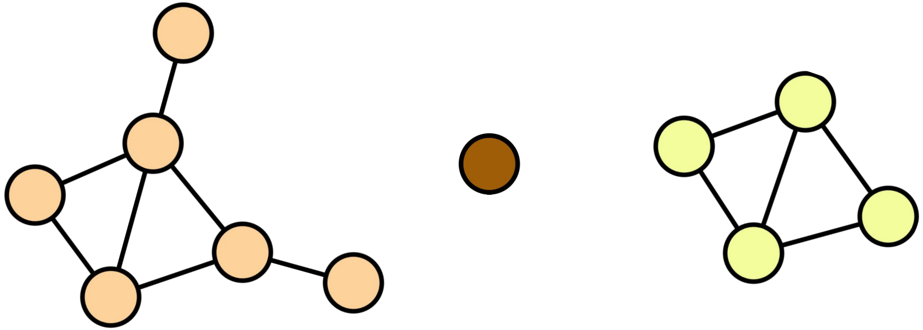
Union Find
(Disjoint Set Union)



Africa To Silicon Valley

Objectives

- Understand Union Find basics
- Learn union and find operations
- Understand path compression
- Apply in problem-solving
- Analyze time and space complexities

Lecture Flow

1) Pre-requisites
2) Problem definition
3) Brute force approach
4) Improved brute force approach
5) Union Find
6) Union Find optimization
7) Solve problem
8) Things to pay attention
9) Practice problems
10) Quote of the day

Pre-requisites

- Graph
- Connected components
- Cycles in graphs
- Set (intersection, union)

A2SV
Africa To Silicon Valley

Problem Definition

You have a graph of cities and roads connecting the cities.

You are given multiple queries and in one query, you are asked to determine if two cities are connected.

How would you approach this?

We need a quick way to determine if two nodes are connected.

Brute Force Approach

- Build a graph

- For every query,
  - make one node a source and
  - the other a destination.
  - do graph traversal to check if path exists

- Time complexity: O(queries * (V + E))

Can we do better?

Improved Brute Force Approach

- What if we group connected cities together?
- For quicker look up we use Set to group
- We use dictionary to track which group each city belongs to
    - City : Group

Africa To Silicon Valley

Improved Brute Force Approach (cont…)

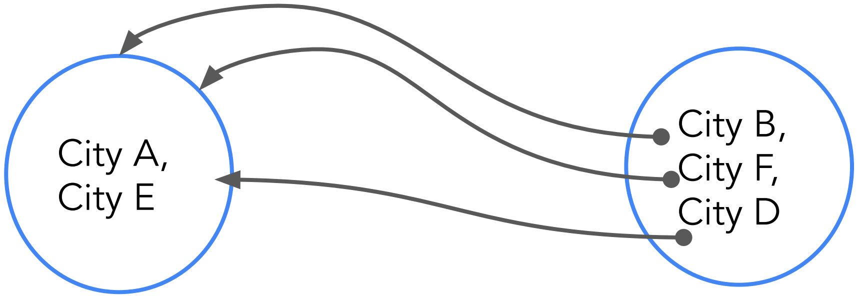
- To check whether there is a connection between City A and City B

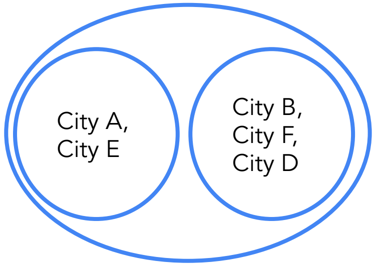
```
return dictionary[city_a] is dictionary[city_b]
```

- If both have the same Set reference, then they must belong to the same group

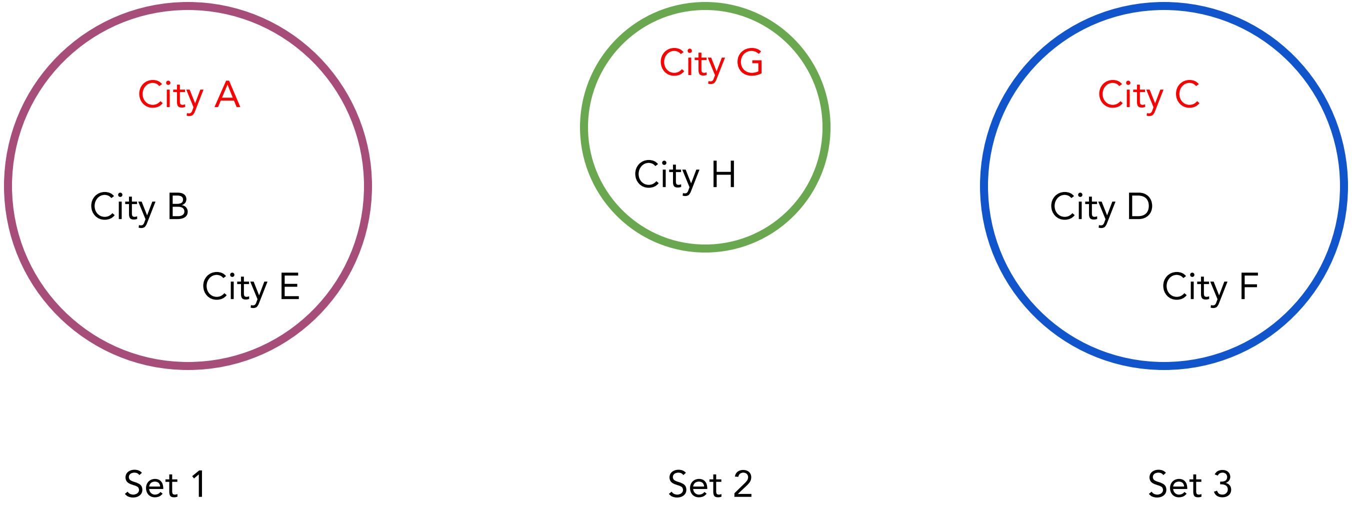
Improved Brute Force Approach (cont…)

- For merging City A and City B, we'd need to change the dictionary reference for every city merged with one of the cities which adds extra time complexity.





Improved Brute Force Approach (cont...)




```
Dictionary:

City A: Set_1
City B: Set_1
City E: Set_1
City G: Set_2
City H: Set_2
City C: Set_3
City D: Set_3
City F: Set_3
```

A2SV
Africa To Silicon Valley

Improved Brute Force Approach (cont…)

- We don't need the entire set to check if two cities belong to the same connected component; we are only checking reference.

- In that case, instead of using sets as an identifier of a group, we can select a representative element for each set.

```
Dictionary:

City A: Set_1
City B: Set_1
City E: Set_1
City G: Set_2
City H: Set_2
City C: Set_3
City D: Set_3
City F: Set_3
```

A2SV

Africa To Silicon Valley

Improved Brute Force Approach (cont…)

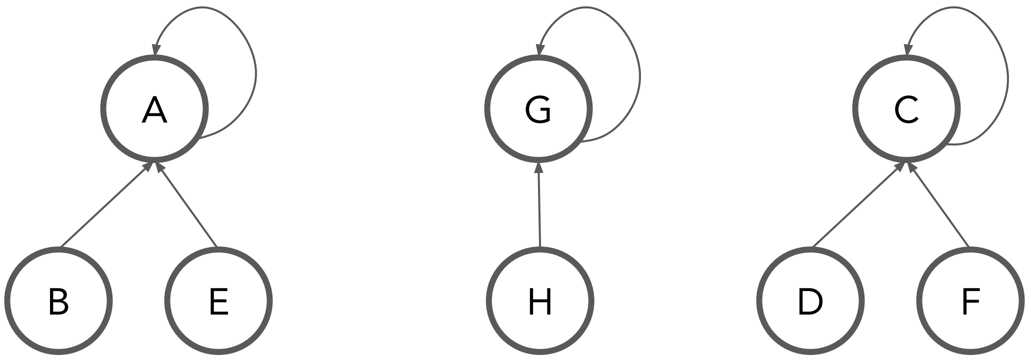
City A belongs to Set_1
City B belongs to Set_1

City A represented by City A
City B represented by City A

```
Dictionary:

City A: Set_1
City B: Set_1
City E: Set_1
City G: Set_2
City H: Set_2
City C: Set_3
City D: Set_3
City F: Set_3
```

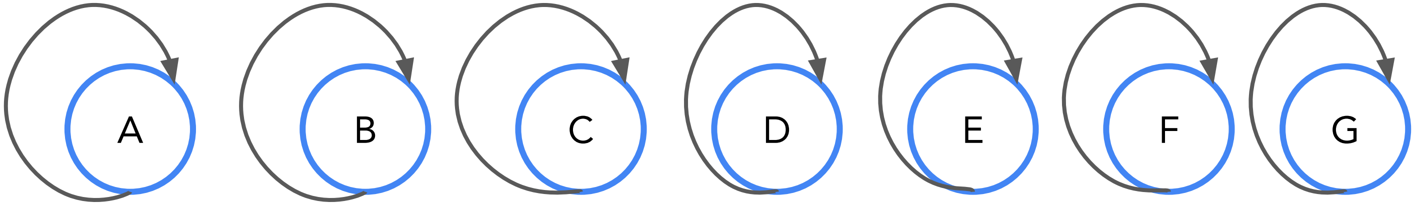
```
Dictionary:

City A: City A
City B: City A
City E: City A
City G: City G
City H: City G
City C: City C
City D: City C
City F: City C
```

What does it mean to connect two cities?



Arrows pointing to their representative

What does it mean to connect two cities?

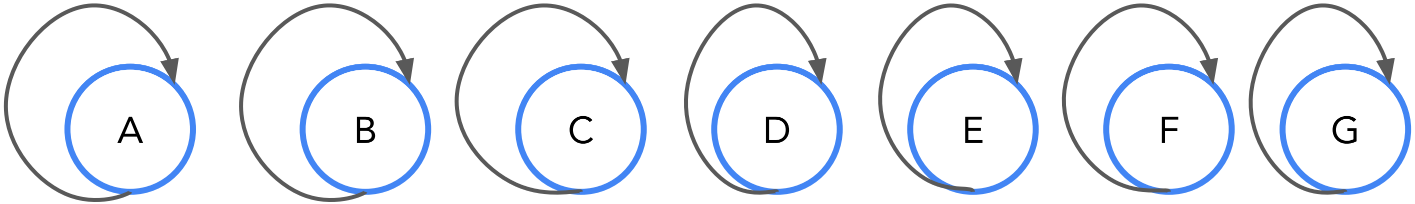


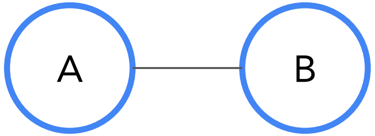
First, all nodes represent themselves

Dictionary:

```
City A: City A
City B: City B
City E: City E
City G: City G
City H: City H
City C: City C
City D: City D
City F: City F
```

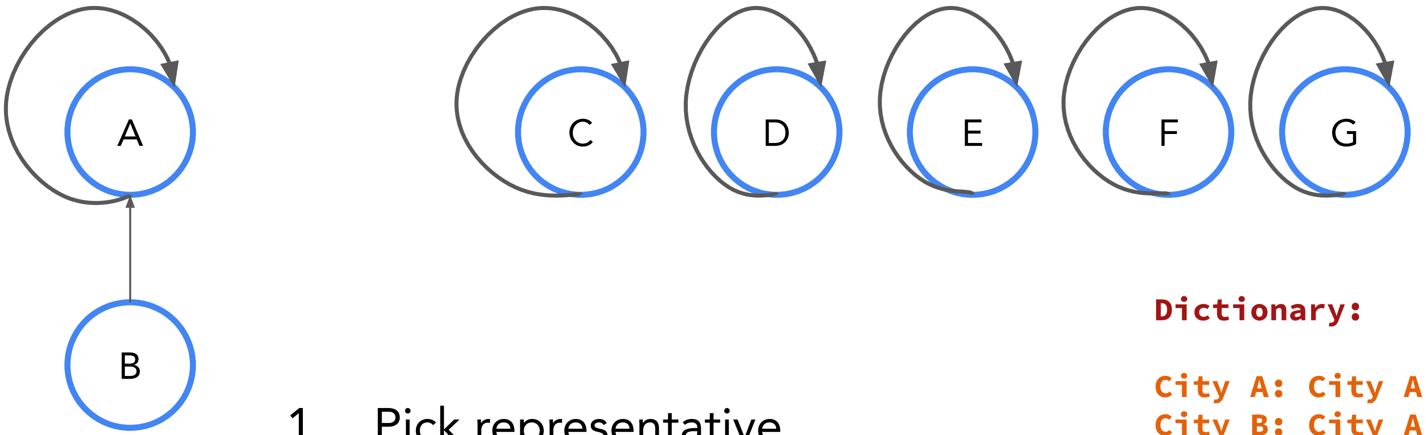
What does it mean to connect two cities?





Connect Node A with Node B?

What does it mean to connect two cities?

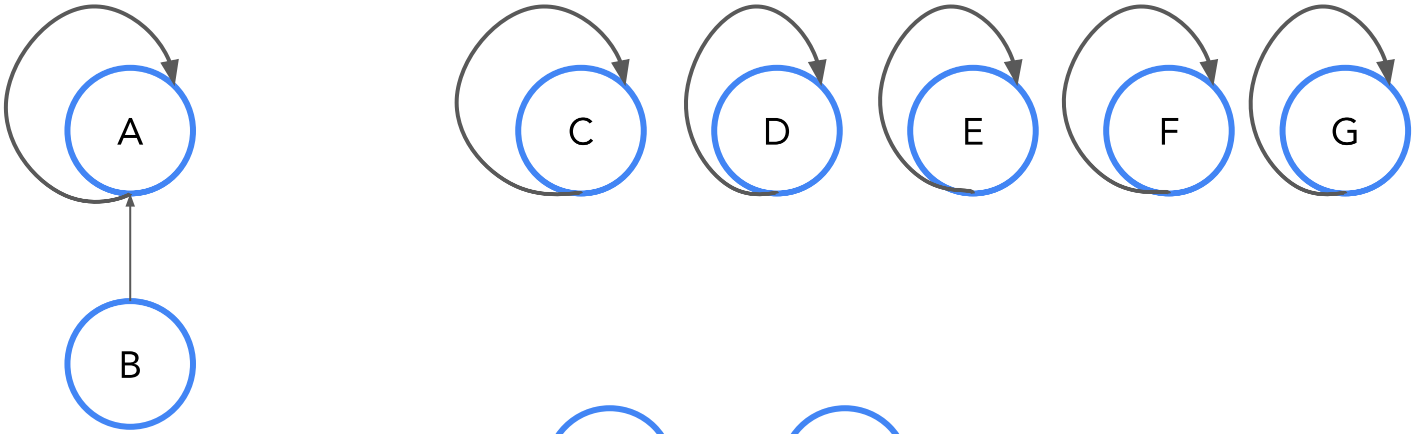


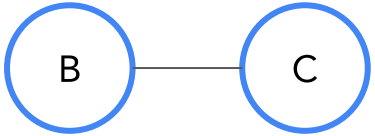
1. Pick representative
2. Then, assign both of them the same representative

Dictionary:

```
City A: City A
City B: City A
City E: City E
City G: City G
City H: City H
City C: City C
City D: City D
City F: City F
```

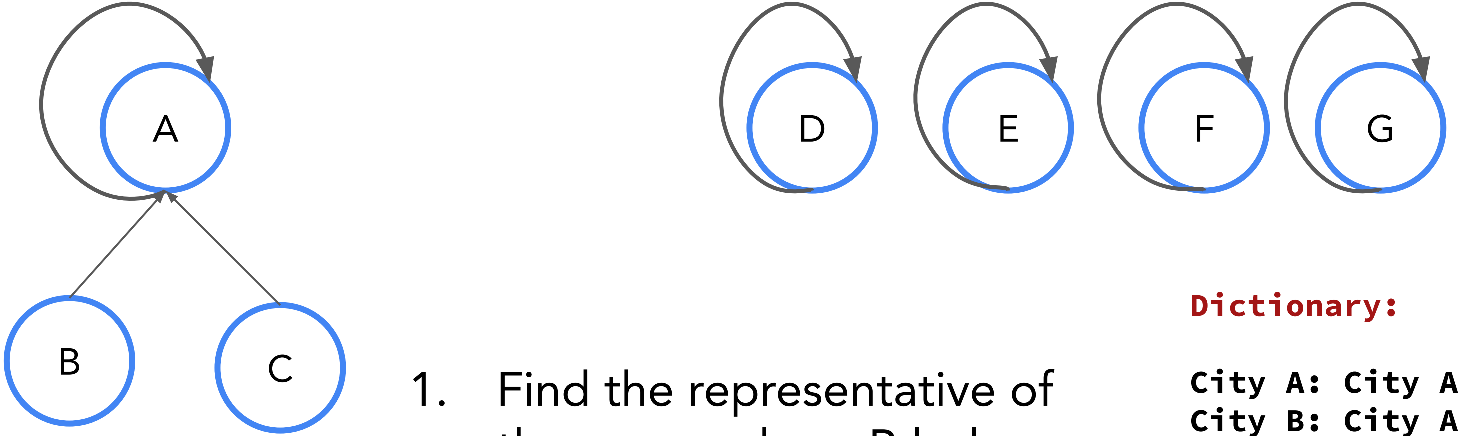
What does it mean to connect two cities?





Connect Node C with Node B?

What does it mean to connect two cities?



1. Find the representative of the group where B belongs
2. Assign C's representative to B's representative

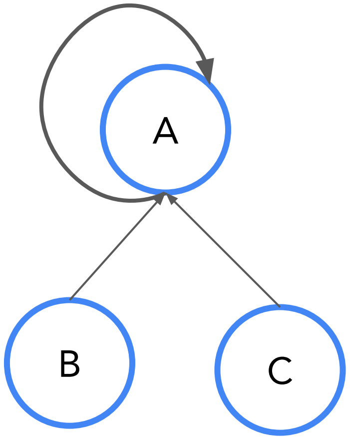
Dictionary:

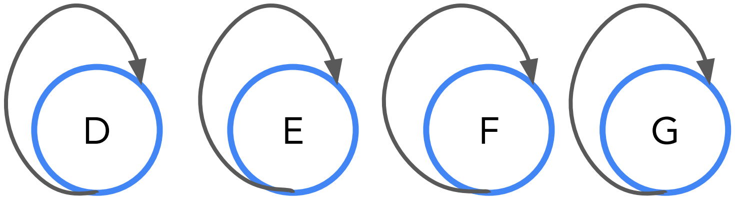
```
City A: City A
City B: City A
City E: City E
City G: City G
City H: City H
City C: City A
City D: City D
City F: City F
```

In summary,

assign the same representative to both

How do we check if they belong to same group?





Check if they have the same representative

```
dictionary[city_a] == dictionary[city_b]
```

Implement Here

Implementation

```python
# Union Find class
class UnionFind:
    def __init__(self, size):
        self.parent = {i: i in range(size)}

    def find(self, x):
        return self.parent[x]

    def union(self, x, y):
        parentX = self.find(x)
        parentY = self.find(y)

        if parentX != parentY:
            for node in self.parent:
                if self.parent[node] == parentX:
                    self.parent[node] = parentY

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

For all nodes in the same group as node x, update their parent pointers to point to the parent of node y.

A2SV
Africa To Silicon Valley

Time & Space Complexity Analysis

Time complexity:

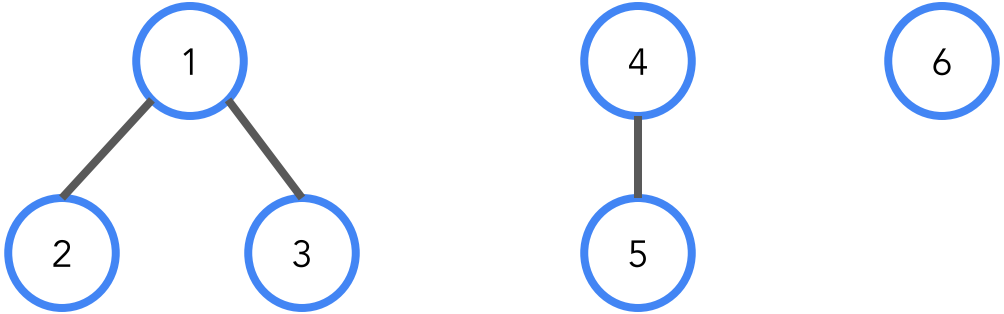
- Union: O(V)
- Find: O(1)
- Connected: O(1)

Space Complexity: O(V)

Topic Introduction

Union-find is a way to group objects and efficiently determine if two objects belong in the same group, as well as join two groups.

It is also known as Disjoint Set and Merge Find Set.



Problems solved using Union-Find

- Check if path between two nodes exists in undirected graph in < O(N) time
- Count connected components in undirected graph
- Detecting cycles in an undirected graph
- Merging sets efficiently
- Kruskal's Minimum Spanning Tree algorithm

Union Find implementation can be optimized.

Let's brainstorm

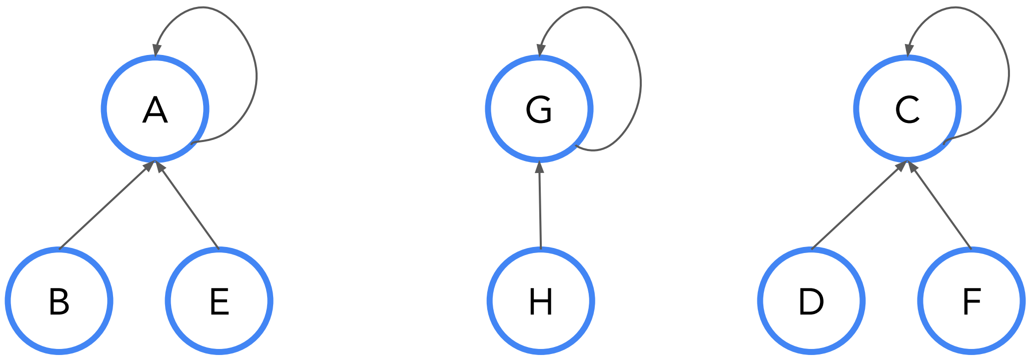
The previous implementation is called Quick Find

```python
# Union Find class
class UnionFind:
    def __init__(self, size):
        self.parent = {i: i in range(size)}

    def find(self, x):
        return self.parent[x]

    def union(self, x, y):
        parentX = self.find(x)
        parentY = self.find(y)

        if parentX != parentY:
            for node in self.parent:
                if self.parent[node] == parentX:
                    self.parent[node] = parentY

    def connected(self, x, y):
        return self.find(x) == self.find(y)
```

$O(\textbf{find}) = O(1)$

$O(\textbf{union}) = O(V)$

Quick Find: Make all nodes point to their representative



Dictionary:

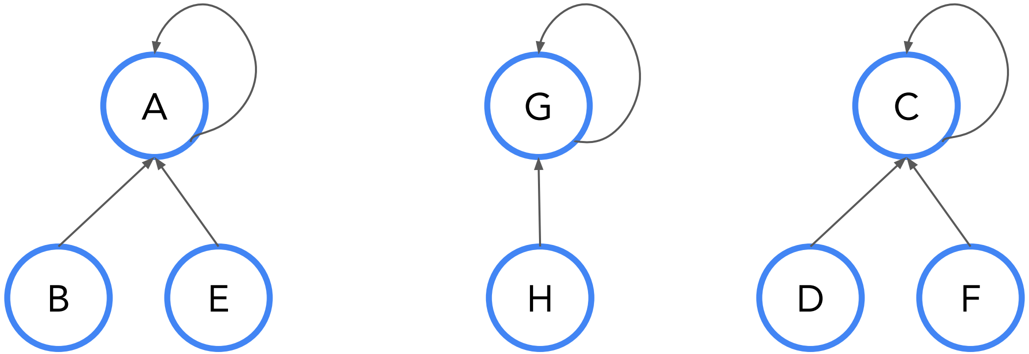
```
City A: City A
City B: City A
City E: City A
City G: City G
City H: City G
City C: City C
City D: City C
City F: City C
```

Note: What happens during UNION for Quick Find?

Let's explore Quick Union

Quick Union Steps

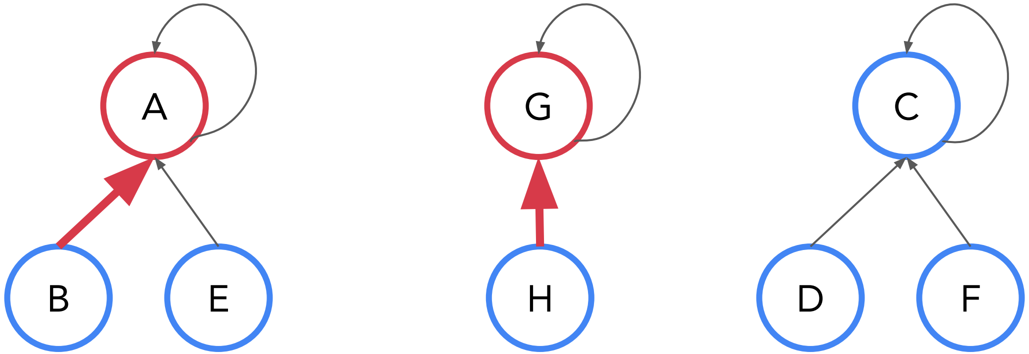
Union H <> B



Quick Union Steps

Union H <> B

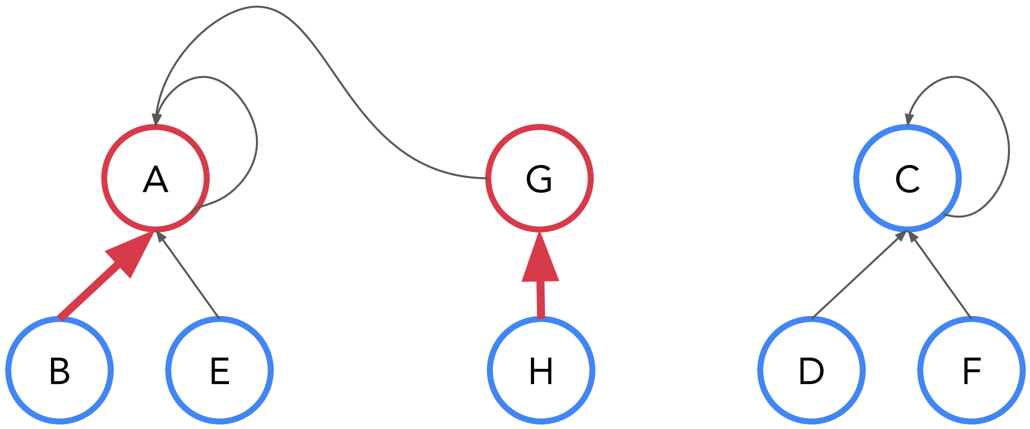
1. Find representative of H
2. Find representative of B
3. ?



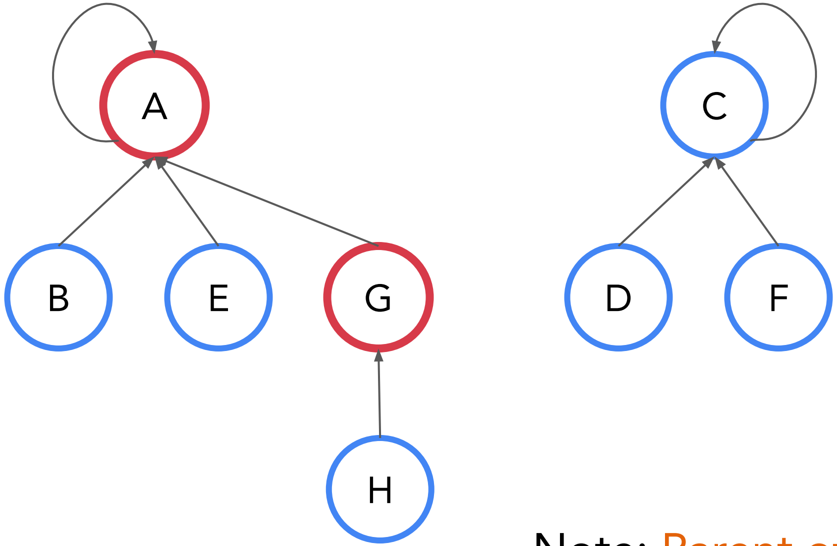
Quick Union Steps

Union H <> B

1. Find representative of H
2. Find representative of B
3. Make G's representative B's Representative



Quick Union H <> B



Dictionary:

```
City A: City A
City B: City A
City E: City A
City G: City A
City H: City G
City C: City C
City D: City C
City F: City C
```

Note: Parent of H is not updated immediately as in Quick Find.

Implement Quick Union Here

A2SV
Africa To Silicon Valley

Quick Union

```python
class UnionFind:
    def __init__(self, size):
        self.parent = [i for i in range(size)]

    def find(self, city):
        if city == self.parent[city]:
             return city
        return self.find(self.parent[city])

    def union(self, city1, city2):
        root1 = self.find(city1)
        root2 = self.find(city2)
        if root1 != root2:
            self.parent[root2] = root1

    def connected(self, member1, member2):
        return self.find(member1) == self.find(member2)
```

Find method changed for Quick Union. Why?

- O(**find**) = **?**
- O(**union**) = **?**

Quick Union

```python
class UnionFind:
    def __init__(self, size):
        self.parent = [i for i in range(size)]

    def find(self, member):
        while member != self.parent[member]:
            member = self.parent[member]
        return member

    def union(self, member1, member2):
        root1 = self.find(member1)
        root2 = self.find(member2)
        if root1 != root2:
            self.parent[root2] = root1

    def connected(self, member1, member2):
        return self.find(member1) == self.find(member2)
```

Worst Case

- O(**find**) = **O(V)**
- O(**union**) = **O(V)**

Which one is faster,

Quick Union or Quick Find?

Quick Union is faster with small few tweaks

- Quick-union is generally considered better than quick-find because:
    - The union operation can be optimized using weighting or height
    - The find operation can be optimized using path compression

How can we improve Quick Union?

A2SV
Africa To Silicon Valley

When we merge two sets,
what relevant information should be considered
when deciding who to make a representative?

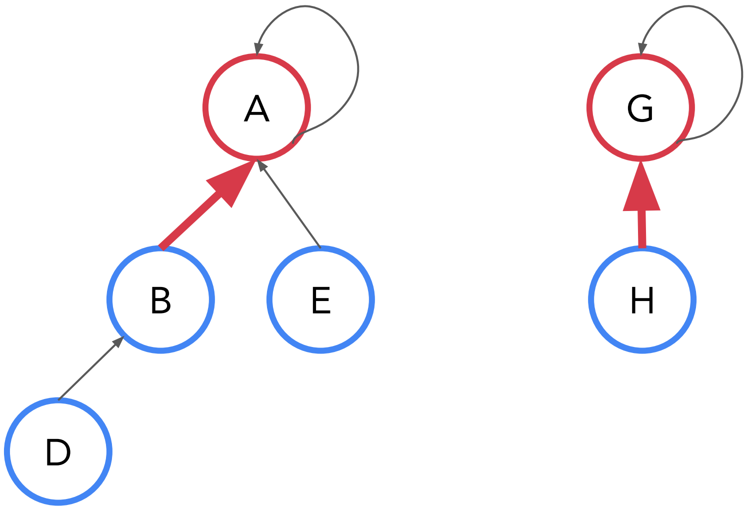
When merging

- Optimize by considering the size of the group
- Merging the smaller group into the larger one prevents an increase in tree depth
- A shallower tree improves the time complexity of the find operation

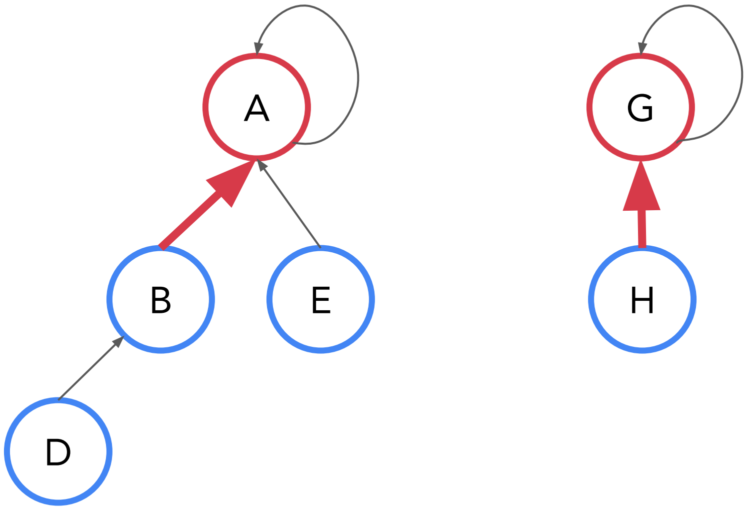
A2SV
Africa To Silicon Valley

Let's use the size

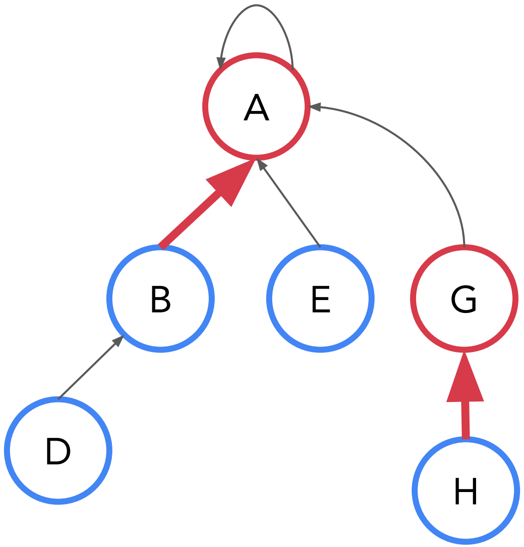
- Size is the number of members in the group



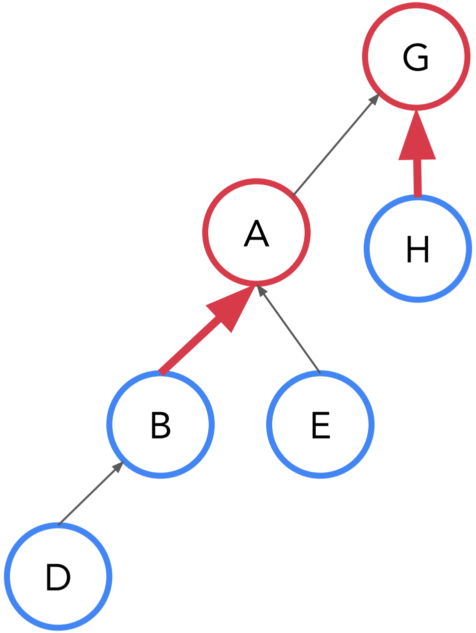
Union H <> B



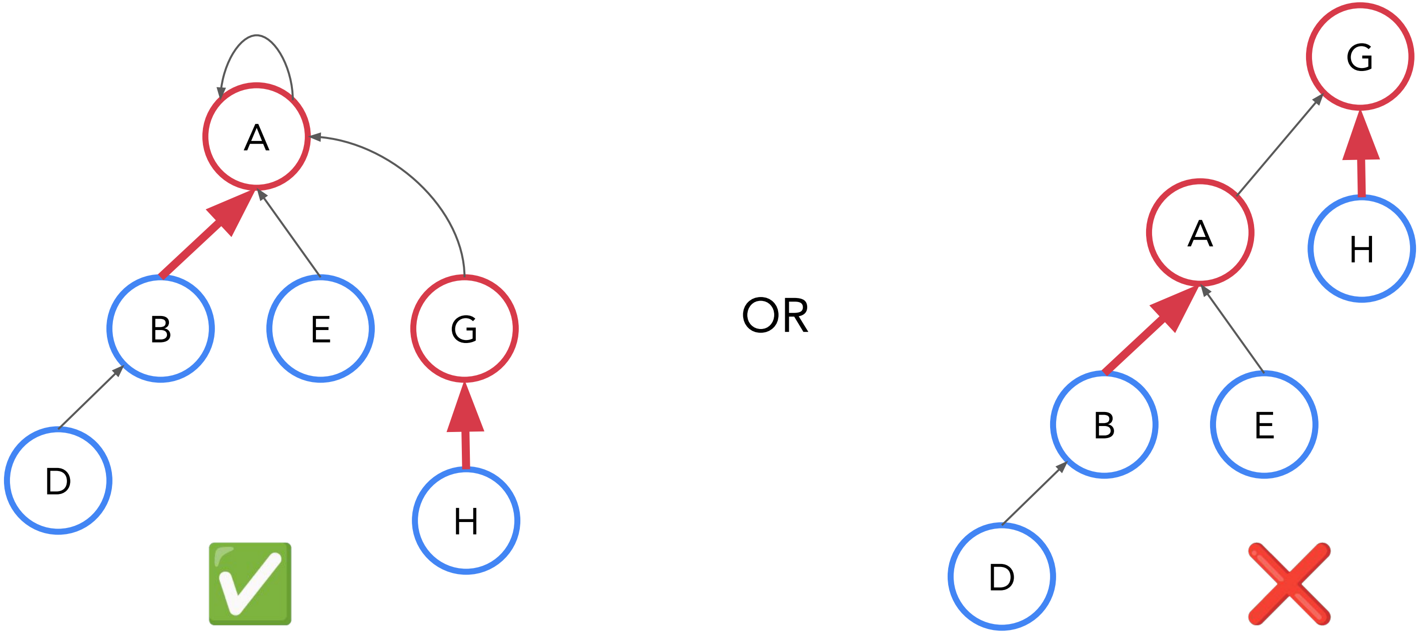
Union H <> B, which merging is better?



OR



Which one would take longer to find the parent for node 'D'?



Implement Union by Size Here

Union by Size

```python
class UnionFind:
    def __init__(self, size):
        self.root = [i for i in range(size)]
        self.size = [1] * size

    def find(self, x):
        while x != self.root[x]:
            x = self.root[x]
        return x

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.size[rootX] > self.size[rootY]:
                self.root[rootY] = rootX
                self.size[rootX] += self.size[rootY]
            else:
                self.root[rootX] = rootY
                self.size[rootY] += self.size[rootX]
```

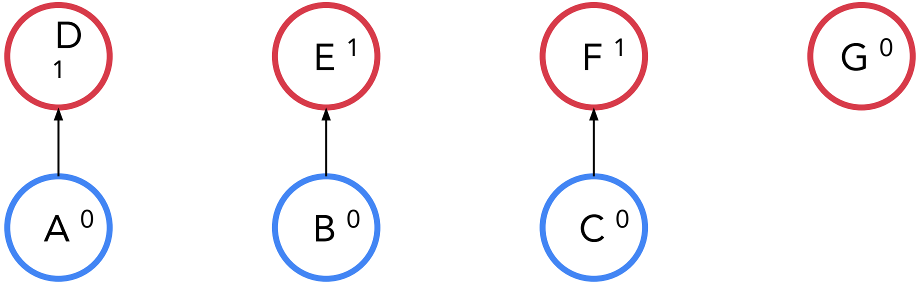
- O(**find**) = ?
- O(**union**) = ?
- O(**constructor**) = ?

Union by Rank

- Rank is an upper bound for its height.
- Initially, its rank is set to zero.
- When merging, first compare their ranks:
  - If the ranks are different, then the larger rank tree becomes the parent, and the ranks do not change.
  - If the ranks are the same, then either one can become the parent, but the new parent's rank is incremented by one.

Union by Rank

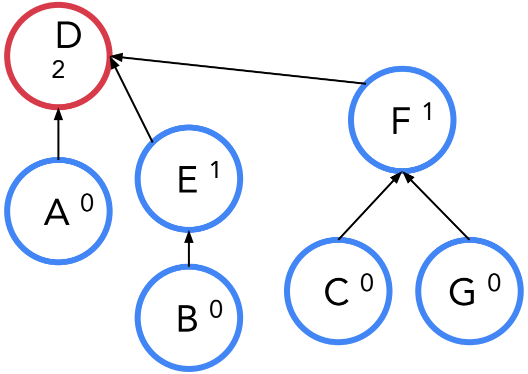
- After union(A, D), union(B, E), and union(C, F):



Union by Rank

- After union(C, G), and union(E, A):

- After union(B, G):



[Implement Union by Rank Here]

Union by Rank

```python
class UnionFind:
    def __init__(self, size):
        self.root = [i for i in range(size)]
        self.rank = [1] * size

    def find(self, x):
        while x != self.root[x]:
            x = self.root[x]
        return x

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.root[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.root[rootx] = rooty
            else:
                self.root[rootX] = rootY
                self.rank[rootY] += 1
```

For both union by rank and by size:

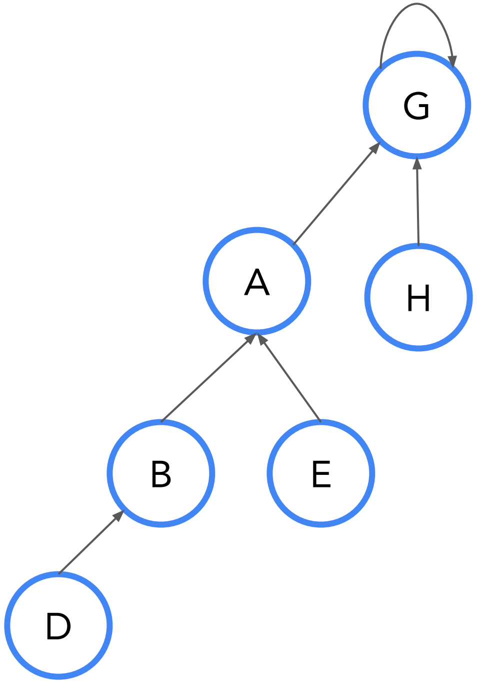
- O(**find**) = **O(logV)**
- O(**union**) = **O(logV)**
- O(**constructor**) = **O(V)**

Can we improve Find?

Path Compression

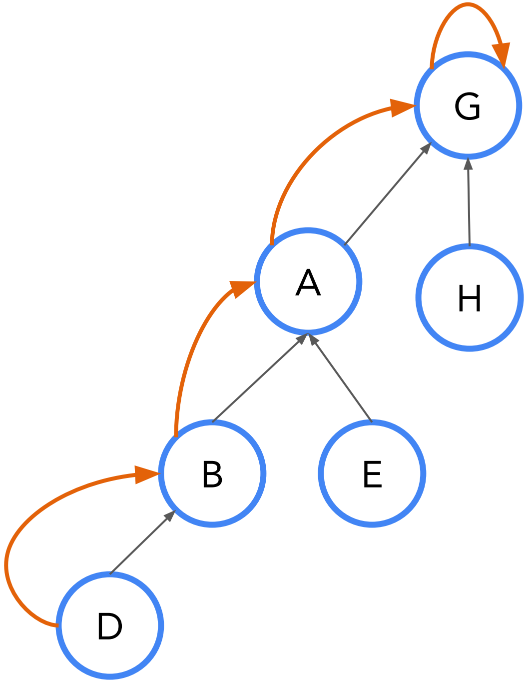
Who is the
representative of D?

How would we find it?



Path Compression

Who is the representative of D?



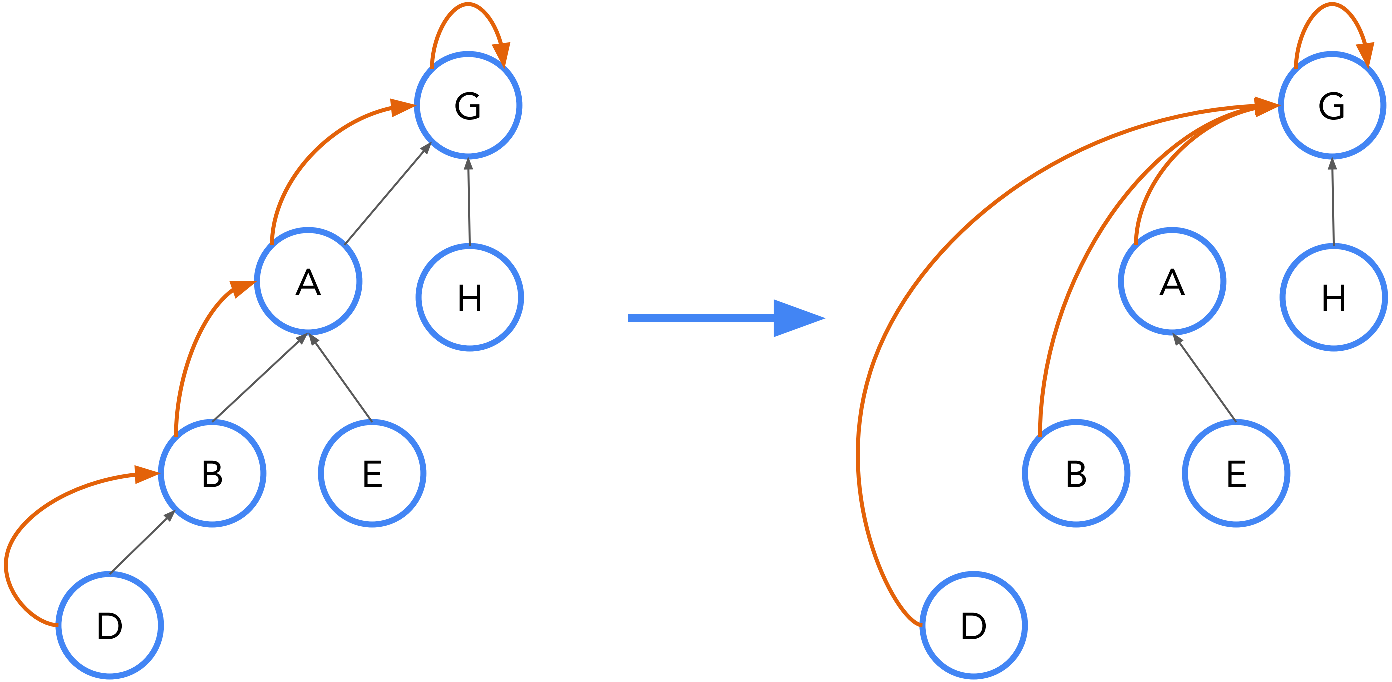
We would traverse up the tree until the root node

What information have we gained by going up the tree?

All the children we touched going up have the root as their representative

So, why not update them accordingly?

Path Compression



Implement Path Compression for Find

What if we use path compression and union by size (rank) together?

Union by Size with Path Compression

```
class UnionFind:
    def __init__(self, size):
        self.root = [i for i in range(size)]
        self.size = [1] * size

    def find(self, x):
        if x == self.root[x]:
            return x
        self.root[x] = self.find(self.root[x])
        return self.root[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.size[rootX] > self.size[rootY]:
                self.root[rootY] = rootX
                self.size[rootX] += self.size[rootY]
            else:
                self.root[rootX] = rootY
                self.size[rootY] += self.size[rootX]
```

Path Compression

- O(find) = amortized O(?)
- O(union) = amortized O(?)
- O(constructor) = O(?)

Union by Size with Path Compression

```
class UnionFind:
    def __init__(self, size):
        self.root = [i for i in range(size)]
        self.size = [1] * size

    def find(self, x):
        if x == self.root[x]:
            return x
        self.root[x] = self.find(self.root[x])
        return self.root[x]

    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.size[rootX] > self.size[rootY]:
                self.root[rootY] = rootX
                self.size[rootX] += self.size[rootY]
            else:
                self.root[rootX] = rootY
                self.size[rootY] += self.size[rootX]
```

Path Compression

- O(**find**) = **amortized O(1)**
- O(**union**) = **amortized O(1)**
- O(**constructor**) = **O(V)**

[**Proof**]

A2SV
Africa To Silicon Valley

Is there an iterative way of path compression?

Yes, there are.

You can implement the previous code iteratively or you can use Path Halving

Path Halving

- Makes every other node on the find path link to its grandparent.
- Retains the same worst-case complexity but are more efficient in practice.
- Simple to implement:

```python
def find(self, x):
    while x != self.root[x]:
        self.root[x] = self.root[self.root[x]]
        x = self.root[x]
    return x
```

Let’s solve a problem

Africa To Silicon Valley

Number of Provinces

Description Editorial Solutions (3.5K) Submissions

547. Number of Provinces

Medium 7.5K 287

Companies

There are `n` cities. Some of them are connected, while some are not. If city `a` is connected directly with city `b`, and city `b` is connected directly with city `c`, then city `a` is connected indirectly with city `c`.

A **province** is a group of directly or indirectly connected cities and no other cities outside of the group.

You are given an `n x n` matrix `isConnected` where `isConnected[i][j] = 1` if the i^{th} city and the j^{th} city are directly connected, and `isConnected[i][j] = 0` otherwise.

Return *the total number of* ***provinces***.

Implementation

```python
class Solution:
    def findCircleNum(self, isConnected):
        n = len(isConnected)
        dsu = UnionFind(n)
        numberOfComponents = n

        for i in range(n):
            for j in range(i + 1, n):
                if isConnected[i][j] and dsu.find(i) != dsu.find(j):
                    numberOfComponents -= 1
                    dsu.union_set(i, j)

        return numberOfComponents
```

Time and Space Complexity

Time complexity: $O(n^2)$

Space Complexity: $O(n)$

Checkpoint

Things to pay attention to

Pitfall 1: Improper initialization

```
class UnionFind:
    def __init__(self, size):
        self.parent = [0] * size  # Incorrect initialization
        self.rank = [0] * size
        self.count = size
```

Pitfall 1: Improper initialization

```python
class UnionFind:
    def __init__(self, size):
        # Incorrect initialization
        self.parent = [0] * size
        self.rank = [0] * size
        self.count = size
```

❌

```python
class UnionFind:
    def __init__(self, size):
        # Correct initialization
        self.parent = [i for i in range(size)]
        self.rank = [0] * size
        self.count = size
```

✅

Pitfall 2: Forgetting to use path compression

```python
class UnionFind:
    def __init__(self, size):
        . . .

    def find(self, member):
        # No path compression implemented
        while member != self.parent[member]:
            member = self.parent[member]
        return member

    def union_set(self, member1, member2):
        . . .
```

Pitfall 2: Forgetting to use path compression

```python
class UnionFind:
    def __init__(self, size):
        . . .

    def find(self, member):
        # No path compression implemented
        while member != self.parent[member]:
            member = self.parent[member]
        return member

    def union_set(self, member1, member2):
        . . .
```

❌

```python
class UnionFind:
    def __init__(self, size):
        . . .
    def find(self, member):
        # Path compression implemented
        root = member
        while root != self.parent[root]:
            root = self.parent[root]

        while member != root:
            parent = self.parent[member]
            self.parent[member] = root
            member = parent

        return root
```

✅

Pitfall 3: Forgetting to use rank-based union

```python
class UnionFind:
    def __init__(self, size):
        . . .

    def find(self, member):
        . . .

    def union_set(self, member1, member2):
        root1 = self.find(member1)
        root2 = self.find(member2)
        # No rank-based union implemented
        self.parent[root1] = root2
```

Pitfall 3: Forgetting to use rank-based union

```python
class UnionFind:
    def __init__(self, size):
        . . .

    def find(self, member):
        . . .

    def union_set(self, member1, member2):
        root1 = self.find(member1)
        root2 = self.find(member2)
        # No rank-based union implemented
        self.parent[root1] = root2
```

❌

```python
class UnionFind:
      . . .
    # rank-based union implemented
    def union(self, x, y):
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX != rootY:
            if self.rank[rootX] > self.rank[rootY]:
                self.root[rootY] = rootX
            elif self.rank[rootX] < self.rank[rootY]:
                self.root[rootX] = rootY
            else:
                self.root[rootY] = rootX
                self.rank[rootX] += 1
      . . .
```

✅

A2SV
Africa To Silicon Valley

Practice Questions

Check if There is a Valid Path in a Grid

Redundant Connection

Regions Cut by Slashes

Accounts Merge

Most Stones Removed with same Row or Column

Satisfiability of Equality Equations

Checking Existence of Edge Length Limited Paths

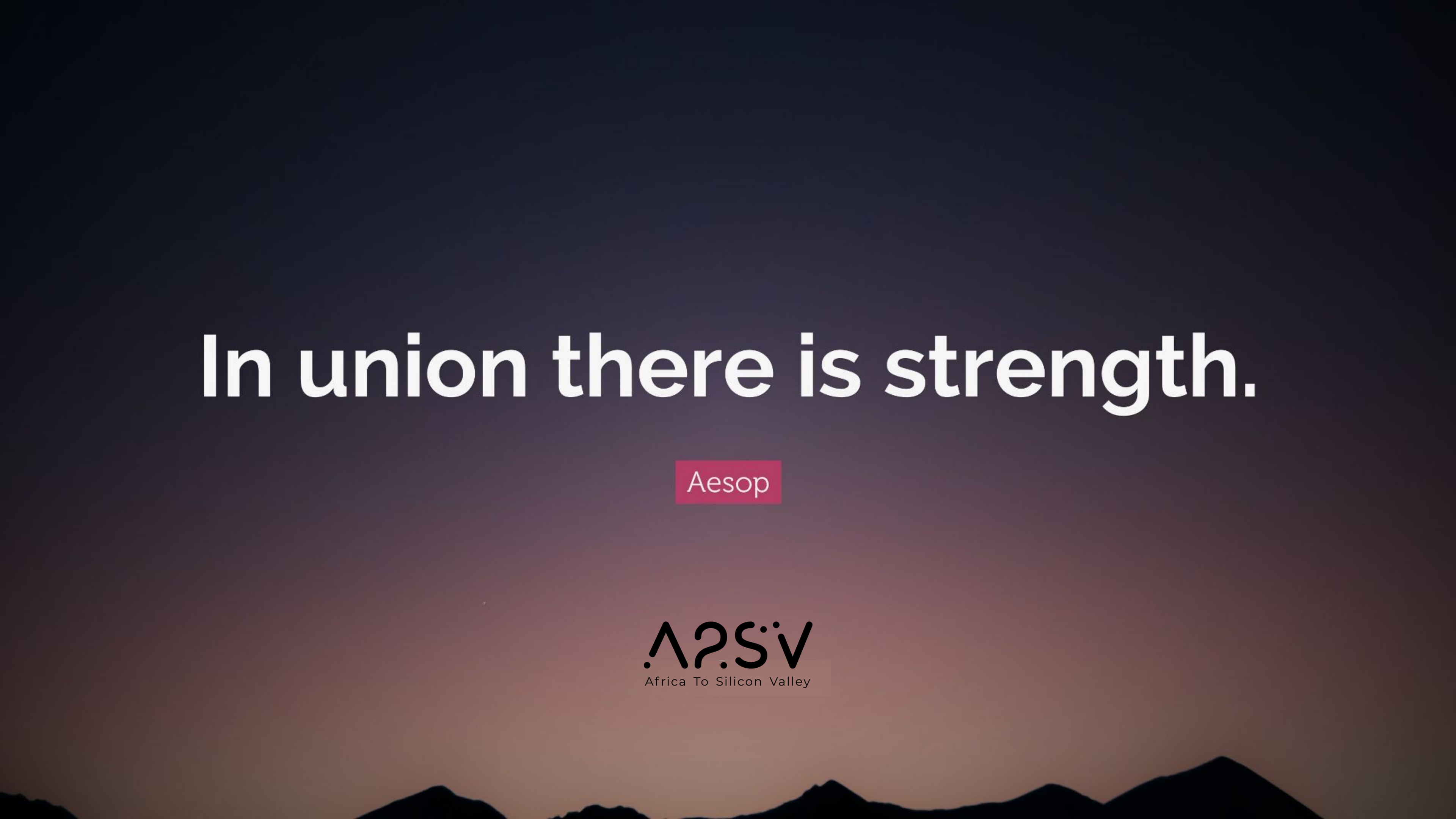
Remove Max Number of Edges to Keep Graph Fully Traversable

Minimize Malware Spread

A2SV Africa To Silicon Valley

Resources and more readables

- Codeforce EDU Course
- Leetcode Explore Card


In union there is strength.
Aesop
Africa To Silicon Valley