851. Loud and Rich

Medium | Depth-First Search | Graph | Topological Sort | Array — Leetcode Link

Problem Description

In this LeetCode problem, we are tasked with finding the least quiet person among a group of people, given their relative wealth and their respective levels of quietness. Each person in the group is identified by a unique number from `0` to `n - 1`.

An array `richer` is provided, where each element `richer[i]` is a pair `[a, b]` that indicates person `a` is wealthier than person `b`. Then there is an array `quiet` where `quiet[i]` represents the level of quietness for person `i`, with a smaller value indicating a quieter person.

Our goal is to return an array `answer` where `answer[x]` is the index `y` of the person who is the least quiet among all those who are as wealthy or wealthier than person `x`. Here, 'as wealthy or wealthier' means they are either directly richer than person `x`, or richer than someone who is richer than person `x`, and so on.

Intuition

To solve this problem, we use a Depth-First Search (DFS) approach. The key observation in this problem is that the relationships between people's wealth create a directed graph without cycles because the richer relations are logically consistent. In this graph, nodes represent people, and an edge from node `a` to node `b` indicates that `a` is wealthier than `b`. Therefore, when we look for the least quiet person who is as rich or richer than a person `x`, we are actually looking for the least quiet person in the subgraph reachable from node `x`.

The solution involves the following steps:

1. Convert the `richer` array into a graph data structure to allow for efficient traversal. This graph is represented using an adjacency list.
2. Initialize an array `answer` with all elements set to `-1`, which will help us track the least quiet person as we traverse the graph as well as serve as a cache to avoid repeated calculations.
3. Perform a DFS from each person. During the search, if we have already computed the result for a node (person), we return immediately to save time.
4. When we visit a node, we compare its quietness with that of the already recorded least quiet person (if any). If the current node is quieter, we update the record in the `answer` array.
5. We recursively visit all richer people than the current person and apply the same logic.
6. After finishing the dfs, the `answer` array will contain the desired results based on the computed least quiet person reachable from each node.

The recursive DFS and caching technique (also known as memoization) helps us avoid re-examining nodes multiple times, leading to efficient calculation of the answer.

Solution Approach

The solution uses Depth-First Search (DFS) to traverse the graph built from the `richer` array, leveraging recursion to navigate through the nodes (people) and explore their wealth relationships. Here is a breakdown of how the implementation works:

- A graph `g` is initialized as a default dictionary of lists, which will hold the adjacency list representation of our directed graph. For every pair `[a, b]` in `richer`, we add `a` to the adjacency list of `b` because `a` is richer than `b`. In this analogy, `b` is the starting node, and `a` is reachable from `b`.

- We create an array `ans` initialized with `-1`s to hold the index of the quietest person richer or equally wealthy as the person at index `i`. This array also helps in caching the results of previous calculations for each person to avoid repeating the expensive DFS operation.

- A `dfs` function is defined, taking a person index `i` as its argument. The purpose of this function is to find and record the quietest person that can be reached from the person `i`. If `ans[i]` is not `-1`, it means that we have already computed the result for person `i`, and there's no need to repeat the computation.

- For the current person `i`, the `dfs` function sets `ans[i]` to `i` itself initially, assuming the person is the quietest among all people wealthier than or as wealthy as them.

- The function then iterates through all people that are richer than person `i` (all `j` in `g[i]`) and performs a recursive DFS call for each of them. After exploring each wealthier person `j`, it compares the quietness of the current quietest person `ans[i]` with the quietest person in the subgraph starting from `j` (`ans[j]`). If a quieter person is found (`quiet[ans[j]] < quiet[ans[i]]`), `ans[i]` is updated to refer to that person.

- The main loop at the end iterates through all the people, calling `dfs(i)` for each person `i`. This step ensures that even if some people are not directly richer than others, they are still explored through the recursive DFS calls.

The use of DFS in this solution is a powerful choice, as it allows us to explore the complete subset of people that are more prosperous than a given person, caching results along the way to prevent redundant calculations. The use of memoization with the `ans` array helps to cut down the execution time significantly, as the DFS will only compute the quietest person for each subgraph once. After the DFS for all people has been performed, the `ans` array contains the quietest person among all richer people for each person in the group, effectively solving the problem.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we have the following input:

- `richer = [[1, 0], [2, 1], [3, 1]]`: This means person 1 is richer than person 0, person 2 is richer than person 1, and person 3 is richer than person 1.
- `quiet = [3, 2, 5, 4]`: This indicates the level of quietness for each person. Person 0 has quietness 3, person 1 has 2, person 2 has 5 and person 3 has 4.

Now, let's walk through the steps:

1. Convert `richer` into a graph data structure as an adjacency list:

   ```
   g[0] = []
   g[1] = [0]
   g[2] = [1]
   g[3] = [1]
   ```

   Here, `g[i]` contains the people that are less wealthy than `i`.

2. Initialize an array `ans = [-1, -1, -1, -1]` to indicate that no answers have been computed yet.

3. Perform DFS for each person to find the least quiet person that is as wealthy or wealthier than them:

   - Start with person 0. Since no one is richer, `ans[0]` will be `0` (the person itself).
   - DFS on person 1: Recursively check for richer people and update `ans[1]`. Person 0 is not richer, so `ans[1]` will be `1`.
   - DFS on person 2: The richer people are persons 1 and 0. We check both, and since person 1 is quieter (`2<5`), we update `ans[2]` with `1`.
   - DFS on person 3: The richer people are persons 1 and 0. Person 3 is quieter than person 1, so `ans[3]` is set to `3`.
4. After performing DFS for each person, `ans` is updated with the least quiet richer people: `ans = [0, 1, 1, 3]`.

To conclude, given our `richer` and `quiet` inputs, the least quiet person who is as wealthy or wealthier than person `x` for `x` in `0` to `n-1` has been effectively found using our DFS approach and is represented in the `ans` array.

Python Solution

```python
from collections import defaultdict

class Solution:
    def loudAndRich(self, richer, quiet):
        # This method finds the quietest person in the group or among their richer acquaintances.

        def dfs(index):
            # Depth-first search to update the answer for each person.
            if answer[index] != -1:
                # If this person's answer is already calculated, return.
                return
            # Otherwise, initialize this person's answer as themselves.
            answer[index] = index
            for neighbor in graph[index]:
                # Visit all richer neighbors.
                dfs(neighbor)
                # If the neighbor has found someone quieter, update this person's answer.
                if quiet[answer[neighbor]] < quiet[answer[index]]:
                    answer[index] = answer[neighbor]

        # Build the graph given the richer relationship.
        graph = defaultdict(list)
        for richer_person, poorer_person in richer:
            graph[poorer_person].append(richer_person)

        # Initialize the answer list with -1 for all persons.
        n = len(quiet)
        answer = [-1] * n

        # Perform dfs for each person.
        for i in range(n):
            dfs(i)

        # Return the completed answer list.
        return answer

```

Java Solution

```java
class Solution {
    private List<Integer>[] graph; // Graph adjacency list
    private int peopleCount; // Number of people in the problem
    private int[] quietness; // Array representing the quietness of each person
    private int[] answer; // Array representing the answer of the quietest person who is richer or as rich

    // The main function that takes richer relations and quietness indices, and returns an array of answers
    public int[] loudAndRich(int[][] richer, int[] quiet) {
        peopleCount = quiet.length; // Set the number of people based on the quiet array length
        this.quietness = quiet; // Set the quietness array
        graph = new List[peopleCount]; // Initialize the graph with peopleCount vertices
        answer = new int[peopleCount]; // Initialize the answer array with peopleCount elements
        Arrays.fill(answer, -1); // Initially fill the answer array with -1 indicating not found
        Arrays.setAll(graph, k -> new ArrayList<>()); // Initialize each vertex's adjacency list

        // Build the graph's adjacency list from the richer array
        for (var r : richer) {
            graph[r[1]].add(r[0]); // r[1] person is poorer so we add r[0] as an outgoing edge from r[1]
        }

        // Perform DFS starting from every vertex to find the answer for each person
        for (int i = 0; i < peopleCount; ++i) {
            dfs(i);
        }
        return answer; // Return the filled answer array
    }

    // Recursive Depth-First Search function that populates the answer array
    private void dfs(int i) {
        if (answer[i] != -1) {
            // If the quietest person for i is already found, terminate the DFS
            return;
        }

        // Set the default quietest person to the person themselves
        answer[i] = i;

        // Explore all richer people coming from node i
        for (int j : graph[i]) {
            dfs(j); // DFS on the richer person j
            // If the quietest person for j is quieter than the current quietest person for i, update it
            if (quietness[answer[j]] < quietness[answer[i]]) {
                answer[i] = answer[j];
            }
        }
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <functional>
using std::vector;

class Solution {
public:
    vector<int> loudAndRich(vector<vector<int>>& richer, vector<int>& quiet) {
        int numPeople = quiet.size(); // The number of people
        vector<vector<int>> graph(numPeople); // Graph to hold richer relationships
        // Build the graph with directed edges from richer to poorer
        for (const auto& pair: richer) {
            graph[pair[1]].push_back(pair[0]);
        }
        // Answer vector with an initial value of -1 for each element
        vector<int> answer(numPeople, -1);
        // Define a lambda function for Depth First Search
        std::function<void(int)> dfs = [&](int node) {
            // If we have already computed the answer for this node, return
            if (answer[node] != -1) {
                return;
            }
            // Initially, assume the person is the quietest
            answer[node] = node;
            // Explore all the richer people
            for (int neighbor : graph[node]) {
                dfs(neighbor); // depth-first search the neighbor
                // If the neighbor has a quieter answer, update this node's answer
                if (quiet[answer[neighbor]] < quiet[answer[node]]) {
                    answer[node] = answer[neighbor];
                }
            }
        };
        // Perform a DFS from each person to find the quietest person they are richer than
        for (int i = 0; i < numPeople; ++i) {
            dfs(i);
        }
        // Return the final answer array
        return answer;
    }
};

```

Typescript Solution

```typescript
function loudAndRich(richer: number[][], quiet: number[]): number[] {
    // Total number of people
    const numPeople = quiet.length;

    // Graph representation, where g[x] contains all people richer than person x
    const graph: number[][] = new Array(numPeople).fill(0).map(() => []);

    // Construct the graph based on the richer relationships
    for (const [richerPerson, lessRichPerson] of richer) {
        graph[lessRichPerson].push(richerPerson);
    }

    // Initialize an array to hold the answer
    // Filled initially with -1 to indicate that we haven't computed the answer for that person yet
    const answer: number[] = new Array(numPeople).fill(-1);

    // Depth-First Search (DFS) function to determine the quietest person in the subgraph
    const dfs = (person: number) => {
        // If we've already computed the answer for this person, return the answer array
        if (answer[person] !== -1) {
            return answer;
        }

        // The quietest person initially being themselves
        answer[person] = person;

        // Explore all the richer people
        for (const richerPerson of graph[person]) {
            dfs(richerPerson);

            // If we find a quieter richer person, update the answer for the current person
            if (quiet[answer[richerPerson]] < quiet[answer[person]]) {
                answer[person] = answer[richerPerson];
            }
        }
    };

    // Perform DFS for each person to find the quietest person they know directly or indirectly
    for (let i = 0; i < numPeople; ++i) {
        dfs(i);
    }

    // Return the array containing the quietest person for each index
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of this code is primarily determined by the depth-first search recursion implemented through the `dfs()` function. Since each node in the graph `g` represents a person and each edge in the graph represents the "richer" relation, the depth-first search will visit each node and each edge at most once. Given that there are `n` nodes (individuals) and `m` edges (relations), the traversal has a complexity of `O(n+m)`.

However, notice that the DFS process is performed for each node in the `for i in range(n):` loop, and during each DFS execution, it processes only the nodes that have not been previously processed (`if ans[i] != -1: return`). Once a node has been processed, it will not be processed again in any subsequent DFS calls. Thus, each node triggers the DFS call at most once, and each edge is considered once across all DFS calls. Therefore, despite the outer loop suggesting `O(n^2)` behavior at first glance, the function still achieves `O(n+m)` because each node and edge is effectively processed only a single time.

Space Complexity

The space complexity of the code is `O(n)` due to multiple factors:

- `ans` array of size `n`.
- `g` dictionary, which in the worst case could have up to `n` keys with a single value in the list (if the graph is a star shape where one person is richer than all `n-1` others), so it uses `O(n)` space.
- The recursion stack for the DFS could, in the worst case, go as deep as `n` if the graph is a long chain, contributing another `O(n)` to the space complexity.

The space complexity, in this case, is determined primarily by the system's recursion stack depth and the additional data structures (`ans` and `g`), which together yield `O(n)`.