Topic Introduction

Today, we're taking a deep dive into the **Union Find** (also known as Disjoint Set Union, or DSU) data structure. If you've ever needed to quickly determine whether two items are in the same group, or efficiently merge groups together, Union Find is your friend.

What is Union Find?
Union Find is a data structure that keeps track of a set of elements split into disjoint (non-overlapping) groups. It supports two main operations:

- `find(x)`: Determine which group element `x` belongs to.
- `union(x, y)`: Merge the groups containing `x` and `y`.

How does it work?
Internally, each element starts as its own "parent." As unions happen, groups form by updating parents so all group members eventually point to a common "root." With two extra tricks—**path compression** (making group lookups faster by flattening the tree) and **union by rank/size** (always attaching smaller trees under larger ones)—Union Find can handle millions of queries with ease.

When is it useful in interviews?
Whenever you need to dynamically group things and answer "are these connected?" for pairs, Union Find is gold. This pops up in problems about connected components, network connectivity, grouping similar things, and more.

Quick Example:
Suppose you have five cities (A, B, C, D, E) and roads between them. As roads are built (`union`), you want to know if two cities are already connected (`find`). Union Find tracks these groupings efficiently, so you can answer queries like "Is city A connected to city E?" quickly.

Let's look at three classic problems that all use Union Find to manage connected components:

- Union Find (LeetCode Explore)
- Number of Islands
- Accounts Merge

Why are these grouped together?
All three require efficiently grouping related items:

- **Union Find**: The classic, minimal case—just implement and use the structure.
- **Number of Islands**: Grouping adjacent land cells into islands (connected components).
- **Accounts Merge**: Grouping emails/accounts that belong to the same real person.

We'll start with the basics, then ramp up the complexity to see Union Find in action.

Problem 1: Union Find (LeetCode Explore)

Problem Statement (reworded):
Given `n` elements (labeled `0` to `n-1`), build a Union Find data structure supporting `find` and `union`. After each operation, you should be able to find out whether two elements are in the same group.

LeetCode Link

Example:

Suppose `n = 5`.

Operations:

- union(1, 2)
- union(3, 4)
- find(1, 2) -> true (since 1 and 2 are now connected)
- find(1, 3) -> false (no connection)
- union(2, 3)
- find(1, 4) -> true (now all in the same group)

Thought Process:

Try this with pen and paper: Draw five dots, connect them as unions happen, and see how groups change.

Extra Test Case:

n = 4

Operations: union(0,1), union(2,3), union(1,2), find(0,3)

Expected Output: true

Brute-force Approach:

You could keep each group as a list or set, and merge lists when you union. But merging lists is slow—O(N) per union in the worst case.

Optimal Approach:

Instead, use Union Find:

- Each element has a parent pointer.
- To find the group, follow parent pointers until you reach the root.
- To union, set the root of one group to the root of the other.
- Optionally, use path compression to speed up repeated finds, and union by size/rank to keep the structure flat.

Let's implement this!

```
class UnionFind:
    def __init__(self, n):
        # Each element starts as its own parent
        self.parent = [i for i in range(n)]
        # For union by rank (optional)
        self.rank = [1] * n

    def find(self, x):
        # Path compression: flatten the tree
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
```

```
        # Find roots
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX == rootY:
            return  # Already in the same group
        # Union by rank: attach smaller tree under larger
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1

    def connected(self, x, y):
        # Check if two elements are in the same group
        return self.find(x) == self.find(y)
```

Time Complexity:

Each `find` and `union` is nearly O(1) (amortized), thanks to path compression and union by rank.

Space Complexity:

O(n) for the parent and rank arrays.

Code Walkthrough:

- `__init__`: Sets up `parent` (initially, each node is its own parent) and `rank` (for union by rank).
- `find`: Returns the root parent of `x`, compressing the path for future speed.
- `union`: Merges the sets containing `x` and `y`, keeping the tree flat.
- `connected`: Simply checks if two elements share the same root.

Trace Example:

Let's try:

```
uf = UnionFind(5)
uf.union(1, 2)
uf.union(3, 4)
print(uf.connected(1, 2)) # True
print(uf.connected(1, 3)) # False
uf.union(2, 3)
print(uf.connected(1, 4)) # True
```

Here's what happens:

- After first two unions: {0}, {1,2}, {3,4}
- After third union: {0}, {1,2,3,4}
- `connected(1, 4)` returns True.

Try This Yourself:

n = 6

Operations: union(0,1), union(1,2), union(3,4), find(2,4), union(2,3), find(5,0), find(4,0)

What should the outputs be?

Give it a Try!

Before moving on, sketch out the structure and try writing your own Union Find. Practice merging and querying groups.

Reflective prompt:

Did you know you could also solve some of these grouping problems using DFS or BFS? Try that for small cases and see the difference!

Problem 2: Number of Islands

Problem Statement (reworded):

Given a 2D grid of "1"s (land) and "0"s (water), count the number of islands. An island is a group of "1"s connected horizontally or vertically.

LeetCode Link

Example:

Input:

```
[
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
```

Output: 3

Explanation: There are three islands.

How is this similar to Problem 1?

We want to group all adjacent lands ("1"s) into connected components. Each island is a group. The union-find structure will help us form these groups efficiently.

Brute-force Approach:

DFS or BFS from each "1" not yet visited, marking all connected "1"s as visited. Time: O(mn), but recursive stack can be deep.

Optimal Approach (Union Find):

- Treat each cell as a node.
- For each "1", union it with its adjacent "1"s (up, down, left, right).
- After processing, count the number of unique root parents among all "1" cells — that's the number of islands.

Step-by-step:

- Initialize Union Find for all cells.
- For each cell, if it's "1", union with neighbors that are also "1".

- Count unique root parents for "1" cells.

Pseudocode:

```
function numIslands(grid):
    m, n = dimensions of grid
    uf = UnionFind(m * n)
    for each cell (i, j):
        if grid[i][j] == "1":
            for each neighbor (ni, nj):
                if neighbor is in bounds and grid[ni][nj] == "1":
                    uf.union(index(i, j), index(ni, nj))
    islands = set()
    for each cell (i, j):
        if grid[i][j] == "1":
            islands.add(uf.find(index(i, j)))
    return size of islands
```

`index(i, j)` converts 2D indices to 1D: `i * n + j`.

Example Dry Run:

With the example above, after unions, the roots will cluster into three separate groups (islands).

Try This Test Case:

Input:

```
[
  ["1","0","1","0"],
  ["0","1","0","1"],
  ["1","0","1","0"],
  ["0","1","0","1"]
]
```

How many islands? (Answer: 8, as each "1" is isolated.)

Trace of Code:

- For every "1", unions only happen with adjacent "1"s.
- At the end, counting unique roots among "1"s gives the number of islands.

Complexity:

- Time: O(mn * alpha(mn)), where alpha is the inverse Ackermann function (very slow-growing).
- Space: O(mn) for the parent array.

Problem 3: Accounts Merge

Problem Statement (reworded):

Given a list of accounts, each with a name and a list of emails, merge accounts that share any email. Output merged accounts (name + all emails, sorted).

LeetCode Link

What's new here?

Instead of a grid, you have lists of emails. Whenever two accounts share an email, they belong to the same user.

Challenge:

Emails can be shared across accounts in any order. Need to group all connected emails.

Brute-force:

Compare each account with every other. If they share an email, merge. This is O(N^2) and slow.

Optimal Approach (Union Find):

- Treat each email as a node.
- For each account, union all emails in that account.
- Afterwards, for each email, use find to determine its root. Group by root.
- For each group, output name and all emails (sorted).

Pseudocode:

```
function accountsMerge(accounts):
    uf = UnionFind(all unique emails)
    email_to_name = {}
    for account in accounts:
        name = account[0]
        emails = account[1:]
        for email in emails:
            email_to_name[email] = name
            uf.union(emails[0], email)
    root_to_emails = {}
    for email in all emails:
        root = uf.find(email)
        add email to root_to_emails[root]
    result = []
    for root, emails in root_to_emails:
        result.append([email_to_name[root]] + sorted(emails))
    return result
```

Example:

Input:

```
[
 ["John", "johnsmith@mail.com", "john_newyork@mail.com"],
 ["John", "johnsmith@mail.com", "john00@mail.com"],
 ["Mary", "mary@mail.com"],
 ["John", "johnnybravo@mail.com"]
]
```

Output:

```
[
```

```
 ["John", 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com'],
 ["Mary", "mary@mail.com"],
 ["John", "johnnybravo@mail.com"]
]
```

Try This Test Case:

Input:

```
[
 ["Alex", "alex1@mail.com", "alex2@mail.com"],
 ["Alex", "alex3@mail.com"],
 ["Alex", "alex2@mail.com", "alex3@mail.com"]
]
```

Expected Output:

```
[
 ["Alex", "alex1@mail.com", "alex2@mail.com", "alex3@mail.com"]
]
```

Complexity:

- Time: O(NK * alpha(NK)), where N is accounts and K is max emails per account.
- Space: O(NK) for storing parents and groups.

Reflective Nudge:

How else could you solve this? Could DFS/BFS work? Try it out and see which is easier to implement or understand.

Summary and Next Steps

Today, you practiced Union Find in three forms:

- The basic structure and its operations.
- Counting connected components in a 2D grid.
- Merging arbitrary groupings based on shared elements.

Key patterns to remember:

- Use Union Find when you need to dynamically group items and answer "are these connected?" queries.
- Path compression and union by rank make it super efficient.
- Mapping 2D/complex data to 1D (like grid cells or emails) is a common trick with Union Find.

Common mistakes:

- Forgetting path compression leads to slow queries.
- Not converting 2D indices to 1D properly.
- Mishandling merging when data is not in a simple flat structure (like emails).

Action List

- Solve all 3 problems on your own, even the one with code provided.
- Try solving Number of Islands and Accounts Merge with DFS or BFS for practice.
- Explore other problems using Union Find, like "Redundant Connection" or "Friend Circles."
- Compare your solution with others—look for better edge case handling and cleaner code.
- Don’t worry if you get stuck! The key is to keep practicing, reviewing, and building your confidence.

Keep going! Mastering Union Find unlocks a whole world of graph and grouping problems for you. See you in the next PrepLetter!