Description

133. Clone Graph

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

OJ's undirected graph serialization:
Nodes are labeled uniquely.

We use # as a separator for each node, and , as a separator for node label and each neighbor of the node.
As an example, consider the serialized graph {0,1,2#1,2#2,2}.

The graph has a total of three nodes, and therefore contains three parts as separated by #.

First node is labeled as 0. Connect node 0 to both nodes 1 and 2.
Second node is labeled as 1. Connect node 1 to node 2.
Third node is labeled as 2. Connect node 2 to node 2 (itself), thus forming a self-cycle.
Visually, the graph looks like the following:

```
       1
      / \
     /   \
    0 --- 2
         / \
         \_/
```

Idea Report

To clone a graph, we can clone all the nodes while traverse the graph, then connecting all the edges. When connecting edges, we need a old node to new node map to find out what edges to connect. So first step is to clone all the nodes using BFS while store all the old-node-to-new-node map information. Then we traverse this map and connect all the new nodes neighbors according to its corresponding old node's neighbors.

Java

```java
public class Solution {
    // BFS 2 pass
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null) {
            return node;
        }

        Map<UndirectedGraphNode, UndirectedGraphNode> map = initMap(node);
        for (Map.Entry<UndirectedGraphNode, UndirectedGraphNode> entry
                : map.entrySet()) {
            for (UndirectedGraphNode nei : entry.getKey().neighbors) {
                entry.getValue().neighbors.add(map.get(nei));
            }
        }
        return map.get(node);
    }

    private Map<UndirectedGraphNode, UndirectedGraphNode> initMap(
                UndirectedGraphNode node) {
        Map<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<>();
        Deque<UndirectedGraphNode> q = new ArrayDeque<>();
        q.offer(node);
        map.put(node, new UndirectedGraphNode(node.label));

        while (!q.isEmpty()) {
            for (UndirectedGraphNode nei : q.poll().neighbors) {
                if (!map.containsKey(nei)) {
                    map.put(nei, new UndirectedGraphNode(nei.label));
                    q.offer(nei);
                }
            }
        }

        return map;
    }
}
```

We can also connecting all the neighbors while traversing the nodes, if a neighbor node is not yet cloned, we clone it first and connecting the edge.

Java

```java
public class Solution {
    // BFS 1 pass
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null) {
            return node;
        }

        Map<Integer, UndirectedGraphNode> map = new HashMap<>();
        map.put(node.label, new UndirectedGraphNode(node.label));
        Deque<UndirectedGraphNode> q = new ArrayDeque<>();
        q.offer(node);

        while (!q.isEmpty()) {
            UndirectedGraphNode cur = q.poll();
            for (UndirectedGraphNode nei : cur.neighbors) {
                if (!map.containsKey(nei.label)) {
                    map.put(nei.label, new UndirectedGraphNode(nei.label));
                    q.offer(nei);
                }
                map.get(cur.label).neighbors.add(map.get(nei.label));
            }
        }

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

We can also do it recursively with DFS. When connecting the neighbor nodes, we just connect it to the cloned new node recursively (neighbors.add(cloneGraph(nei))).

Java

```java
public class Solution {
    // DFS
    Map<Integer, UndirectedGraphNode> map = new HashMap<>();
    public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {
        if (node == null) {
            return node;
        }

        if (map.containsKey(node.label)) {
            return map.get(node.label);
        }

        map.put(node.label, new UndirectedGraphNode(node.label));
        for (UndirectedGraphNode nei : node.neighbors) {
            map.get(node.label).neighbors.add(cloneGraph(nei));
        }

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

Summary

- Basically a graph traverse.
- Use map to store old-node-to-new-node relation.
- Since we traverse the graph, the time complexity is O(V+E), vertices + edges
- Space complexity is O(V), the queue's largest size or deepest call stack if DFS.

C++ implementation

```
class Solution {
public:
    // BFS 2 pass
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
        if (node == NULL) {
            return node;
        }

        unordered_map<UndirectedGraphNode*, UndirectedGraphNode*> map
            = initMap(node);
        for (auto it = map.begin(); it != map.end(); it++) {
            for (UndirectedGraphNode* nei : it->first->neighbors) {
                it->second->neighbors.push_back(map[nei]);
            }
        }

        return map[node];
    }

    unordered_map<UndirectedGraphNode*, UndirectedGraphNode*> initMap(
            UndirectedGraphNode* node) {
        unordered_map<UndirectedGraphNode*, UndirectedGraphNode*> map;
        map[node] = new UndirectedGraphNode(node->label);
        deque<UndirectedGraphNode*> q;
        q.push_back(node);

        while (!q.empty()) {
            UndirectedGraphNode* cur = q.front();
            q.pop_front();
            for (UndirectedGraphNode* nei : cur->neighbors) {
                if (map.find(nei) == map.end()) {
                    map[nei] = new UndirectedGraphNode(nei->label);
                    q.push_back(nei);
                }
            }
        }

        return map;
    }
};
```

```
class Solution {
public:
    // BFS 1 pass
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
        if (node == NULL) {
            return node;
        }

        unordered_map<int, UndirectedGraphNode*> map;
        map[node->label] = new UndirectedGraphNode(node->label);
        deque<UndirectedGraphNode*> q;
        q.push_back(node);

        while (!q.empty()) {
            UndirectedGraphNode* cur = q.front();
            q.pop_front();
            for (UndirectedGraphNode* nei : cur->neighbors) {
                if (map.find(nei->label) == map.end()) {
                    map[nei->label] = new UndirectedGraphNode(nei->label);
                    q.push_back(nei);
                }
                map[cur->label]->neighbors.push_back(map[nei->label]);
            }
        }

        return map[node->label];
    }
};
```

```
class Solution {
public:
    // DFS
    unordered_map<int, UndirectedGraphNode*> map;
    UndirectedGraphNode *cloneGraph(UndirectedGraphNode *node) {
        if (node == NULL) {
            return node;
        }

        if (map.find(node->label) != map.end()) {
            return map[node->label];
        }

        map[node->label] = new UndirectedGraphNode(node->label);
        for (UndirectedGraphNode* nei : node->neighbors) {
            map[node->label]->neighbors.push_back(cloneGraph(nei));
        }

        return map[node->label];
    }
};
```