LeetCode 133 — Clone Graph

🔗 https://leetcode.com/problems/clone-graph/

1. Problem Title & Link

Title: LeetCode 133: Clone Graph
Link: https://leetcode.com/problems/clone-graph/

2. Problem Statement (Short Summary)

Given a reference to a node in an undirected connected graph, return a **deep copy (clone)** of the entire graph. Each node contains a value and a list of neighbors. The clone must be a new graph with new nodes and the same structure.

3. Examples (Input → Output)

Example 1

Input: adjacency list [[2,4],[1,3],[2,4],[1,3]] (1↔2↔3↔4 cycle)
Output: a new graph identical in structure to the original (nodes values preserved, separate objects).

Example 2

Input: [] (empty graph)
Output: []

Example 3

Input: [[2],[1]] (1↔2)
Output: cloned graph with two nodes connected.
(Explanation: LeetCode represents graphs with adjacency lists; returned node should be the clone of the input node.)

4. Constraints

- Number of nodes in the graph ≤ 100
- Node values are 1..100 (unique for nodes in input)
- Graph is undirected and may contain cycles/self-loops
- Must create deep copy (no shared references to original)
- Return the clone of the given input node (or null for empty input)

5. Core Concept (Pattern / Topic)

⭐ **Graph Traversal + Hashmap (BFS or DFS)**

Pattern: clone while traversing; use map from original node → cloned node to prevent cycles and repeated cloning.

6. Thought Process (Step-by-Step Explanation)

Brute/naive idea

Try to copy nodes and neighbors without tracking → will duplicate nodes and infinite loops on cycles.

Correct idea (use map)

1. Maintain visited map: orig_node -> cloned_node.
2. Traverse the graph (BFS or DFS).
    - When you see a node not in map, create its clone and put in map.
    - For each neighbor, ensure neighbor is cloned (create if needed) and append cloned neighbor to current clone's neighbor list.
3. Return map[start_node].

This handles cycles/self-loops because map prevents revisiting clones.

BFS vs DFS

- Both work. BFS is iterative and avoids recursion depth issues; DFS (recursive or iterative) is fine for small graphs.

7. Visual / Intuition Diagram (ASCII)

Original:

```
1 -- 2
|    |
4 -- 3
```

Clone process (map):

orig: 1 -> clone: 1'

orig: 2 -> clone: 2'

...

Link neighbors accordingly:

1'.neighbors = [2',4']

8. Pseudocode (Language Independent)

```
function cloneGraph(node):
    if node is null: return null

    map = {}  # original node -> cloned node
    queue = [node]
    map[node] = new Node(node.val)

    while queue not empty:
        curr = queue.pop()
        for nei in curr.neighbors:
            if nei not in map:
```

```
                map[nei] = new Node(nei.val)
                queue.push(nei)
            map[curr].neighbors.append(map[nei])

    return map[node]
```

9. Code Implementation

✅ Python (BFS)

```python
# 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 []

from collections import deque

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

        old_to_new = {node: Node(node.val)}
        q = deque([node])

        while q:
            cur = q.popleft()
            for nei in cur.neighbors:
                if nei not in old_to_new:
                    old_to_new[nei] = Node(nei.val)
                    q.append(nei)
                old_to_new[cur].neighbors.append(old_to_new[nei])

        return old_to_new[node]
```

✅ Python (DFS recursive)

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

        old_to_new = {}
```

```
        def dfs(n):
            if n in old_to_new:
                return old_to_new[n]
            copy = Node(n.val)
            old_to_new[n] = copy
            for nei in n.neighbors:
                copy.neighbors.append(dfs(nei))
            return copy

        return dfs(node)
```

✅ **Java (BFS)**

```
// Definition for Node.
// class Node {
//     public int val;
//     public List<Node> neighbors;
//     public Node() { val = 0; neighbors = new ArrayList<Node>(); }
//     public Node(int _val) { val = _val; neighbors = new ArrayList<Node>(); }
//     public Node(int _val, ArrayList<Node> _neighbors) { val = _val; neighbors
= _neighbors; }
// }

import java.util.*;

class Solution {
    public Node cloneGraph(Node node) {
        if (node == null) return null;

        Map<Node, Node> map = new HashMap<>();
        Queue<Node> q = new LinkedList<>();
        q.add(node);
        map.put(node, new Node(node.val));

        while (!q.isEmpty()) {
            Node curr = q.poll();
            for (Node nei : curr.neighbors) {
                if (!map.containsKey(nei)) {
                    map.put(nei, new Node(nei.val));
                    q.add(nei);
                }
                map.get(curr).neighbors.add(map.get(nei));
            }
```

```
        }
        return map.get(node);
    }
}
```

✅ **Java (DFS recursive)**

```
class Solution {
    private Map<Node, Node> map = new HashMap<>();

    public Node cloneGraph(Node node) {
        if (node == null) return null;
        if (map.containsKey(node)) return map.get(node);

        Node copy = new Node(node.val);
        map.put(node, copy);
        for (Node nei : node.neighbors) {
            copy.neighbors.add(cloneGraph(nei));
        }
        return copy;
    }
}
```

10. Time & Space Complexity

- **Time:** O(N + E) — visit each node and edge once during traversal.
- **Space:** O(N) — for the old_to_new map + BFS queue / recursion stack (N = #nodes, E = #edges).

11. Common Mistakes / Edge Cases

- ❌ Not handling null input (should return null)
- ❌ Forgetting to use a map → infinite loop for cycles or duplicated clones
- ❌ Copying values only and not wiring neighbors correctly
- ❌ Using node values as keys (values may not be unique) — always use node references/objects as keys
- Edge cases:
    - Empty graph (node == null)
    - Single node with self-loop
    - Graph with cycles
    - Graph with isolated nodes (LeetCode input gives connected graph by problem statement, but handle general cases)

12. Detailed Dry Run (Step-by-Step Table)

Graph:

1 -- 2

| |

4 -- 3

(start at node 1)

Step	Action
init	old_to_new[1] = 1' ; queue = [1]
pop 1	visit neighbors 2 and 4; create 2' and 4'; link 1'.neighbors = [2',4']; queue add 2,4
pop 2	neighbors 1 and 3; 1 already in map; create 3'; link 2'.neighbors += [1',3']; queue add 3
pop 4	neighbors 1 and 3; 1 in map; 3 already created; link 4'.neighbors += [1',3']
pop 3	neighbors 2 and 4; both in map; link 3'.neighbors += [2',4']
done	return 1' (clone of start) — fully wired deep copy

13. Common Use Cases (Real-Life / Interview)

- Deep-copying complex object graphs (serialization, cloning structures)
- Network topology cloning for simulation/testing
- Object graph cloning in compilers/AST transformations
- Interview focus: handling cycles and preserving identity mapping

14. Common Traps (Important!)

- Using node.val as a unique key — **wrong** unless guaranteed unique (LeetCode values are unique in tests but don't rely on it).
- Accidentally shallow-copying neighbor lists (references to original nodes).
- Not preserving self-loops correctly.

15. Builds To (Related LeetCode Problems)

- LC 133 → foundational for:
    - LC 207 (Course Schedule) — graph trav & cycle detect
    - LC 547 (Number of Provinces) — connected components
    - LC 323 (Number of Connected Components in an Undirected Graph)
    - LC 271/286 (graph cloning variations)

16. Alternate Approaches + Comparison

- **BFS cloning** — iterative, good for breadth-first creation, avoids recursion depth issues.
- **DFS cloning (recursive)** — concise and direct mapping from recursion to clone logic.
- Both use map and have same complexity; choice depends on style and recursion depth constraints.

17. Why This Solution Works (Short Intuition)

We create a clone for each original node exactly once (stored in a map). While traversing edges, we link the clones accordingly. The map prevents infinite loops and ensures all references in the clone graph refer to cloned nodes, producing a correct deep copy.

18. Variations / Follow-Up Questions

- Clone a **directed** graph (same approach).
- Support graphs with additional properties (weights/labels) — copy those fields too.
- Clone only a **subgraph** reachable within k steps.
- Serialize + deserialize the graph (use adjacency list encoding).