133. Clone Graph **Breadth-First Search** Medium **Depth-First Search** Graph **Hash Table**

Problem Description

In a graph, nodes are interconnected through edges. To represent this in code, each node has an integer value and a list of its neighboring nodes. The goal is to copy all the nodes and their connections in such a way that if you manipulate the copied graph, the original graph remains unaffected.

The problem is about creating a deep copy of a connected undirected graph. A deep copy means a new graph where each node and

edge is a replica of the original graph but is an entirely new instance, with no shared references with the original graph.

The representation of the graph is done using an adjacency list, which is an array of lists where each list represents a node and contains all of its neighbors. For the purposes of this problem, every node's value is assumed to be unique and equal to its 1-based

The challenge here is to ensure that while copying the nodes, their neighbors are also copied correctly, and any neighbor connections in the new graph mirror those in the original graph, preserving the structure.

To solve the problem, the idea is to traverse the graph, starting from the given node, and create new nodes as we go. Since the original graph could have cycles or shared neighbors, it is crucial to keep track of the nodes that have already been copied to avoid

infinite loops and ensure that neighbors are not duplicated in the new graph. The algorithm involves a depth-first search (DFS) traversal with the following steps:

been visited, and also provide a reference to its cloned counterpart. 2. Define a recursive function clone(node) that takes a node from the original graph.

1. Initialize a visited dictionary that will map original nodes to their corresponding cloned nodes. This will help check if a node has

• If the node is None, return None (this handles empty graph cases).

index in the adjacency list, simplifying the graph's representation and the copying process.

- If the node has already been visited, return its cloned counterpart from the visited dictionary. If the node is new, create a clone with the same value.
- Iteratively clone all the neighbor nodes using the same clone function and append them to the neighbors list of the cloned node. 3. Start the cloning process by calling clone(node) on the given node and return its result, which is a reference to the cloned
 - graph's starting node.
- By following these steps, we ensure that each node is visited once, and a deep copy of the graph is created, with all the original connections between the nodes preserved in the new graph.

Store this cloned node in the visited dictionary with the original node as the key.

- The solution leverages the depth-first search (DFS) algorithm to iterate through the graph, and a hash map (defaultdict in Python) to keep track of visited nodes and their clones. This ensures each node is copied exactly once, and we do not run into an infinite
- Here's a step-by-step breakdown of the solution approach:

• A dictionary visited is created to map original nodes to their cloned counterparts. This is essential to:

• Retrieve the cloned version of a node to set up the neighbors references correctly in the cloned graph. 2. Clone Function:

3. Base cases:

4. Cloning Process:

6. Returning the Clone:

Example Walkthrough

1. Initialization:

2. Start Cloning:

4. Node 2:

5. **Node 3:**

C3.

link both as neighbors to C4.

interconnected following their original structure.

7. Completing the Graph:

6. **Node 4:**

are:

7. Invocation:

neighbors properly linked.

1. Initialization:

Solution Approach

loop due to cycles in the graph.

• The clone function is a recursive helper function that performs the DFS. • The function signature clone(node) takes a node from the original graph as its argument.

 \circ If the node is being visited for the first time, we create a new Node instance with the same value (c = Node(node.val)).

We then iterate over the neighbors of the current node. For each neighbor, we recursively call the clone function (clone(e))

o Once all neighbors for the node have been processed, the function returns c, the clone of the current node, along with all its

• The DFS cloning begins with return clone(node), where node is the reference to the node that was provided as input to the

• If the input node is None, meaning either the graph is empty or we've hit the end of a path, the function returns None. • If the input node is found in the visited dictionary, it means we've already created a clone of this node. We immediately

and append the result to the c.neighbors.

node and edge is visited once in the traversal and copying process.

Track which nodes have already been copied.

• The new node c is then stored in visited[node]. This step marks the node as copied and provides a reference for its clone. 5. Neighbors Copy:

return the cloned node to avoid duplicating nodes or falling into a cycle.

- This recursive approach ensures that we explore each neighbor (and subsequently, each neighbor's neighbor, and so on), cloning nodes and stitching together the cloned graph as we traverse the original graph.
- cloneGraph function. Upon completion, this invocation returns a reference to the newly created deep copy of the graph, which can now be used independently of the original graph.

This solution scales well because the DFS and usage of a hash map ensure that each node is visited and copied exactly once. The

complexity of the algorithm is O(N + M), where N is the number of nodes and M is the number of edges in the graph, since every

following graph: 1 Node 1 -- Node 2 3 Node 4 -- Node 3

Let's say we have a small undirected graph represented by the adjacency list [[2,4],[1,3],[2,4],[1,3]], which corresponds to the

 We begin by calling clone(node) on the node with value 1 (Node 1). 3. **Node 1:**

• Since Node 1 isn't in visited, we create a cloned node C1 and add it to visited with visited[1] = C1.

• We then clone Node 2's neighbors (Node 1 and Node 3). Since Node 1 is already visited, we link C1 (Node 1's clone) as a neighbor to C2.

 When cloning Node 2's neighbors, we call clone(node) on Node 3. Node 3 isn't in visited, so we create a clone C3 and add it to visited.

• We clone Node 3's neighbors (Node 2 and Node 4). Node 2's clone, C2, is already in visited, so we link C2 as a neighbor to

 Now, from Node 1, we clone the second neighbor, Node 4, by calling clone(node) on it. • We create C4 since Node 4 isn't in visited and add it to visited. • We clone Node 4's neighbors (Node 1 and Node 3). Clones for both these nodes (C1 and C3) are found in visited, so we

• We initialize an empty dictionary called visited to keep track of the cloned nodes.

We proceed to clone Node 1's neighbors (Node 2 and Node 4).

Node 2 isn't in visited, so we create a clone C2 and add it to visited.

• We call clone(node) on Node 1's first neighbor, Node 2.

8. Return: • We return C1, which now has C2 and C4 as its neighbors, representing the start of the cloned graph.

The cloned graph is now a deep copy of the original and can be manipulated without affecting the original graph. The connections

Each Ci denotes the cloned node corresponding to the original node of the same index. The deep copy process ensures that even if

the initial graph had a more complex structure with cycles, the algorithm would have correctly cloned it without entering an infinite

With all nodes visited, the DFS cloning is complete. Each node in the graph is cloned exactly once, and their neighbors are

loop, since each node is visited only once. Python Solution

class Node:

class Solution:

9

10

11

14

15

16

17

19

20

21

22

23

26

27

28

29

33

34

35

43

44

45

46

47

8

9

10

11

13

14

15

16

17

18

20

21

22

23

24

25

27

28

29

30

31

11

13

16

18

19

20

21

23

24

25

26

27

29

30

31

32

33

34

36

37

38

39

42

6

8

15

16

17

18

19

20

22

23

24

25

26

28

29

30

31

32

33

34

36

35 }

41 };

12 };

15 public:

class Solution {

from collections import defaultdict

self.val = val

Definition for a Node in the graph.

visited = defaultdict(Node)

if node is None:

return None

if node in visited:

return clone_node

return clone(node)

public List<Node> neighbors;

neighbors = new ArrayList<>();

neighbors = new ArrayList<>();

neighbors = _neighbors;

Node(int _val) : val(_val) {}

// Function to clone a graph.

Node* cloneGraph(Node* node) {

if (!node) return nullptr;

visited[node] = cloneNode;

// Return the cloned node.

return cloneNode;

std::unordered_map<Node*, Node*> visited;

Node* cloneNode = new Node(node->val);

for (auto& neighbor: node->neighbors) {

// Iterate over all neighbors of the input node.

function cloneGraph(originalNode: Node | null): Node | null {

if (originalNode === null) return null;

const visitedMap = new Map<Node, Node>();

// Traverse the graph in a breadth-first manner

// Process all the neighbors of the current node.

for (let neighbor of currentNode.neighbors) {

if (!visitedMap.has(neighbor)) {

queue.push(neighbor);

return visitedMap.get(originalNode);

const currentNode = queue.shift();

public Node(int _val, ArrayList<Node> _neighbors) {

private Map<Node, Node> visited = new HashMap<>();

// This function returns the clone of the graph.

// A HashMap to keep track of all the nodes which have already been copied.

Node(int _val, std::vector<Node*> _neighbors) : val(_val), neighbors(_neighbors) {}

// If the input node is null, return null indicating no node to clone.

// If the node has been already visited, return the clone from visited.

// Record the visited node by mapping the original node to the clone.

// Recursively call cloneGraph for each neighbor and add it to the cloned node's neighbors.

// Function to clone a graph. Returns a cloned copy of the given graph node or null if the input node is null.

// A dictionary to keep track of all visited nodes and their clones.

if (visited.find(node) != visited.end()) return visited[node];

cloneNode->neighbors.push_back(cloneGraph(neighbor));

// If the original node is null, return null as there is nothing to clone.

// Remove the first node from the queue to process its neighbors.

// If this neighbor hasn't been visited/processed yet.

visitedMap.set(neighbor, new Node(neighbor.val));

clonedCurrentNode.neighbors.push(visitedMap.get(neighbor));

const clonedCurrentNode = visitedMap.get(currentNode);

// Return the cloned node that corresponds to the original input node.

// Add the cloned neighbor to the neighbors list of the cloned current node.

// Clone the neighbor and put it in the map.

// Map to hold visited nodes to avoid duplication and for constant-time look-up.

// Create a new node with the same value as the input node.

public Node() {

val = 0;

public Node(int _val) {

val = _val;

val = _val;

class Solution {

return visited[node]

clone_node = Node(node.val)

def __init__(self, val=0, neighbors=None):

def cloneGraph(self, node: 'Node') -> 'Node':

def clone(node: 'Node') -> 'Node':

self.neighbors = neighbors if neighbors is not None else []

Dictionary to keep track of visited nodes and their clones.

of the graph to create a deep copy of it.

:param node: The graph node to be cloned

Return None for a non-existent node

:return: The cloned copy of the input node

Start the graph cloning process from the input node

clone is a helper function which performs a depth-first traversal

If the node has already been visited, return the cloned node

Create a new Node clone with the value of the original node

36 # Map the original node to the cloned one in visited dictionary 37 visited[node] = clone_node 38 # For every adjacent node, create a clone copy to the neighbors of the cloned node 39 for neighbor in node.neighbors: 40 clone_node.neighbors.append(clone(neighbor))

```
Java Solution
  import java.util.HashMap;
  import java.util.List;
   import java.util.Map;
   // Definition for a graph node.
   class Node {
       public int val;
```

```
32
       public Node cloneGraph(Node node) {
33
           // If the input node is null, then return null.
34
           if (node == null) {
35
               return null;
36
37
38
           // If the node has already been visited i.e., already cloned,
           // return the cloned node from the map.
39
           if (visited.containsKey(node)) {
40
               return visited.get(node);
41
42
43
           // Create a new node with the value of the input node (clone it).
44
45
           Node cloneNode = new Node(node.val);
46
           // Mark this node as visited by putting into the visited map.
           visited.put(node, cloneNode);
47
48
49
           // Iterate over all the neighbors of the input node.
50
           for (Node neighbor: node.neighbors) {
               // Perform a depth-first search (DFS) for each neighbor,
51
52
               // and add the clone of the neighbor to the neighbors list
53
               // of the clone node.
54
               cloneNode.neighbors.add(cloneGraph(neighbor));
55
56
57
           // Return the cloned graph node.
58
           return cloneNode;
59
60 }
61
C++ Solution
1 #include <unordered_map>
2 #include <vector>
   // Forward declaration of the Node class to use it in the Solution.
  class Node {
  public:
       int val;
       std::vector<Node*> neighbors;
       Node() : val(0) {}
```

9 // Clone the original node and store the mapping. visitedMap.set(originalNode, new Node(originalNode.val)); 10 11 12 // Queue for BFS traversal starting at the original node. const queue: Node[] = [originalNode]; 13 14

while (queue.length > 0) {

Typescript Solution

```
<u>Time and Space Complexity</u>
Time Complexity
The time complexity of the cloneGraph function is O(N + M), where N is the number of nodes in the graph and M is the number of
edges. This is because each node is visited exactly once during the clone operation. When visiting each node, all of its neighbors are
```

iterated through to copy their references, contributing to the M term in the complexity. **Space Complexity** The space complexity of the cloneGraph function is also O(N), where N is the number of nodes in the graph. This space is used to

store cloned nodes in the visited dictionary to keep track of already cloned nodes, ensuring nodes are not cloned multiple times. The space complexity also includes the recursion stack, which in case of a deep graph could be O(N) in the worst case (when the graph is a linked list). However, in the average case of a typical graph, the recursion stack will not grow linearly with the number of nodes and will be smaller.