Great question 🙌 — Union-Find (also called **Disjoint Set Union, DSU**) is one of the most powerful and elegant algorithms in graph theory. Let’s go step by step so you deeply understand it.

**🔹 What is Union-Find?**

Union-Find is a **data structure** that helps you keep track of a collection of disjoint (non-overlapping) sets.  
It supports two main operations efficiently:

1. **Find(x):**  
   Determines which set a particular element x belongs to. (Returns the "representative" / root of the set.)
2. **Union(x, y):**  
   Merges the sets containing x and y.

**🔹 Why is Union-Find Useful?**

* Detecting **cycles** in undirected graphs
* Building **Minimum Spanning Trees (Kruskal’s Algorithm)**
* Tracking **connected components**
* Solving problems like:
  + Friend groups (social networks)
  + Island merging
  + Dynamic connectivity queries

**🔹 How It Works**

Think of each set as a **tree**, where each node points to its parent.

* Initially: each node is its own parent (self-root).
* **Find:** climb up to the root to identify which set it belongs to.
* **Union:** connect the root of one tree to the root of another.

Example:

Initially:

1 2 3 4 5

(each is its own set)

Union(1, 2):

1 - 2 3 4 5

Union(3, 4):

1 - 2 3 - 4 5

Union(2, 3):

1 - 2 - 3 - 4 5

Now Find(4) and Find(1) will both return the same root → meaning they’re in the same set.

**🔹 Optimizations**

Two clever tricks make Union-Find **almost O(1)** in practice:

**1. Path Compression (in Find)**

When doing Find(x), make every node on the path point directly to the root.  
This flattens the tree → very fast future lookups.

**2. Union by Rank/Size**

When merging, attach the **smaller tree under the bigger one**.  
This avoids tall trees → keeps operations efficient.

**🔹 Time Complexity**

With both optimizations:

* **Find:** ~ O(1) (amortized, inverse Ackermann function ≈ constant)
* **Union:** ~ O(1)
* Much faster than naïve methods.

**🔹 Pseudocode**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

self.rank = [1] \* n # size of each tree

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x]) # path compression

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX == rootY:

return False # already connected

# union by rank

if self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

self.rank[rootY] += self.rank[rootX]

else:

self.parent[rootY] = rootX

self.rank[rootX] += self.rank[rootY]

return True

**🔹 Example Use Case: Detecting Cycle in Undirected Graph**

1. Initialize Union-Find for all nodes.
2. For each edge (u, v):
   * If find(u) == find(v) → cycle exists.
   * Else → union(u, v).

✅ So Union-Find = **super-fast way to manage connectivity in graphs.**

Do you want me to also show you **step-by-step visualization with a small graph (like edges forming sets)** so you can *see* how Union-Find detects cycles?

Perfect 👍 let’s walk through a **visual example** step by step.  
We’ll use Union-Find to check if an **undirected graph has a cycle**.

**🔹 Example Graph**

Nodes: 1, 2, 3, 4  
Edges:

(1, 2), (2, 3), (3, 4), (1, 4)

We’ll check edge by edge with Union-Find.

**Step 1: Initialization**

Each node is its own parent (disjoint sets):

1 2 3 4

Parent array: [1, 2, 3, 4]

**Step 2: Add edge (1, 2)**

* Find(1) = 1, Find(2) = 2 (different)
* Union → connect them

1 - 2 3 4

Parent: [1, 1, 3, 4]

**Step 3: Add edge (2, 3)**

* Find(2) → root = 1
* Find(3) = 3
* Different → Union(1, 3)

1 - 2 - 3 4

Parent: [1, 1, 1, 4]

**Step 4: Add edge (3, 4)**

* Find(3) → root = 1
* Find(4) = 4
* Different → Union(1, 4)

1 - 2 - 3 - 4

Parent: [1, 1, 1, 1]

**Step 5: Add edge (1, 4)**

* Find(1) → root = 1
* Find(4) → root = 1
* **Same root!** → Cycle detected ✅

**🔹 Key Takeaway**

Union-Find quickly detected that adding (1, 4) creates a cycle, because 1 and 4 were already connected through the path 1-2-3-4.

⚡ That’s the **visual flow of Union-Find**:

* Keeps track of connected components
* Detects cycles efficiently
* Works even on large graphs in nearly constant time

Would you like me to also give you a **practice problem (LeetCode style)** on Union-Find so you can apply this right away?

Awesome 🚀 Let’s try a **classic LeetCode-style Union-Find problem**.

**📝 Problem: Detect Cycle in an Undirected Graph**

You are given n nodes (labeled from 0 to n-1) and a list of edges where each edge is a pair of nodes (u, v).

Write a function to determine if the graph contains a cycle.

**Example 1:**

Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]

Output: False

Explanation: This is just a chain 0-1-2-3-4, no cycles.

**Example 2:**

Input: n = 4, edges = [[0,1],[1,2],[2,3],[0,3]]

Output: True

Explanation: 0-1-2-3-0 forms a cycle.

**🔹 Python Solution (Union-Find)**

class UnionFind:

def \_\_init\_\_(self, n):

self.parent = [i for i in range(n)]

self.rank = [1] \* n # for union by rank

def find(self, x):

if self.parent[x] != x:

self.parent[x] = self.find(self.parent[x]) # path compression

return self.parent[x]

def union(self, x, y):

rootX = self.find(x)

rootY = self.find(y)

if rootX == rootY:

return False # cycle detected

# Union by rank

if self.rank[rootX] > self.rank[rootY]:

self.parent[rootY] = rootX

elif self.rank[rootX] < self.rank[rootY]:

self.parent[rootX] = rootY

else:

self.parent[rootY] = rootX

self.rank[rootX] += 1

return True

def hasCycle(n, edges):

uf = UnionFind(n)

for u, v in edges:

if not uf.union(u, v):

return True # found a cycle

return False

# Test

print(hasCycle(5, [[0,1],[1,2],[2,3],[3,4]])) # False

print(hasCycle(4, [[0,1],[1,2],[2,3],[0,3]])) # True

👉 This combines **Union-Find with path compression + union by rank**, giving nearly O(1) per operation (actually O(α(N)), inverse Ackermann).

⚡ Challenge for you: Try solving **LeetCode 684 – Redundant Connection**, which is exactly this problem (but you return the extra edge that causes the cycle).

Would you like me to **build a step-by-step visual walkthrough of LeetCode 684 with Union-Find** so you can practice it deeper?