📘 UnionFind (Disjoint Set Union) – A Complete Guide

---

1. What is UnionFind?

UnionFind (also called **Disjoint Set Union, DSU**) is a data structure that efficiently manages a collection of disjoint sets and supports two key operations:

- **Find(x)** → Determine which set an element `x` belongs to (find the root parent).
- **Union(x, y)** → Merge the sets containing `x` and `y`.
- Connected(x,y) → Does X and Y have the same parent?

Each set is represented as a tree, where each node points to a parent. The **root parent** represents the set.

2. Why is UnionFind Needed?

Without UnionFind, checking if two nodes are connected often requires **DFS/BFS** which costs **O(N + E)** per query. With UnionFind:

- **Union and Find run in nearly O(1)** (amortized inverse Ackermann time).
- Perfect for dynamic connectivity problems where edges are added, and we frequently query connections.

Common Uses:

- Detecting **cycles** in graphs
- Building **Minimum Spanning Trees (Kruskal's algorithm)**
- Tracking **connected components**
- Clustering & network analysis
- Real-world: fraud detection, social networks, network percolation

3. UnionFind Implementation (JavaScript)

```
class UnionFind {
  constructor(n) {
    this.parent = Array.from({ length: n }, (_, i) => i);
    this.rank = new Array(n).fill(1); // size or depth
  }

  find(x) {
    if (this.parent[x] !== x) {
      this.parent[x] = this.find(this.parent[x]); // path compression
    }
    return this.parent[x];
  }

  union(x, y) {
    let rootX = this.find(x);
    let rootY = this.find(y);

    if (rootX === rootY) return false; // already connected

    // union by rank (attach smaller tree under bigger one)
    if (this.rank[rootX] < this.rank[rootY]) {
      this.parent[rootX] = rootY;
    } else if (this.rank[rootX] > this.rank[rootY]) {
      this.parent[rootY] = rootX;
    } else {
      this.parent[rootY] = rootX;
      this.rank[rootX] += 1;
```

```
    }
    return true;
  }

  connected(x, y) {
    return this.find(x) === this.find(y);
  }
}
```

---

4. 5 Common Interview Problems Solved with UnionFind

🧩 Problem 1: Cycle Detection in an Undirected Graph

Given an undirected graph, detect if it contains a cycle.

✅ **Solution**: Add edges one by one. If two nodes are already connected (same root) before adding an edge → cycle.

```
function hasCycle(n, edges) {
  let uf = new UnionFind(n);

  for (let [u, v] of edges) {
    if (!uf.union(u, v)) return true; // cycle detected
  }
  return false;
}
// Example:
console.log(hasCycle(3, [[0,1],[1,2],[2,0]])); // true
```

🧩 Problem 2: Number of Connected Components

Given `n` nodes and edges, return how many connected components exist.

✅ **Solution**: Each successful union reduces the number of components.

```
function countComponents(n, edges) {
  let uf = new UnionFind(n);
  let components = n;

  for (let [u, v] of edges) {
    if (uf.union(u, v)) components--;
  }
  return components;
}

// Example:
console.log(countComponents(5, [[0,1],[1,2],[3,4]])); // 2
```

🧩 Problem 3: Redundant Connection

In a graph that started as a tree and added one extra edge, return that edge.

✅ **Solution**: The edge that forms a cycle is the redundant one.

```
function findRedundantConnection(edges) {
  let n = edges.length;
  let uf = new UnionFind(n + 1);

  for (let [u, v] of edges) {
    if (!uf.union(u, v)) return [u, v];
  }
}

// Example:
console.log(findRedundantConnection([[1,2],[1,3],[2,3]])); // [2,3]
```

🧩 Problem 4: Kruskal's Algorithm (Minimum Spanning Tree)

Given weighted edges, find the MST cost.

✅ **Solution**: Sort edges by weight, union if not connected.

```
function kruskal(n, edges) {
  let uf = new UnionFind(n);
  edges.sort((a, b) => a[2] - b[2]);

  let mstCost = 0;
  let count = 0;

  for (let [u, v, w] of edges) {
    if (uf.union(u, v)) {
      mstCost += w;
```

```
      count++;
      if (count === n - 1) break;
    }
  }
  return mstCost;
}
// Example:
console.log(kruskal(4, [[0,1,1],[1,2,2],[2,3,3],[0,3,4]])); // 6
```

🧩 Problem 5: Accounts Merge (LeetCode 721)

Merge accounts if they share emails.

✅ **Solution**: Use UnionFind to group accounts by email.

```
function accountsMerge(accounts) {
  let uf = new UnionFind(accounts.length);
  let emailToId = {};

  for (let i = 0; i < accounts.length; i++) {
    for (let j = 1; j < accounts[i].length; j++) {
      let email = accounts[i][j];
      if (email in emailToId) {
        uf.union(i, emailToId[email]);
      } else {
        emailToId[email] = i;
      }
    }
  }
  let groups = {};
  for (let email in emailToId) {
```

```
    let root = uf.find(emailToId[email]);
    if (!(root in groups)) groups[root] = new Set();
    groups[root].add(email);
  }
  return Object.entries(groups).map(([id, emails]) =>
    [accounts[id][0], ...Array.from(emails).sort()]
  );
}
// Example:
console.log(accountsMerge([
  ["John","john@mail.com","john2@mail.com"],
  ["John","john2@mail.com","john3@mail.com"],
  ["Mary","mary@mail.com"]
]));
```

5. Summary

- **UnionFind** helps manage groups of connected elements efficiently.
- It supports find (with path compression) and union (with rank/size).
- It shines in connectivity, cycle detection, MST, and clustering problems.
- With **path compression + union by rank**, operations are effectively constant time.