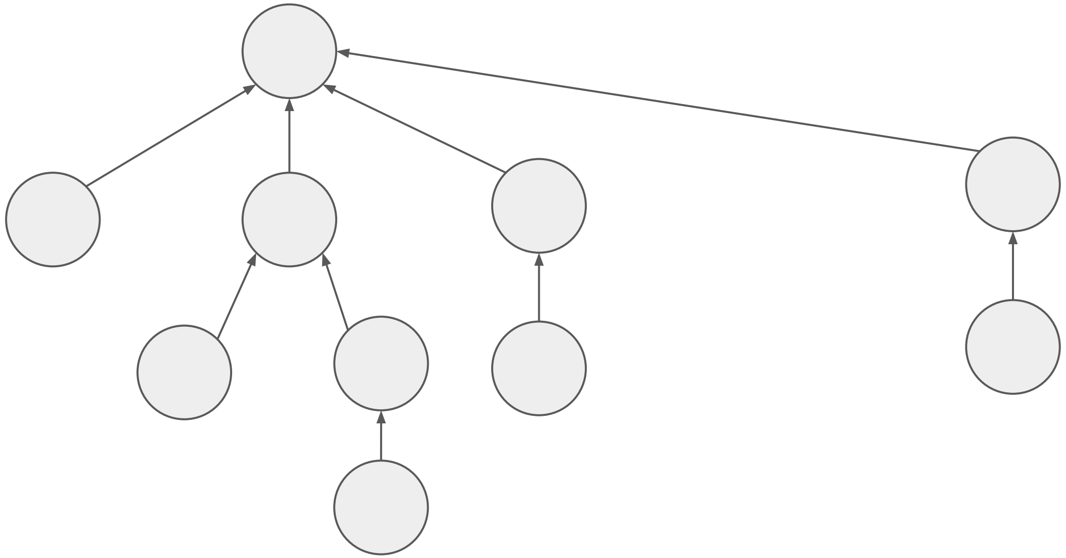
Merge by size

For each component, track number of nodes in it.
When merging, you have 2 options, choose option that merges small -> large.



Merge by size

For each component, track number of nodes in it.
When merging, you have 2 options, choose option that merges small -> large.



Merge by size

For each component, track number of nodes in it.
When merging, you have 2 options, choose option that merges small -> large.

Complexity:
O(logN) per find operation
O(MlogN) for M operations

Proof:
Height after merge is always max(height of large, height of small + 1)
Height only increases when height of large and height of small are the same
Idea: height only increases when the size roughly doubles. Therefore, height is at most logN
To prove this, assume any tree of height h has at least 2^{h-1} nodes, then use induction

Implementation

C++

```cpp
const int MN = 1e6 + 5;
int n;
int dsu[MN], sz[MN];

void init(){
    for(int i = 0; i < n; i++) dsu[i] = i;
    for(int i = 0; i < n; i++) sz[i] = 1;
}

int find(int a){
    while(dsu[a] != a) a = dsu[a];
    return a;
}

void merge(int a, int b){
    a = find(a);
    b = find(b);
    if(a == b) return;
    if(sz[a] < sz[b]) swap(a, b);
    dsu[b] = a;
    sz[a] += sz[b];
}
```

Python

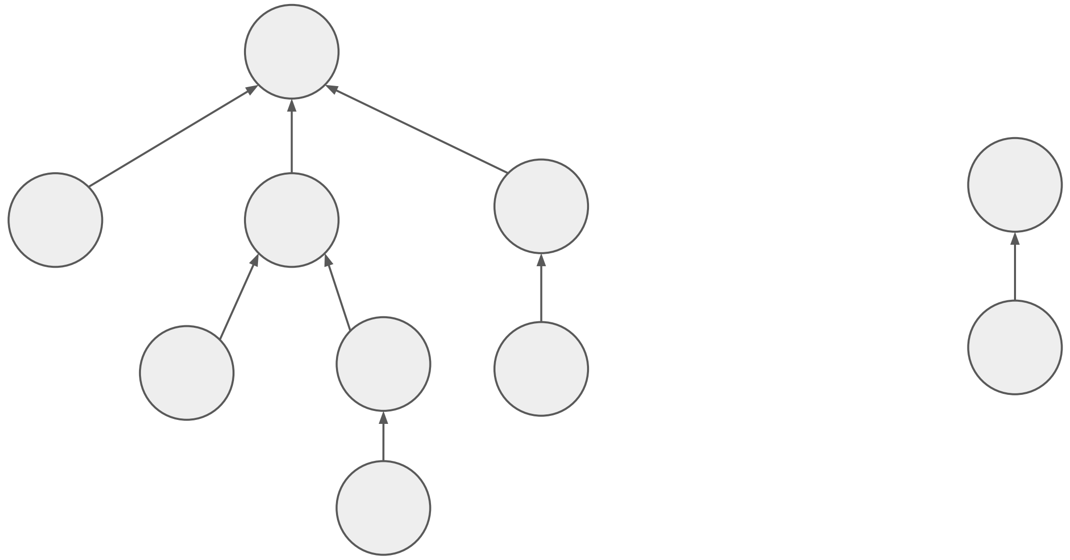
```python
dsu = [i for i in range(n)]
sz = [1]*n

def find(a):
    while dsu[a] != a: a = dsu[a]
    return dsu[a]

def merge(a, b):
    a = find(a)
    b = find(b)
    if a == b: return
    if sz[a] < sz[b]: a, b = b, a
    dsu[b] = a
    sz[a] += sz[b]
```

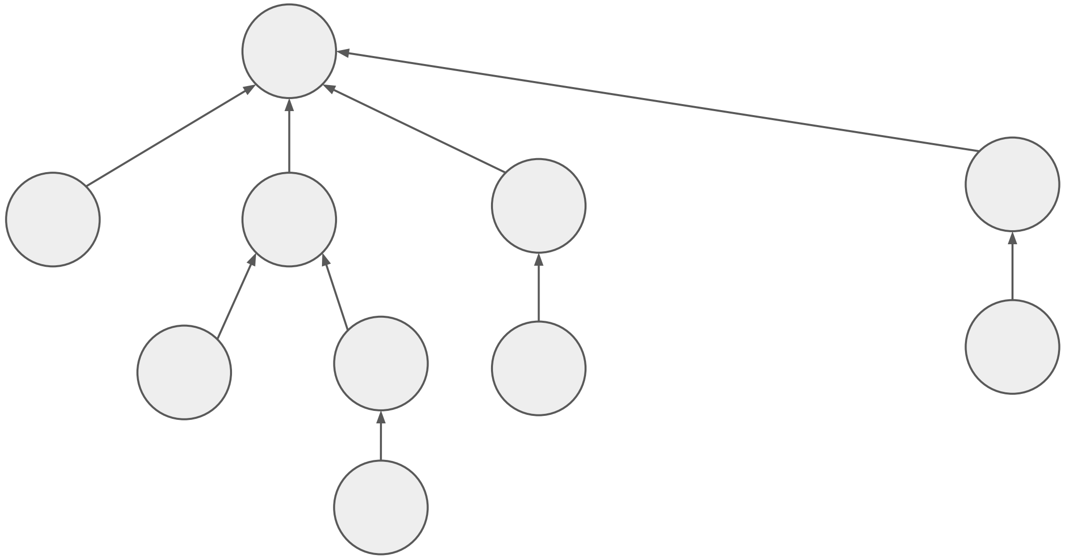
Merge by rank

Same as merge by size but instead of keeping track of size, we track height of each tree.



Merge by rank

Same as merge by size but instead of keeping track of size, we track height of each tree.



Merge by rank

Same as merge by size but instead of keeping track of size, we track height of each tree.

Complexity:
O(logN) per find operation
O(MlogN) for M operations

Proof:
Assume tree with height h contains at least 2^{h-1} nodes
Obviously true for h = 1
For all other h, merging two trees of different heights sets height to max of them, doesn’t increase
Height only increases when two trees of same height are merged
Increasing height by 1 means number of nodes at least doubled
Tree with N nodes has height at most $\log_2(N)+1$

Implementation

```
const int MN = 1e6 + 5;
int n;
int dsu[MN], height[MN];

void init(){
    for(int i = 0; i < n; i++) dsu[i] = i;
}

int find(int a){
    while(dsu[a] != a) a = dsu[a];
    return a;
}

void merge(int a, int b){
    a = find(a);
    b = find(b);
    if(a == b) return;
    if(height[a] < height[b]) swap(a, b);
    dsu[b] = a;
    height[a] = max(height[a], height[b]+1);
}
```

Assume tree with 1 node is height 0 and not 1 because lazy and no difference.

Advantage of union by size vs union by rank

¯\_(ツ)_/¯

idk. none. Maybe size is better if question asks for size of components since then you have to store size anyways.

Path compression + merge by size/rank

Why not combine both optimizations?

Complexity for M operations is $O(M\alpha(N))$

α is inverse Ackermann function

Normal Ackermann function:

A(0, 0) = 1

A(1, 1) = 3

A(2, 2) = 7

A(3, 3) = 61

A(4, 4) = 2^2^2^65536 - 3

Proof of complexity:

Analysis even harder than just path compression

Again, read paper if you want (Theorem 3 in the same paper)

Implementation

C++

```cpp
const int MN = 1e6 + 5;
int n;
int dsu[MN], sz[MN];

void init(){
    for(int i = 0; i < n; i++) dsu[i] = i;
    for(int i = 0; i < n; i++) sz[i] = 1;
}

int find(int a){
    if(dsu[a] == a) return a;
    dsu[a] = find(dsu[a]);
    return dsu[a];
}

void merge(int a, int b){
    a = find(a);
    b = find(b);
    if(a == b) return;
    if(sz[a] < sz[b]) swap(a, b);
    dsu[b] = a;
    sz[a] += sz[b];
}
```

Python

```python
dsu = [i for i in range(n)]
sz = [1]*n

def find(a):
    if dsu[a] == a: return a
    dsu[a] = find(dsu[a])
    return dsu[a]

def merge(a, b):
    a = find(a)
    b = find(b)
    if a == b: return
    if sz[a] < sz[b]: a, b = b, a
    dsu[b] = a
    sz[a] += sz[b]
```

Implementation

C++

```cpp
const int MN = 1e6 + 5;
int n;
int dsu[MN], sz[MN];

void init(){
    for(int i = 0; i < n; i++) dsu[i] = i;
    for(int i = 0; i < n; i++) sz[i] = 1;
}

int find(int a){
    return dsu[a] == a ? a : dsu[a] = find(dsu[a]);
}

void merge(int a, int b){
    a = find(a);
    b = find(b);
    if(a == b) return;
    if(sz[a] < sz[b]) swap(a, b);
    dsu[b] = a;
    sz[a] += sz[b];
}
```

Python

```python
dsu = [i for i in range(n)]
sz = [1]*n

def find(a):
    if dsu[a] == a: return a
    dsu[a] = find(dsu[a])
    return dsu[a]

def merge(a, b):
    a = find(a)
    b = find(b)
    if a == b: return
    if sz[a] < sz[b]: a, b = b, a
    dsu[b] = a
    sz[a] += sz[b]
```

Another trick for simplifying implementation

Size only needs to be stored at root

Root doesn’t have to store a parent pointer anyways

Store size and parent pointers in the same array

Make size negative to differentiate between root and non-root

Implementation

C++

```cpp
const int MN = 1e6 + 5;
int n;
int dsu[MN];

void init(){
    fill(dsu, dsu + n, -1);
}

int find(int a){
    return dsu[a] < 0 ? a : dsu[a] = find(dsu[a]);
}

void merge(int a, int b){
    a = find(a);
    b = find(b);
    if(a == b) return;
    if(dsu[a] > dsu[b]) swap(a, b);
    dsu[a] += dsu[b];
    dsu[b] = a;
}
```

Python

```python
dsu = [-1]*n

def find(a):
    if dsu[a] < 0: return a
    dsu[a] = find(dsu[a])
    return dsu[a]

def merge(a, b):
    a = find(a)
    b = find(b)
    if a == b: return
    if dsu[a] > dsu[b]: a, b = b, a
    dsu[a] += dsu[b]
    dsu[b] = a
```

Recommendations for implementation

C++/Java: use path compression or path compression + merge by size
Python: use merge by size or merge by size + path compression

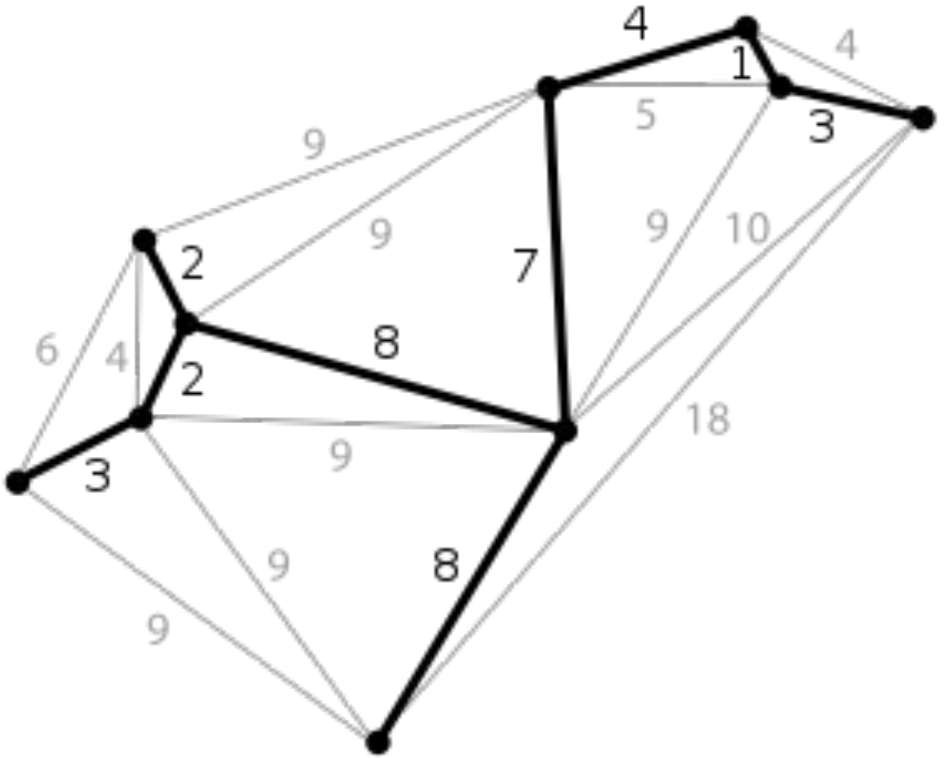
Python recursion bad so path compression only is not the best

DSU problems

https://dmoj.ca/problem/dmpg17s2 (template DSU)
https://dmoj.ca/problem/ds2 (template DSU)
https://dmoj.ca/problem/examprep (simple DSU but must maintain size)
https://dmoj.ca/problem/ccc15s3 (greedy then DSU)
https://dmoj.ca/problem/mcco19c2d1p1 (one observation then DSU)
https://cses.fi/problemset/task/1676 (DSU and maintain size)
https://dmoj.ca/problem/noi02p1 (DSU while storing extra values in parent pointer)

Minimum Spanning Tree

Definition



Kruskal’s Algorithm

Maintain a forest that’s a subgraph of the original graph and contains all nodes. The edges in this forest will be part of the final spanning tree.
Sort the edges in increasing order of weight.
For each edge, if the nodes it connects are not yet connected, add the edge to the forest. Otherwise do nothing.

Proof of correctness

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm#Proof_of_correctness