1376. Time Needed to Inform All Employees

Medium | Tree | Depth-First Search | Breadth-First Search | Leetcode Link

Problem Description

In this problem, we have a company structure that resembles a tree, where each employee except the head has a direct manager. The head of the company has a unique ID 'headID' where `manager[headID] = -1`, indicating that they have no managers above them. Every employee has a unique ID from 0 to n - 1. Each employee is responsible for passing on an urgent message to their direct subordinates, and this chain of communication continues until all employees are aware of the message. The time taken for each employee to inform their subordinates is given in an array `informTime`, where `informTime[i]` represents the minutes employee i takes to inform their direct subordinates. The goal is to find out the total number of minutes needed to inform all the employees.

The communication follows a hierarchical tree structure where the head of the company is the root. When the head of the company starts the communication, the message flows down the tree from manager to subordinate. Each employee waits until they receive the message before they start their clock and spend the specified `informTime[i]` minutes to pass the message to their subordinates. The total time taken for the entire company to be informed is the longest path of communication from the head down to any leaf in the tree.

Intuition

The solution utilizes Depth-First Search (DFS), a classic algorithm to traverse or search tree or graph data structures. The idea is to track the time taken for an employee to spread the news to all their subordinates. We simulate this propagation of information from the head of the company downwards.

First, we construct a graph that represents the company structure, with edges from managers to subordinates. This adjacency list is built from the `manager` array where each index corresponds to an employee and the value at that index is the employee's manager.

Next, we define the DFS function, which recursively calculates the total time needed by an employee to inform all their direct and indirect subordinates. The function performs the following steps:

1. It receives an employee ID `i`.
2. Iterates over all direct subordinates `j` of `i` (retrieved from the adjacency list).
3. Calls itself (DFS) for each subordinate `j` to calculate the time `j` will take to inform their subordinates.
4. Adds the `informTime[i]` to the time taken by `j` to distribute the message further, and keeps track of the maximum time encountered.
5. Returns the maximum time taken for employee `i` to inform all their subordinates.

Applying the DFS function starting from `headID` will provide us with the total time needed for the head of the company to spread the message throughout the organization. This time corresponds to the longest path from the root (head of the company) to any leaf node (employee) in this tree structure.

Solution Approach

The solution to this problem is implemented via a Depth-First Search (DFS) approach, which is a standard algorithm used to traverse trees or graphs. Here's an in-depth walkthrough of how the solution is implemented:

1. **Graph Construction**: First, we build an adjacency list `g`, using a `defaultdict` from Python's collections module. This data structure maps each manager to their list of direct subordinates. We populate this adjacency list by iterating through the `manager` array and appending the index `i` (which represents the employee) to the list of subordinates for the manager `x`.

2. **DFS Function**: We define a recursive function `dfs(i)` where `i` is the ID of the employee whose total inform time needs to be calculated.

   - If the employee `i` does not have any subordinates (i.e., a leaf node), the function returns 0 immediately because they do not need to inform anyone else.
   - If the employee has subordinates, the function iterates through the list of direct subordinates `j` in `g[i]` and recursively calls `dfs(j)` on each. This call calculates the time it takes for the subtree rooted at `j` to be fully informed.
   - We keep track of the maximum time taken among all subordinates by calculating `max(ans, dfs(j) + informTime[i])`, where `ans` is the maximum time found so far and `informTime[i]` is the time `i` takes to inform their direct subordinates.
   - After all subordinates of `i` have been processed, the function returns the maximum time taken as the complete inform time for employee `i`'s subordinates.
3. **Calculating the Answer**: Finally, the total number of minutes to inform all employees is given by `dfs(headID)`. This call starts the recursive process at the head of the company, and since we use the maximum inform time at each step, the returned value represents the longest time path from the head down to any leaf node. This value is the answer to our problem.

By utilizing the `defaultdict` to create a graph and the recursive DFS function to explore the tree structure, we obtain an efficient way to calculate the minimum time needed to distribute the news to all employees. The `max` function ensures that we consider the longest path in the tree structure, corresponding to the worst-case time scenario given the hierarchical company structure.

Example Walkthrough

Let's say we have these inputs for our company communication problem:

- The `manager` array is `[3, 3, -1, 2]`, representing that:
  - Employee 0 reports to manager 3.
  - Employee 1 also reports to manager 3.
  - Employee 2 is the head of the company as `manager[2]` is `-1`.
  - Employee 3 reports to manager 2.
- The `informTime` array is `[1, 2, 10, 5]`:
  - It takes 1 minute for employee 0 to inform their subordinates (in this case, they have none).
  - It takes 2 minutes for employee 1 to inform their subordinates (they also have none).
  - It takes 10 minutes for employee 2 (the head) to inform their subordinate (employee 3).
  - It takes 5 minutes for employee 3 to inform their subordinates (employees 0 and 1).

With the given structure, we can construct the following company tree:

```
      (Head)
    Employee 2
      /    \
     /       \
Employee 3    ---
/    \
E0    E1
```

Here is the step-by-step approach using our solution:

1. We build our adjacency list `g` from the `manager` array. After building, it will look like this:

```
{
  3: [0, 1], // Employee 3 manages Employee 0 and 1
  2: [3],    // The head (Employee 2) manages Employee 3
}
```

2. We define and use our DFS function to calculate the total inform time. We begin with `dfs(2)` as 2 is the head:
   - Since employee 2 is not a leaf node, we check for its subordinates.
   - We find that employee 2 has a subordinate, employee 3, and we call `dfs(3)`.
   - Now within `dfs(3)`, since employee 3 also has subordinates (0 and 1), we call `dfs` on each.
   - `dfs(0)` returns 0 because employee 0 is a leaf node.
   - Similarly, `dfs(1)` returns 0.
   - Now, with both subordinates checked, `dfs(3)` returns `max(0+5, 0+5)` (since both subordinates took 0 minutes and employee 3 needs 5 minutes to inform).
   - Back to `dfs(2)`, it now returns `max(10, 5+10)`, which is 15, since employee 2 takes 10 minutes to inform their direct subordinate (employee 3) and then employee 3 takes another 5 minutes, totaling to 15.
3. `dfs(headID)` sums up the total time taken to the head of the company to distribute the message:
   - Since `headID` is 2, we return `dfs(2)` which we have calculated to be 15.
   - Thus, the total number of minutes to inform all employees is 15.

In this example, the longest path of communication, which determines the total inform time, is from the head (Employee 2) to Employee 3, and then to either Employee 0 or Employee 1, totaling 15 minutes.

Python Solution

```python
from typing import List
from collections import defaultdict

class Solution:
    def numOfMinutes(self, n: int, head_id: int, managers: List[int], inform_time: List[int]) -> int:
        # Recursive depth-first search function to calculate the time taken to inform each employee.
        def dfs(employee_id: int) -> int:
            max_time = 0
            # Iterate over each subordinate of the current employee.
            for subordinate in graph[employee_id]:
                # Recursively call dfs for the subordinate and add the current employee's inform time.
                # We want the maximum time required for all subordinates as they can be informed in parallel.
                max_time = max(max_time, dfs(subordinate) + inform_time[employee_id])
            return max_time

        # Create a graph where each employee is a node and the edges are from managers to subordinates.
        graph = defaultdict(list)
        for i, mng_id in enumerate(managers):
            # Skip the head of the company as they have no manager.
            if mng_id != -1:
                graph[mng_id].append(i)

        # Kick off the dfs from the head of the company to calculate the total time required.
        return dfs(head_id)

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {

    // Graph representation where each node is an employee and edges point to subordinates
    private List<Integer>[] graph;

    // Array representing the time each employee takes to inform their direct subordinates
    private int[] informTime;

    // Calculates the total time needed to inform all employees starting from the headID
    public int numOfMinutes(int n, int headID, int[] manager, int[] informTime) {
        // Initialize the graph with the number of employees
        graph = new List[n];
        Arrays.setAll(graph, k -> new ArrayList<>());
        this.informTime = informTime;

        // Build the graph, by adding subordinates to the manager's list
        for (int i = 0; i < n; ++i) {
            if (manager[i] >= 0) {
                graph[manager[i]].add(i);
            }
        }

        // Start depth-first search from the headID to calculate the total time
        return dfs(headID);
    }

    // Recursive method for depth-first search to calculate maximum inform time from a given node
    private int dfs(int nodeId) {
        // Initialize max time as 0 for current path
        int maxTime = 0;

        // Go through all direct subordinates of the current employee (nodeId)
        for (int subordinateId : graph[nodeId]) {
            // Recursive call to calculate the time for the current path,
            // comparing it with the maxTime found in other paths
            int currentTime = dfs(subordinateId) + informTime[nodeId];
            maxTime = Math.max(maxTime, currentTime);
        }

        // Return the maximum time found for all subordinates from this node
        return maxTime;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to calculate the total time required to inform all employees
    int numOfMinutes(int n, int headID, vector<int>& manager, vector<int>& informTime) {
        // Create a graph represented as an adjacency list to store the reporting hierarchy
        vector<vector<int>> graph(n);
        for (int i = 0; i < n; ++i) {
            if (manager[i] >= 0) {
                graph[manager[i]].push_back(i); // Populate the graph with employee-manager relationships
            }
        }

        // Recursive lambda function to perform depth-first search on the graph
        // to find the maximum inform time for each manager
        function<int(int)> dfs = [&](int employeeId) -> int {
            int maxInformTime = 0; // Initialize the maximum inform time for the current employee
            // Iterate over the direct reports of the current employee
            for (int reportId : graph[employeeId]) {
                // Calculate the total inform time using the direct report's inform time
                // and add the current employee's inform time
                maxInformTime = max(maxInformTime, dfs(reportId) + informTime[employeeId]);
            }
            return maxInformTime; // Return the maximum inform time for this branch
        };

        // Start the DFS traversal from the headID which is the root of the graph
        // to calculate the total inform time for the company
        return dfs(headID);
    }
};

```

Typescript Solution

```typescript
function numOfMinutes(employeeCount: number, headID: number, managers: number[], informTime: number[]): number {
    // Creating a graph where each node represents an employee and edges represent their direct reports
    const graph: number[][] = new Array(employeeCount).fill(0).map(() => []);

    // Populating the graph based on the manager array
    for (let i = 0; i < employeeCount; ++i) {
        if (managers[i] !== -1) {
            graph[managers[i]].push(i);
        }
    }

    // Depth-first-search function to find the maximum inform time for each employee
    const dfs = (employeeID: number): number => {
        let maxInformTime = 0;
        // Traverse all subordinates of the current employee
        for (const subordinateID of graph[employeeID]) {
            // Recursively get the inform time for each subordinate and find the maximum
            maxInformTime = Math.max(maxInformTime, dfs(subordinateID) + informTime[employeeID]);
        }
        // Return the maximum inform time from this node
        return maxInformTime;
    };

    // Initiate the DFS from the head of the company to calculate total inform time
    return dfs(headID);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is indeed `O(n)`. This is because each employee is visited exactly once in the depth-first search (DFS). In the DFS, the function `dfs` is called recursively for each subordinate of a manager, creating a tree-like structure of calls. Since there are `n` employees, and each of them will be processed once to calculate the time required to inform them, the time to traverse the entire employee hierarchy (or graph) is proportional to the number of employees, thus linear in the number of employees.

Space Complexity

As for the space complexity, it is also `O(n)`. The main factors contributing to space complexity are:

1. The recursive call stack of the DFS, which, in the worst case, could be `O(n)` if the organizational chart is a straight line (i.e., each employee has only one direct subordinate, forming a chain).
2. The adjacency list `g`, which stores the subordinate information. In the worst case, it will contain an entry for every employee, leading to `O(n)` space usage.

Combining both factors, the space complexity remains `O(n)`.