Topic Introduction

Today's theme is **Graph Structure and Connectivity** — a classic trio that shows up all over technical interviews. Graphs are everywhere, from social networks to maps to dependency systems, and understanding how to traverse and analyze them is *essential*.

What is a Graph?
A graph is a collection of nodes (also called vertices) connected by edges. Graphs can be directed or undirected, and edges can be weighted or unweighted. The two most common ways to represent graphs in code are adjacency lists (a dictionary where each node maps to a list of its neighbors) and adjacency matrices (a 2D array).

Why are Graphs Important in Interviews?
Graphs model relationships. Questions about connectivity, shortest paths, cycles, and components often hide in plain sight as interview questions. Mastery here means you can handle problems that at first seem unrelated.

How do Graph Traversals Work?
Two foundational techniques are:

- **Depth-First Search (DFS):** Explore as far as possible along each branch before backtracking. Think of it as a maze explorer who always takes one path until hitting a wall.
- **Breadth-First Search (BFS):** Explore all neighbors at the current depth before going deeper. Like ripples expanding from a pebble dropped in water.

Quick Example (not one of our problems):
Suppose you want to check if there's a path between node A and node B in a friendship network. You could use BFS or DFS to traverse from A and see if B is reachable.

Now let's dive into today's three problems:

- **Clone Graph** (LeetCode #133): Given a node from an undirected graph, create a deep copy of the entire graph.
- **Graph Valid Tree** (LeetCode #261): Given n nodes and a list of undirected edges, determine if the edges form a valid tree (connected and acyclic).
- **Number of Connected Components** (LeetCode #323): Find how many disconnected groups (components) exist in a graph.

Why are these grouped together?
They all deal with the core idea of traversing and understanding graph structure and connectivity:

- **Clone Graph:** Copying the whole structure using DFS or BFS.
- **Valid Tree:** Ensuring the graph is a single connected piece with no cycles (tree property).
- **Connected Components:** Counting separate "islands" in the graph.

The order we'll use today builds up from traversing and cloning, to checking connectivity, to counting separate pieces.

Problem 1: Clone Graph

Problem Link: LeetCode #133: Clone Graph

Problem Statement (in my words):

You're given a reference to a node in an undirected, connected graph. Each node has a value and a list of its neighbors. Return a *deep copy* of the entire graph, which means creating new nodes and edges so that the copy is completely independent of the original.

Example:

```
Input:
  Node 1 is connected to 2 and 4
  Node 2 is connected to 1 and 3
  Node 3 is connected to 2 and 4
  Node 4 is connected to 1 and 3

  (Graph visual: a square/cycle of 4 nodes)

Output:
  A new graph with the same structure as above, but all new nodes.
```

Step-by-step Thought Process:

- When you clone a graph, you need to make sure:
  - Each node is copied only once.
  - The neighbors of each node are also cloned and connected appropriately.
  - The graph might have cycles, so you must avoid infinite loops.

Try this example on pen and paper:

- Draw a square (cycle of 4 nodes) and try to "clone" each node and their connections one by one.
- Notice: If you just naively recurse, you'll keep copying the same nodes again and again!

Additional test case to try:

```
Input: Node 1 connected to Node 2; Node 2 connected to Node 1 (just two nodes connected)
Output: Deep copy of this simple two-node graph.
```

Brute-force approach:

You could try to copy each node as you visit it, but if you don't keep track of what you've already copied, you'll get stuck in cycles and duplicate nodes.

- **Time Complexity:** Potentially infinite if cycles exist!
- **Space Complexity:** Also unbounded if nodes are duplicated.

Optimal Approach:

Use **DFS** (or BFS) with a *visited* dictionary to keep track of already-copied nodes.

Core Pattern:

- Traverse with DFS.
- For each node, if it hasn't been cloned yet, clone it and recursively clone its neighbors.
- Use a mapping from original node to cloned node to avoid duplicates and handle cycles.

Python Solution:

```
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []

class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        if not node:
            return None

        # visited maps original nodes to their clones
        visited = {}

        def dfs(current):
            if current in visited:
                return visited[current]

            # Clone the node (without neighbors for now)
            clone = Node(current.val)
            visited[current] = clone

            # Clone all neighbors
            for neighbor in current.neighbors:
                clone.neighbors.append(dfs(neighbor))

            return clone

        return dfs(node)
```

Time Complexity: O(N), where N is the number of nodes (each node visited once).

Space Complexity: O(N) for the recursion stack and the visited map.

Explanation:

- The `visited` dictionary ensures we never clone the same node twice, which is crucial for graphs with cycles.
- The `dfs` function clones the current node, then recursively clones its neighbors.
- The base case is when a node is already in `visited` — we return the already-cloned node.

Trace with Example:

Suppose the input is:

- 1 -- 2

- | |
- 4 -- 3

When starting at node 1:

- Clone node 1, add to visited.
- Clone neighbor 2:
  - Clone node 2, add to visited.
  - Clone neighbor 3:
    - Clone node 3, add to visited.
    - Clone neighbor 4:
      - Clone node 4, add to visited.
      - Clone neighbor 1: already visited, use cloned version.
      - Clone neighbor 3: already visited, use cloned version.
    - Continue back up, linking neighbors as you go.

Another test case for you to try:

Input:
Node 1 connected to Node 2 and Node 3; Node 2 connected to Node 1; Node 3 connected to Node 1.

Try drawing and cloning this by hand.

Take a moment to solve this on your own before jumping into the solution.

Reflective Prompt:
Did you know this can also be solved using BFS instead of DFS? Give that a shot once you're comfortable with this version!

Problem 2: Graph Valid Tree

Problem Link: LeetCode #261: Graph Valid Tree

Problem Statement (in my words):
Given `n` nodes labeled from 0 to n-1 and a list of undirected edges, determine if the graph forms a *valid tree*:

- It must be connected (there's a path between every pair of nodes).
- It must be acyclic (no cycles).

Example:

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

Explanation: All nodes are connected, and there are no cycles, so this is a valid tree.
```

How is this similar/different to Clone Graph?
Both require traversing the graph, but here we're checking two properties: connectivity (all nodes are reachable) and acyclicity (no cycles). Instead of cloning, we're verifying structure.

Brute-force approach:

Try every possible path to check for cycles and see if every node can be reached. This is infeasible for large graphs.

- **Time Complexity:** Exponential for cycle checking.
- **Space Complexity:** High for storing all paths.

Optimal Approach:

Use **DFS or BFS** to traverse from one node:

- Use a visited set to keep track of visited nodes.
- If you try to visit a node that's already in visited and not the parent, you've found a cycle.
- After traversal, check if all nodes were visited (to confirm connectivity).
- Also, a quick check: For a tree with n nodes, there must be exactly n-1 edges!

Step-by-step Logic:

- If number of edges != n-1, return False immediately.
- Build the adjacency list for the graph.
- Use DFS to traverse the graph, detecting cycles.
- After traversal, check if all nodes were visited.

Pseudocode:

```
function validTree(n, edges):
    if len(edges) != n-1:
        return False

    build adjacency_list from edges
    visited = set()

    function dfs(node, parent):
        add node to visited
        for neighbor in adjacency_list[node]:
            if neighbor == parent:
                continue
            if neighbor in visited or not dfs(neighbor, node):
                return False
        return True

    if not dfs(0, -1):
        return False

    return len(visited) == n
```

Example Trace:

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

- Start DFS from node 0.

- Visit 0, neighbor is 1.
- Visit 1, neighbors are 0, 2, 3.
- 0 is parent (skip), visit 2.
- 2 has neighbor 1 (parent, skip).
- Back to 1, visit 3.
- 3 has neighbor 1 (parent, skip).
- End: visited = {0,1,2,3} (all nodes).

Another test case to try:

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

Try detecting the cycle using the pseudocode above.

Time Complexity: O(N) for traversal.
Space Complexity: O(N) for adjacency list and visited set.

Problem 3: Number of Connected Components

Problem Link: LeetCode #323: Number of Connected Components in an Undirected Graph

Problem Statement (in my words):
Given n nodes labeled from 0 to n-1 and a list of undirected edges, return the *number of connected components* in the graph (i.e., groups of nodes all connected with each other and not with nodes in other groups).

What's different here?
Instead of checking for cycles or connectivity, we want to count *how many separate groups* exist.

Brute-force approach:
For each node, check which nodes it can reach (BFS/DFS), then mark them all as visited. Repeat for all unvisited nodes.

- **Time Complexity:** O(N + E), where E is the number of edges.
- **Space Complexity:** O(N).

Optimal Approach:
Use DFS (or BFS) repeatedly:

- Initialize a count to 0 and a visited set.
- For each node:
    - If node not in visited:
        - Start DFS from node, marking all reachable nodes as visited.
        - Increment count.
- Continue until all nodes are visited.

Pseudocode:

```
function countComponents(n, edges):
    build adjacency_list from edges
    visited = set()
```

```
    count = 0

    for node in range(0, n):
        if node not in visited:
            dfs(node)
            count += 1

    return count

function dfs(node):
    add node to visited
    for neighbor in adjacency_list[node]:
        if neighbor not in visited:
            dfs(neighbor)
```

Example Trace:

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

- Nodes 0-2 are all connected, so one component.
- Nodes 3-4 are together, second component.
- Node 5 is alone, third component.

Another test case to try:

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

Expected Output: 2

Try implementing this logic and dry-running through the test case to see how the count increases.

Time Complexity: O(N + E)
Space Complexity: O(N)

Reflective Prompt:
Can you think of how to solve this problem using Union-Find (Disjoint Set Union) instead of DFS/BFS? Try implementing that for extra practice!

Summary and Next Steps

Let’s recap:

- All three problems are about **graph traversal and understanding structure and connectivity**.
- The same traversal pattern (DFS/BFS) appears in all, but is used for:
    - Cloning the graph structure (problem 1)
    - Checking for cycles and connectivity (problem 2)
    - Counting separate groups (problem 3)

- **Key insights:**
  - Always track visited nodes to avoid cycles/infinite loops.
  - For trees, check for n-1 edges and no cycles.
  - For connected components, use repeated traversals.

Common traps:

- Forgetting to handle cycles, leading to infinite recursion.
- Not tracking visited nodes properly.
- Not checking all nodes in disconnected graphs.

Action List

- Solve all three problems on your own, including the one with code provided.
- Try solving Problem 2 and 3 with Union-Find (Disjoint Set Union) for a different perspective.
- Explore other graph problems: shortest path (BFS), detecting cycles in directed graphs, etc.
- Compare your solutions with others — pay attention to edge case handling and style.
- If you get stuck, that’s totally normal! The more you practice, the more “graph sense” you’ll build.

Keep traversing, keep connecting, and keep leveling up your graph skills!