# Problem 133: Clone Graph

## Problem Information

**Difficulty:** Medium
**Acceptance Rate:** 0.00%
**Paid Only:** No

## Problem Description

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

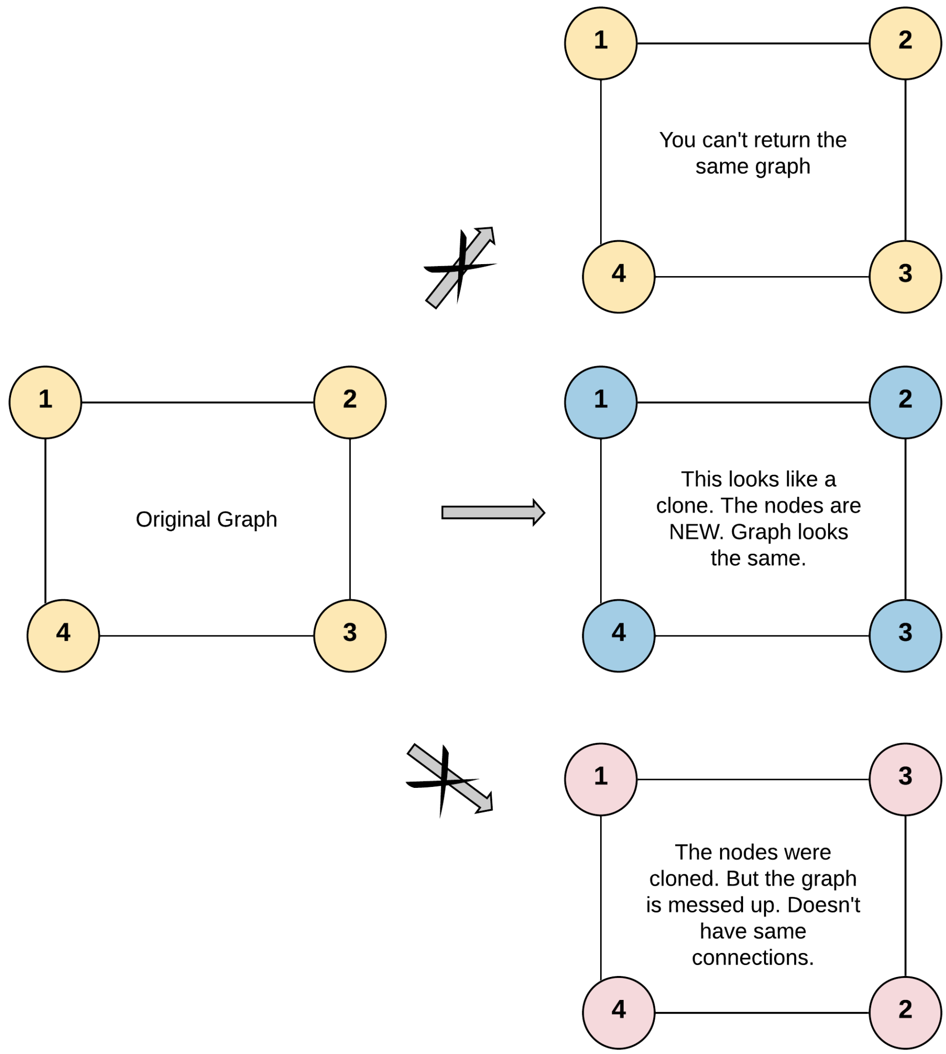
val = 1

. You must return the

copy of the given node

as a reference to the cloned graph.

Example 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 node 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.

Constraints:

The number of nodes in the graph is in the range

[0, 100]

.

1 <= Node.val <= 100

Node.val

is unique for each node.

There are no repeated edges and no self-loops in the graph.

The Graph is connected and all nodes can be visited starting from the given node.

## Code Snippets

**C++:**

```
/*
// Definition for a Node.
class Node {
public:
int val;
vector<Node*> neighbors;
Node() {
val = 0;
neighbors = vector<Node*>();
}
Node(int _val) {
val = _val;
neighbors = vector<Node*>();
}
Node(int _val, vector<Node*> _neighbors) {
val = _val;
neighbors = _neighbors;
}
};
*/
```

```
class Solution {
public:
Node* cloneGraph(Node* node) {

}
};
```

**Java:**

```
/*
// Definition for a 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;
}
}
*/

class Solution {
public Node cloneGraph(Node node) {

}
}
```

**Python3:**

```
"""
# 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']:
```

**Python:**

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

class Solution(object):
def cloneGraph(self, node):
"""
:type node: Node
:rtype: Node
"""
```

**JavaScript:**

```
/**
* // Definition for a _Node.
* function _Node(val, neighbors) {
* this.val = val === undefined ? 0 : val;
* this.neighbors = neighbors === undefined ? [] : neighbors;
* };
*/

/**
* @param {_Node} node
* @return {_Node}
*/
var cloneGraph = function(node) {

};
```

**TypeScript:**

```
/**
* Definition for _Node.
* class _Node {
* val: number
* neighbors: _Node[]
*
* constructor(val?: number, neighbors?: _Node[]) {
* this.val = (val===undefined ? 0 : val)
* this.neighbors = (neighbors===undefined ? [] : neighbors)
* }
* }
*
*/


function cloneGraph(node: _Node | null): _Node | null {

};
```

**C#:**

```
/*
// Definition for a Node.
public class Node {
public int val;
public IList<Node> neighbors;

public Node() {
val = 0;
neighbors = new List<Node>();
}

public Node(int _val) {
val = _val;
neighbors = new List<Node>();
}

public Node(int _val, List<Node> _neighbors) {
val = _val;
neighbors = _neighbors;
```

```
}
}
*/

public class Solution {
public Node CloneGraph(Node node) {

}
}
```

**C:**

```
/**
* Definition for a Node.
* struct Node {
* int val;
* int numNeighbors;
* struct Node** neighbors;
* };
*/

struct Node *cloneGraph(struct Node *s) {

}
```

**Go:**

```
/**
* Definition for a Node.
* type Node struct {
* Val int
* Neighbors []*Node
* }
*/

func cloneGraph(node *Node) *Node {

}
```

**Kotlin:**

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var neighbors: ArrayList<Node?> = ArrayList<Node?>()
* }
*/

class Solution {
fun cloneGraph(node: Node?): Node? {

}
}
```

**Swift:**

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var neighbors: [Node?]
* public init(_ val: Int) {
* self.val = val
* self.neighbors = []
* }
* }
*/

class Solution {
func cloneGraph(_ node: Node?) -> Node? {

}
}
```

**Ruby:**

```
# Definition for a Node.
# class Node
# attr_accessor :val, :neighbors
# def initialize(val = 0, neighbors = nil)
# @val = val
# neighbors = [] if neighbors.nil?
# @neighbors = neighbors
# end
```

```
# end

# @param {Node} node
# @return {Node}
def cloneGraph(node)

end
```

**PHP:**

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $neighbors = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->neighbors = array();
* }
* }
*/

class Solution {
/**
* @param Node $node
* @return Node
*/
function cloneGraph($node) {

}
}
```

**Scala:**

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var neighbors: List[Node] = List()
* }
*/
```

```
object Solution {
def cloneGraph(graph: Node): Node = {

}
}
```

## Solutions

### C++ Solution:

```
/*
* Problem: Clone Graph
* Difficulty: Medium
* Tags: graph, hash, search
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/*
// Definition for a Node.
class Node {
public:
int val;
vector<Node*> neighbors;
Node() {
val = 0;
neighbors = vector<Node*>();
}
Node(int _val) {
val = _val;
neighbors = vector<Node*>();
}
Node(int _val, vector<Node*> _neighbors) {
val = _val;
neighbors = _neighbors;
}
};
*/
```

```
class Solution {
public:
Node* cloneGraph(Node* node) {

}
};
```

### Java Solution:

```
/**
* Problem: Clone Graph
* Difficulty: Medium
* Tags: graph, hash, search
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/*
// Definition for a 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;
}
}
*/

class Solution {
```

```
public Node cloneGraph(Node node) {

}
}
```

**Python3 Solution:**

```
"""
Problem: Clone Graph
Difficulty: Medium
Tags: graph, hash, search

Approach: Use hash map for O(1) lookups
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(n) for hash map
"""

"""
# 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']:
# TODO: Implement optimized solution
pass
```

**Python Solution:**

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

```
class Solution(object):
def cloneGraph(self, node):
"""
:type node: Node
:rtype: Node
"""
```

### JavaScript Solution:

```
/**
* Problem: Clone Graph
* Difficulty: Medium
* Tags: graph, hash, search
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/**
* // Definition for a _Node.
* function _Node(val, neighbors) {
* this.val = val === undefined ? 0 : val;
* this.neighbors = neighbors === undefined ? [] : neighbors;
* };
*/

/**
* @param {_Node} node
* @return {_Node}
*/
var cloneGraph = function(node) {

};
```

### TypeScript Solution:

```
/**
* Problem: Clone Graph
* Difficulty: Medium
* Tags: graph, hash, search
```

```
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/**
* Definition for _Node.
* class _Node {
* val: number
* neighbors: _Node[]
*
* constructor(val?: number, neighbors?: _Node[]) {
* this.val = (val===undefined ? 0 : val)
* this.neighbors = (neighbors===undefined ? [] : neighbors)
* }
* }
*
*/



function cloneGraph(node: _Node | null): _Node | null {

};
```

### C# Solution:

```
/*
* Problem: Clone Graph
* Difficulty: Medium
* Tags: graph, hash, search
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/*
// Definition for a Node.
public class Node {
public int val;
```

```
public IList<Node> neighbors;

public Node() {
val = 0;
neighbors = new List<Node>();
}

public Node(int _val) {
val = _val;
neighbors = new List<Node>();
}

public Node(int _val, List<Node> _neighbors) {
val = _val;
neighbors = _neighbors;
}
}
*/

public class Solution {
public Node CloneGraph(Node node) {

}
}
```

### C Solution:

```
/*
* Problem: Clone Graph
* Difficulty: Medium
* Tags: graph, hash, search
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/**
* Definition for a Node.
* struct Node {
* int val;
```

```
* int numNeighbors;
* struct Node** neighbors;
* };
*/

struct Node *cloneGraph(struct Node *s) {

}
```

### Go Solution:

```
// Problem: Clone Graph
// Difficulty: Medium
// Tags: graph, hash, search
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

/**
* Definition for a Node.
* type Node struct {
* Val int
* Neighbors []*Node
* }
*/

func cloneGraph(node *Node) *Node {

}
```

### Kotlin Solution:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var neighbors: ArrayList<Node?> = ArrayList<Node?>()
* }
*/

class Solution {
```

```
fun cloneGraph(node: Node?): Node? {

}
}
```

**Swift Solution:**

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var neighbors: [Node?]
* public init(_ val: Int) {
* self.val = val
* self.neighbors = []
* }
* }
*/

class Solution {
func cloneGraph(_ node: Node?) -> Node? {

}
}
```

**Ruby Solution:**

```
# Definition for a Node.
# class Node
# attr_accessor :val, :neighbors
# def initialize(val = 0, neighbors = nil)
# @val = val
# neighbors = [] if neighbors.nil?
# @neighbors = neighbors
# end
# end

# @param {Node} node
# @return {Node}
def cloneGraph(node)
```

```
end
```

### PHP Solution:

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $neighbors = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->neighbors = array();
* }
* }
*/

class Solution {
/**
* @param Node $node
* @return Node
*/
function cloneGraph($node) {

}
}
```

### Scala Solution:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var neighbors: List[Node] = List()
* }
*/

object Solution {
def cloneGraph(graph: Node): Node = {

}
}
```