7. Number of Connected Components in an Undirected Graph

Pattern: Graph Traversal (DFS/BFS) + Union-Find (Disjoint Set Union)

Problem Statement

You are given an undirected graph with `n` nodes labeled from `0` to `n - 1`. The graph is represented as an integer `n` and a list of edges `edges`, where each `edges[i] = [a, b]` indicates an undirected edge between nodes `a` and `b`.

Return the number of **connected components** in the graph.

Sample Input & Output

```
Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2
Explanation: Nodes 0-1-2 form one component; nodes 3-4 form another.
```

```
Input: n = 5, edges = []
Output: 5
Explanation: No edges → each node is its own component.
```

```
Input: n = 1, edges = []
Output: 1
Explanation: Single node with no edges → one component.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        # STEP 1: Build adjacency list
        #   - Why? To enable efficient graph traversal.
        #   - Undirected → add both directions.
        graph = [[] for _ in range(n)]
        for a, b in edges:
            graph[a].append(b)
            graph[b].append(a)

        # STEP 2: Track visited nodes
        #   - Prevent revisiting and infinite loops.
        visited = [False] * n
        components = 0

        # STEP 3: DFS helper to mark all nodes in a component
        def dfs(node):
            visited[node] = True
            for neighbor in graph[node]:
                if not visited[neighbor]:
                    dfs(neighbor)

        # STEP 4: Iterate through all nodes
        #   - Each unvisited node starts a new component.
        for i in range(n):
            if not visited[i]:
                dfs(i)
                components += 1

        # STEP 5: Return total count
        return components

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.countComponents(5, [[0,1],[1,2],[3,4]]) == 2
```

```python
#   Test 2: Edge case - no edges
assert sol.countComponents(5, []) == 5

#   Test 3: Tricky/negative - single node
assert sol.countComponents(1, []) == 1

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

```
We'll walk through **Test 1**: `n = 5`, `edges = [[0,1],[1,2],[3,4]]`.

1. **Build graph**:
   - Initialize `graph = [[], [], [], [], []]`
   - Process `[0,1]` → `graph[0] = [1]`, `graph[1] = [0]`
   - Process `[1,2]` → `graph[1] = [0,2]`, `graph[2] = [1]`
   - Process `[3,4]` → `graph[3] = [4]`, `graph[4] = [3]`
   - Final `graph = [[1], [0,2], [1], [4], [3]]`

2. **Initialize**:
   - `visited = [False, False, False, False, False]`
   - `components = 0`

3. **Loop over nodes**:
   - `i = 0`: not visited → start DFS
     - `dfs(0)`:
       - Mark `visited[0] = True`
       - Visit neighbor `1` → not visited → `dfs(1)`
         - Mark `visited[1] = True`
         - Neighbors: `0` (visited), `2` → `dfs(2)`
           - Mark `visited[2] = True`
           - Neighbor `1` already visited → return
       - Backtrack → DFS ends
     - `components = 1`
   - `i = 1`: already visited → skip
   - `i = 2`: already visited → skip
```

```
   - `i = 3`: not visited → start DFS
     - `dfs(3)`:
       - Mark `visited[3] = True`
       - Visit `4` → not visited → `dfs(4)`
         - Mark `visited[4] = True`
         - Neighbor `3` visited → return
     - `components = 2`
   - `i = 4`: visited → skip

4. **Return** `2`

Final `visited = [True, True, True, True, True]`
Output: `2`
```

Complexity Analysis

- **Time Complexity**: `O(n + e)`

  We visit each node once (`n`) and each edge twice (once per direction, but still `O(e)` total). DFS visits every reachable node/edge exactly once.

- **Space Complexity**: `O(n + e)`

  Adjacency list uses `O(n + e)` space. Recursion stack in worst case (e.g., a line graph) uses `O(n)` space. So total is `O(n + e)`.

8. Graph Valid Tree

Pattern: Graph — Union-Find / DFS Cycle Detection

Problem Statement

You are given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges (each edge is a pair of nodes). Write a function to check whether these edges make up a valid tree.

A **valid tree** must satisfy **two conditions**:
1. There are **exactly `n - 1` edges**.
2. The graph is **fully connected and acyclic** (i.e., one connected component with no cycles).

Sample Input & Output

```
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: True
Explanation: 5 nodes, 4 edges, connected and no cycles → valid tree.
```

```
Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output: False
Explanation: Contains a cycle (1-2-3-1), so not a tree.
```

```
Input: n = 1, edges = []
Output: True
Explanation: Single node with no edges is a valid tree.
```

LeetCode Editorial Solution + Inline Tests

We'll use **Union-Find (Disjoint Set Union)** — a classic pattern for cycle detection in undirected graphs.
- If we ever try to union two nodes already in the same set → **cycle detected**.
- Also verify edge count = `n - 1`.

```python
from typing import List

class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        # STEP 1: Quick edge count check
        #   - A tree must have exactly n - 1 edges
        if len(edges) != n - 1:
            return False
```

```python
        # STEP 2: Initialize Union-Find parent array
        #   - Each node starts as its own parent
        parent = list(range(n))

        # Helper: Find root with path compression
        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x])  # Path compression
            return parent[x]

        # STEP 3: Process each edge
        #   - If two nodes share root → cycle → invalid
        for a, b in edges:
            root_a = find(a)
            root_b = find(b)
            if root_a == root_b:
                return False  # Cycle detected!
            parent[root_a] = root_b  # Union

        # STEP 4: Return True
        #   - Passed edge count + no cycles → valid tree
        return True

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.validTree(5, [[0,1],[0,2],[0,3],[1,4]]) == True

    #  Test 2: Edge case - single node
    assert sol.validTree(1, []) == True

    #  Test 3: Tricky/negative - cycle present
    assert sol.validTree(5, [[0,1],[1,2],[2,3],[1,3],[1,4]]) == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `n = 5`, `edges = [[0,1],[0,2],[0,3],[1,4]]`

Initial state:
- `parent = [0, 1, 2, 3, 4]`
- Edge count = 4 → equals `5 - 1` → proceed.

Edge [0,1]:
- `find(0)` → 0, `find(1)` → 1 → different roots
- Union: set `parent[0] = 1` → `parent = [1, 1, 2, 3, 4]`

Edge [0,2]:
- `find(0)` → `find(1)` → 1; `find(2)` → 2
- Union: `parent[1] = 2` → `parent = [1, 2, 2, 3, 4]`

Edge [0,3]:
- `find(0)` → `find(1)` → `find(2)` → 2; `find(3)` → 3
- Union: `parent[2] = 3` → `parent = [1, 2, 3, 3, 4]`

Edge [1,4]:
- `find(1)` → `find(2)` → `find(3)` → 3; `find(4)` → 4
- Union: `parent[3] = 4` → `parent = [1, 2, 3, 4, 4]`

No cycles found → return `True`.

Now **Test 3**: `edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]`
- First 3 edges connect 0–1–2–3 into one component.
- When processing `[1,3]`:
- `find(1)` → root = 3 (after unions)
- `find(3)` → root = 3
- Same root → **cycle detected** → return `False` immediately.

Complexity Analysis

- **Time Complexity**: `O(n · (n))  O(n)`

  We process `n - 1` edges. Each `find` uses path compression, making amortized cost nearly constant (= inverse Ackermann function).

- **Space Complexity**: `O(n)`

  The `parent` array stores one entry per node. No recursion stack (iterative union).