Clone Graph

Difficulty	Medium
Category	Graph
Question	https://leetcode.com/problems/clone-graph/
Solution	https://youtu.be/mQeF6bN8hMk
Status	Done

Question

> Given a reference of a node in a **connected** undirected graph.
>
> Return a **deep copy** (clone) of the graph.
>
> Each node in the graph contains a value (`int`) and a list (`List[Node]`) of its neighbors.

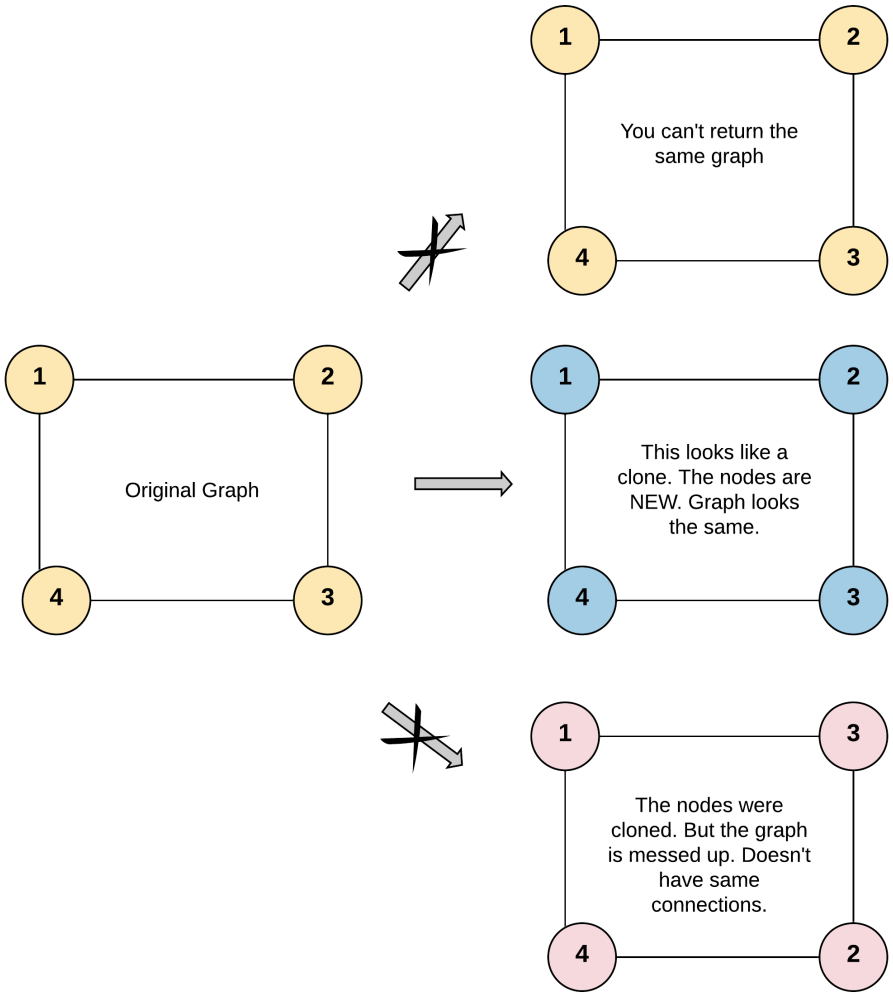
```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

> **Test case format:**
>
> For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with `val == 1`, the second node with `val == 2`, and so on. The graph is represented in the test case using an adjacency list.

> **An adjacency list** is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.
>
> The given node will always be the first node with `val = 1`. You must return the **copy of the given node** as a reference to the cloned graph.

Example

Example 1:



```
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
```

Example 2:


```
Input: adjList = [[]]
Output: [[]]
Explanation: Note that the input contains one empty list. The graph consists of only one n
ode with val = 1 and it does not have any neighbors.
```

Example 3:

```
Input: adjList = []
Output: []
Explanation: This an empty graph, it does not have any nodes.
```

Idea

💡 Use BFS to assign current neighbors, and then iterate through all neighbors to find their neighbors and so on. Use HashMap to keep track of assigned nodes

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 []
"""

from typing import Optional
class Solution:
    def cloneGraph(self, node: Optional['Node']) -> Optional['Node']:
        # Create a dictionary to map original nodes to their corresponding cloned nodes.
        oldToNew = {}

        # Define a depth-first search (DFS) function to clone the graph.
        def dfs(node):
            # If the node has already been cloned, return the corresponding cloned node.
            if node in oldToNew:
                return oldToNew[node]

            # Create a new node with the same value as the original node.
            copy = Node(node.val)

            # Add the new node to the dictionary to track the mapping.
            oldToNew[node] = copy

            # Iterate through the neighbors of the original node and clone them recursively.
            for nei in node.neighbors:
                copy.neighbors.append(dfs(nei))

            # Return the cloned node.
            return copy

        # If the input node is not None, start the DFS process to clone the graph.
        # If the input is None, return None to indicate an empty graph.
        return dfs(node) if node else None
```

Explanation

> 1. The `Node` class is defined to represent a node in the graph. Each node has a `val` (value) and a `neighbors` list, which

contains references to neighboring nodes. The `neighbors` list represents the edges of the graph.

2. The `Solution` class defines a `cloneGraph` method that takes a node as input and returns a deep copy of the entire graph.
3. Inside the `cloneGraph` method, the `oldToNew` dictionary is initialized. This dictionary will be used to map nodes from the original graph to their corresponding nodes in the copied graph.
4. The `dfs` function is defined as an inner function within `cloneGraph`. It takes a node as an argument and returns a deep copy of the subgraph rooted at that node.
5. The `dfs` function first checks if the input node already exists in the `oldToNew` dictionary. If it does, it means that the node has already been copied, so it returns the corresponding copied node from the dictionary.
6. If the node is not in the dictionary, it creates a new node with the same value as the original node and adds it to the dictionary to keep track of the mapping between old and new nodes.
7. It then iterates through the neighbors of the input node using a for loop. For each neighbor, it recursively calls the `dfs` function to copy the neighbor and its subgraph. The copied neighbor is added to the `neighbors` list of the current copied node.
8. The `dfs` function returns the copied node, and the process continues until all nodes and their connections in the graph have been copied.

9. Finally, the `cloneGraph` method is called with the input node. If the input node is not None (i.e., the graph is not empty), the `dfs` function is invoked to create the deep copy of the graph. If the input node is None, it returns None to indicate that the copied graph is also empty.

In summary, this code performs a deep copy of a graph represented as nodes with values and neighbors using a depth-first search (DFS) approach. It ensures that the copied graph maintains the same structure and connections as the original graph while avoiding cycles. The `oldToNew` dictionary is crucial for tracking the mapping between old and new nodes.