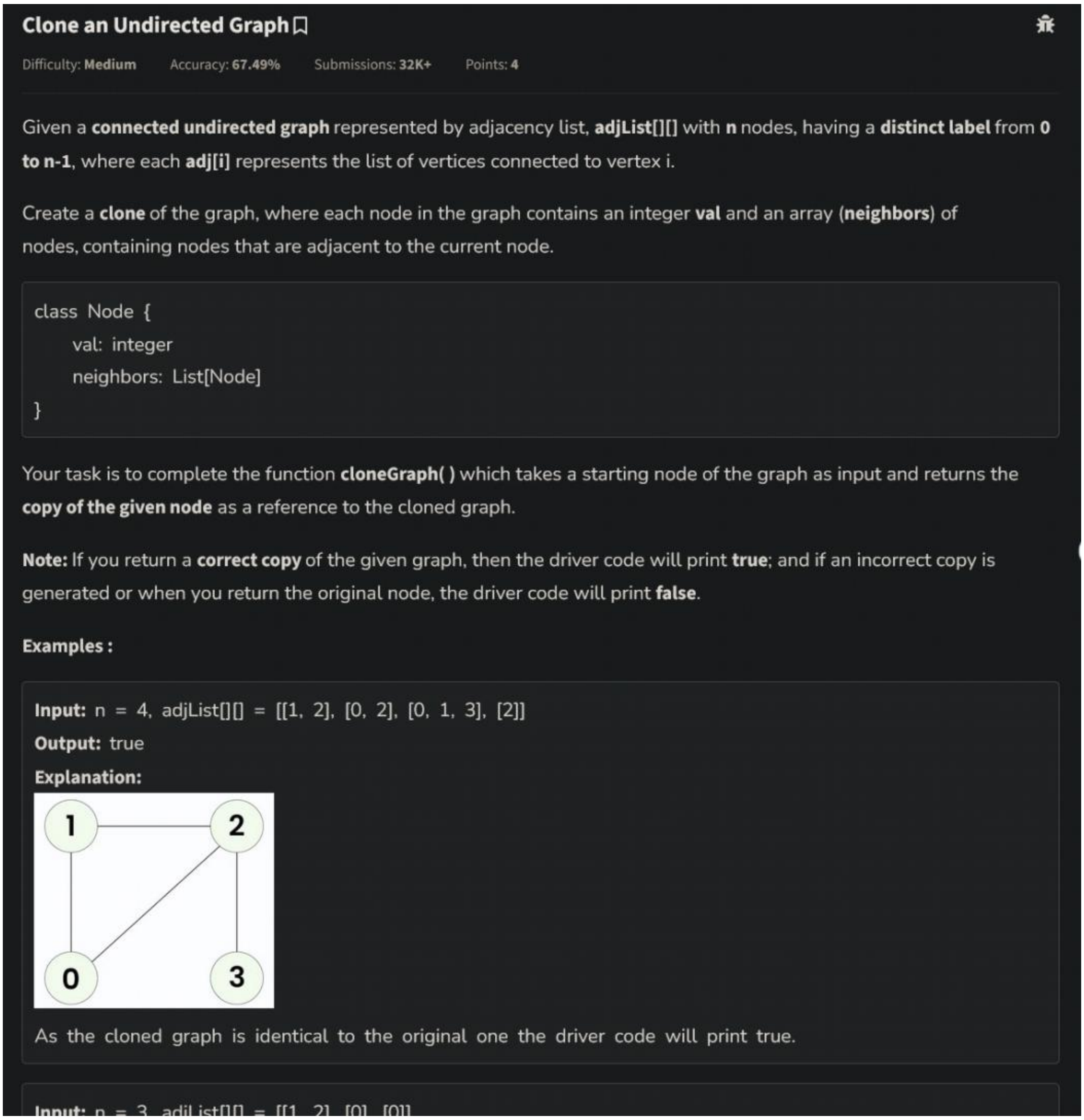
Clone an Undirected Graph:-



Clone an Undirected Graph

Difficulty: **Medium** Accuracy: **67.49%** Submissions: **32K+** Points: **4**

Given a **connected undirected graph** represented by adjacency list, **adjList[][]** with **n** nodes, having a **distinct label** from **0 to n-1**, where each **adj[i]** represents the list of vertices connected to vertex i.

Create a **clone** of the graph, where each node in the graph contains an integer **val** and an array (**neighbors**) of nodes, containing nodes that are adjacent to the current node.

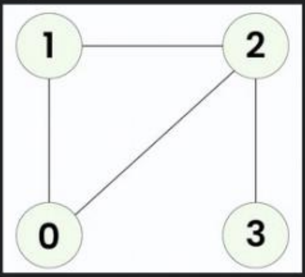
```
class Node {
    val: integer
    neighbors: List[Node]
}
```

Your task is to complete the function **cloneGraph( )** which takes a starting node of the graph as input and returns the **copy of the given node** as a reference to the cloned graph.

Note: If you return a **correct copy** of the given graph, then the driver code will print **true**; and if an incorrect copy is generated or when you return the original node, the driver code will print **false**.

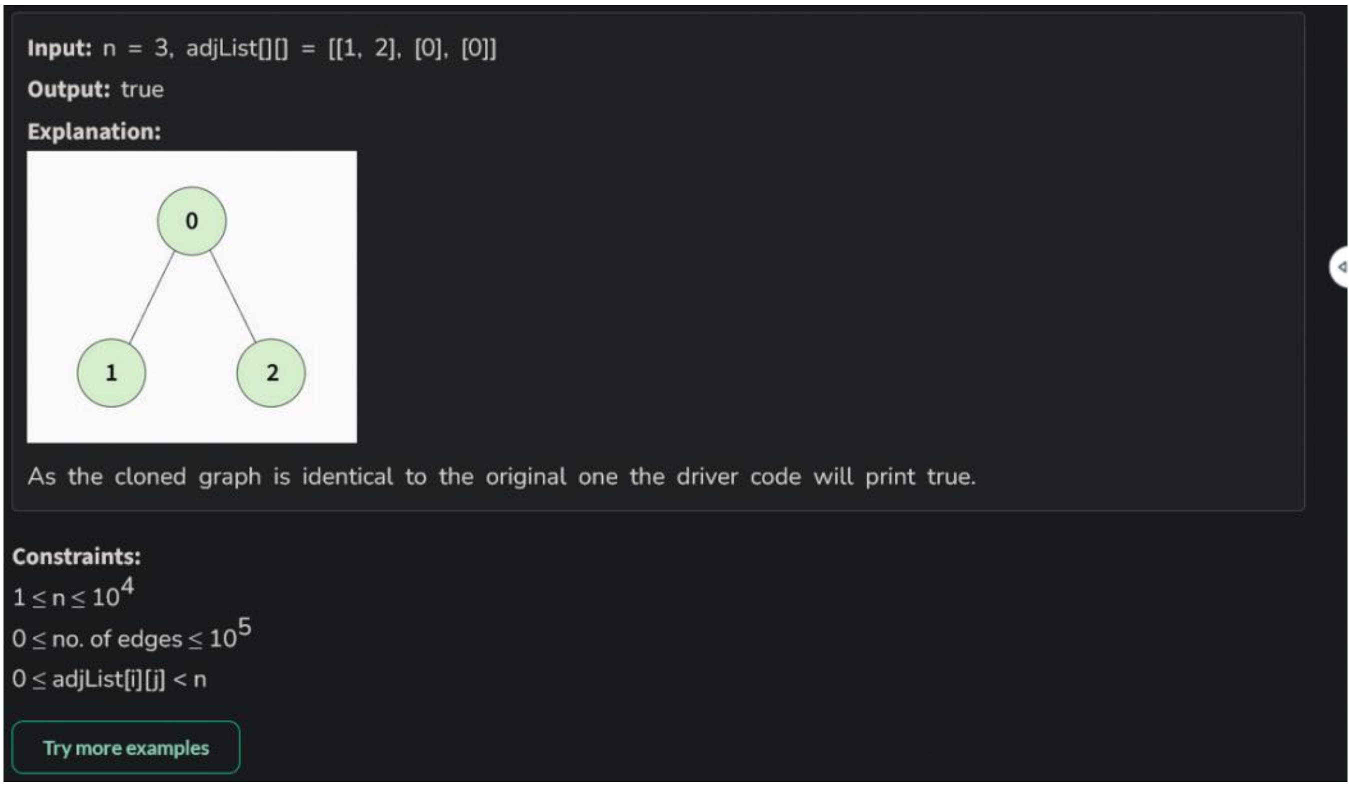
Examples :

Input: n = 4, adjList[][] = [[1, 2], [0, 2], [0, 1, 3], [2]]
Output: true
Explanation:



As the cloned graph is identical to the original one the driver code will print true.

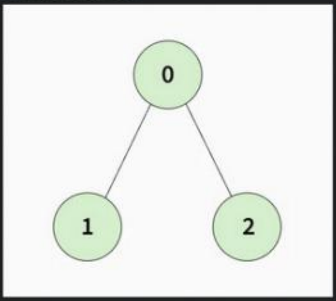
Input: n = 3, adjList[][] = [[1, 2], [0], [0]]



Input: n = 3, adjList[][] = [[1, 2], [0], [0]]

Output: true

Explanation:



As the cloned graph is identical to the original one the driver code will print true.

Constraints:

$1 \leq n \leq 10^4$

$0 \leq$ no. of edges $\leq 10^5$

$0 \leq adjList[i][j] < n$

Q) <u>why we need to track of the visited/ cloned nodes?</u>

We need to track visited or cloned nodes to avoid infinite recursion & redundant work when cloning a graph. Since graphs can contain cycles (where a node can point back to a previously visited node),

without keeping track of the nodes we've already cloned, the cloning function would endlessly revisit the same nodes, resulting in a stack overflow or incorrect u

in a stack overflow or incorrect duplication.

Q) <u>How to keep track of the visited/cloned nodes?</u>

A HashMap/Map is required in order to maintain all the nodes which have already been created. <u>Key stores</u>: Reference/Address of original Node

<u>Value stores</u>: Reference/Address of a cloned Node.

Q) <u>How to connect clone nodes?</u>

While visiting the neighboring vertices of a node u, get the corresponding cloned node for u, let's call that v, now visit all the neighboring nodes for u & for each neighbor find the corresponding cloned node (if not found create one) & then push node (if not found create one) & then push into the neighboring vector of U node.

Q) How to verify if the cloned graph is correct?

Perform a BFS traversal on the original graph before cloning, & then again on the cloned graph after cloning is complete. During each traversal, print the value of each node along with its address (or referenal).

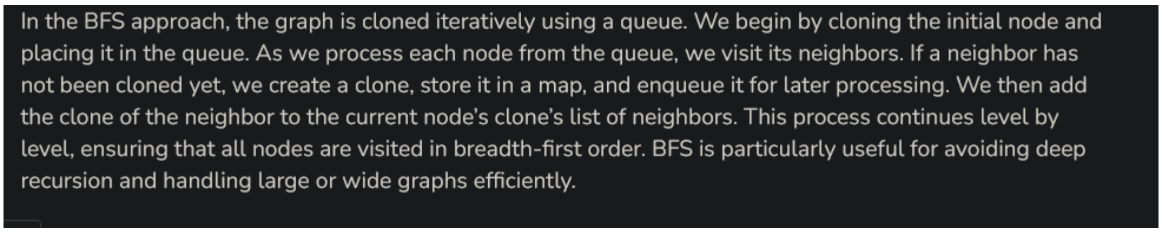
To verify the correctness of the cloning, compare the order of nodes visited in both traversal. If the nodes values appear in the same order but their addresses (or references) differ, it confirms that the graph has been successfully cloned & correctly.

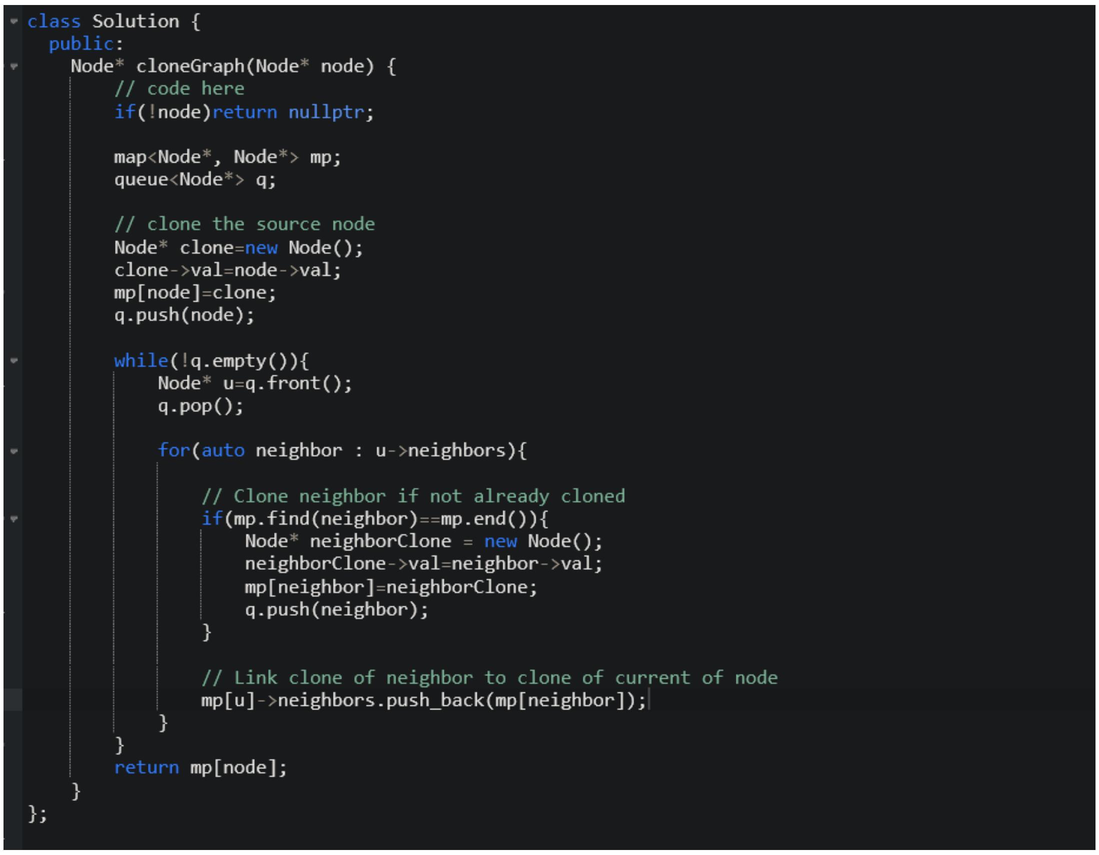
[Approach 1] Using BFS traversal - O(V+E) Time

& O(V) Space :-

[illegible]

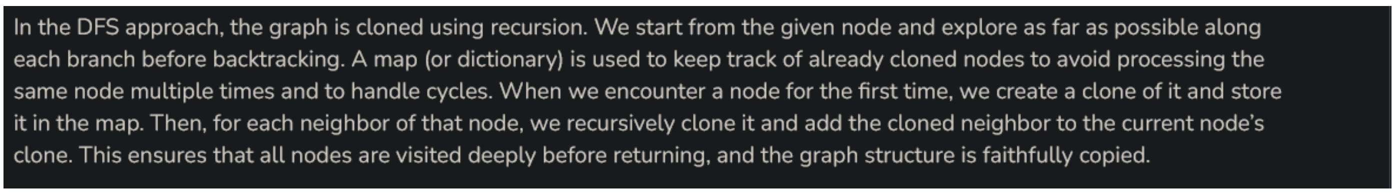
& O(V) Space :-


In the BFS approach, the graph is cloned iteratively using a queue. We begin by cloning the initial node and placing it in the queue. As we process each node from the queue, we visit its neighbors. If a neighbor has not been cloned yet, we create a clone, store it in a map, and enqueue it for later processing. We then add the clone of the neighbor to the current node's clone's list of neighbors. This process continues level by level, ensuring that all nodes are visited in breadth-first order. BFS is particularly useful for avoiding deep recursion and handling large or wide graphs efficiently.


```
class Solution {
  public:
    Node* cloneGraph(Node* node) {
        // code here
        if(!node)return nullptr;

        map<Node*, Node*> mp;
        queue<Node*> q;

        // clone the source node
        Node* clone=new Node();
        clone->val=node->val;
        mp[node]=clone;
        q.push(node);

        while(!q.empty()){
            Node* u=q.front();
            q.pop();

            for(auto neighbor : u->neighbors){

                // Clone neighbor if not already cloned
                if(mp.find(neighbor)==mp.end()){
                    Node* neighborClone = new Node();
                    neighborClone->val=neighbor->val;
                    mp[neighbor]=neighborClone;
                    q.push(neighbor);
                }

                // Link clone of neighbor to clone of current of node
                mp[u]->neighbors.push_back(mp[neighbor]);
            }
        }
        return mp[node];
    }
};
```

[Approach 2] Using DFS traversal - O(V+E) Time &

O(V) Space :-


In the DFS approach, the graph is cloned using recursion. We start from the given node and explore as far as possible along each branch before backtracking. A map (or dictionary) is used to keep track of already cloned nodes to avoid processing the same node multiple times and to handle cycles. When we encounter a node for the first time, we create a clone of it and store it in the map. Then, for each neighbor of that node, we recursively clone it and add the cloned neighbor to the current node's clone. This ensures that all nodes are visited deeply before returning, and the graph structure is faithfully copied.

```
// Map to hold original node to its copy
unordered map <Node*, Node*> copies;
```

```
unordered_map <Node*, Node*> copies;

//Function to clone the graph
Node* cloneGraph (Node* node){
    // If the node is NULL, return NULL
    if(!node) return NULL;
   // If node is not yet cloned, clone it
   if(copies.find(node)== copies.end()){
       Node* clone = new Node() ;
       clone -> val = node ->val;
       copies[node]= clone;

       //Recursively clone neighbors
     for(Node* neighbor : node -> neighbors){
   clone -> neighbors.push_back(cloneGraph
                                    (neighbor));
    }}


   // Return the clone
return copies[node];
}
```