690. Employee Importance

Depth-First Search Breadth-First Search

Problem Description

Medium

a company. There is a data structure for employee information where each employee has a unique ID, a numeric value representing their importance, and a list of IDs for their direct subordinates. Our goal is to find the total importance value for a single employee specified by their ID, along with the importance of their direct and indirect subordinates. Essentially, we are tasked with performing a sum of importance values that spans across multiple levels of the employee reporting chain.

The problem revolves around calculating a collective importance score for an employee and their entire reporting hierarchy within

Hash Table

Intuition To solve this problem, we need a way to traverse through an employee's subordinates and all of their subsequent subordinates

recursively until there are none left. We should think of each employee as a node in a tree, where the given employee is the root, and their direct and indirect subordinates form the branches and leaves.

The first step is to map all the employees by their IDs for quick access. This is because we're provided with a list, and checking each employee's ID to find their subordinates would take a lot of time especially if there are many employees. With this mapping in place, we'll use recursion, which is a natural fit for this tree-like structure. The idea of the recursive depth-

first search (DFS) is to start with the given employee ID, find the employee's importance, and then call the function recursively for

each of its subordinates. This process will sum up all the importance values from the bottom of the hierarchy (the leaf nodes with no subordinates) all the way to the top (the employee whose ID was provided). An alternative approach could also be using breadth-first search (BFS), where we process all direct subordinates first before

moving on to the next level down in the hierarchy. However, the provided solution uses DFS, which is more intuitive for such hierarchical structures, often yielding simpler and more concise code. **Solution Approach**

The solution employs a depth-first search (DFS) algorithm which is a common technique used to traverse tree or graph data structures. Here's a step-by-step explanation of the implementation:

Hash Table Creation: We start by creating a hash table (in Python, a dictionary) to map each employee's ID to the

corresponding employee object. This mapping allows for quick access to any employee's details when given their ID, which is

the sum s.

total importance is to be calculated.

• Employee 3: Importance = 6, Subordinates = [4]

The dfs function operates as follows when we call dfs(1):

2. Start with a sum s = 5, the importance value of Employee 1.

■ Sum the importance for Employee 3 (s = 6).

Employee 1 and their entire reporting hierarchy.

self.importance = importance

self.subordinates = subordinates

def calculate_importance(emp_id: int) -> int:

Start with the employee's importance.

total_importance = employee.importance

employee = employee_map[emp_id]

return calculate_importance(id)

// Definition for Employee class provided.

Solution Implementation

self.id = id

4. After traversing all subordinates of Employee 1, the total sum s is 16.

1. Look up Employee 1 in the hash table 'm' and find its importance and subordinates.

■ Employee 3 has a subordinate, Employee 4, call dfs on Employee 4.

def __init__(self, id: int, importance: int, subordinates: list[int]):

Get the employee object based on the employee ID.

Call the recursive function starting from the given ID.

Add Employee 4's importance to the sum s (now s = 8).

efficient data access.

Return the total sum s after traversing all subordinates.

essential for efficient traversal. The hash table creation is represented by the line $m = \{emp.id: emp for emp in employees\}$ in the code. **Recursive DFS Function:** A recursive function named dfs is defined inside the Solution class. This function takes an

- employee's ID as an input and returns the total importance value of that employee, plus the importance of all their subordinates, both direct and indirect. This is implemented as follows: • Retrieve the Employee object corresponding to the given ID from the hash table. Initialize a sum s with the importance of the employee. Loop through the list of subordinates of the employee. For each subordinate ID, call the dfs function and add the returned importance to
- The importance of recursion here is that it naturally follows the hierarchy structure by diving deep into each branch (subordinate chain) and unwinding the importance values as the call stack returns. This DFS approach ensures that all employees in the hierarchy are accounted for and their importances are added up correctly to yield the final answer.

The code snippet return dfs(id) initiates the process by starting the recursive traversal, passing the ID of the employee whose

with the employee ID provided as an input to the problem and returns the importance value obtained from this call.

Initiating the DFS Call: The getImportance function is the entry point for the solution. It calls the DFS dfs function starting

Example Walkthrough

The key algorithm and data structure pattern used here is a combination of a DFS algorithm for traversal and a hash table for

Let's consider a small set of employee data: • Employee 1: Importance = 5, Subordinates = [2, 3] Employee 2: Importance = 3, Subordinates = []

• Employee 4: Importance = 2, Subordinates = [] We are asked to calculate the total importance score for Employee 1.

Recursive DFS Function:

Hash Table Creation:

```
3. Loop over Employee 1's subordinates (Employee 2 and 3):
    \circ For Employee 2: no subordinates, so just add 3 to the sum s (now s = 8).

    For Employee 3: Call dfs for Employee 3.
```

■ Look up Employee 3 in hash table, get importance (6), and its subordinate (Employee 4).

■ No more subordinates, return s (8). Add this to our main sum s which is now s = 16.

First, we create a hash table mapping employee IDs to their respective Employee objects for quick access:

 $m = \{1: Employee(1, 5, [2, 3]), 2: Employee(2, 3, []), 3: Employee(3, 6, [4]), 4: Employee(4, 2, [])\}$

```
Initiating the DFS Call:
```

The getImportance function initiates the DFS call with return dfs(1) and would return the total importance value of 16 for

Definition for Employee class.

class Employee:

Python

class Solution: def get_importance(self, employees: list['Employee'], id: int) -> int: # Create a dictionary where the key is the employee ID and the value is the employee object. # This allows for quick access to any employee object. employee_map = {employee.id: employee for employee in employees}

Recursive function to calculate the total importance value starting from the given employee ID.

For each of the employee's subordinates, add their importance recursively.

```
for subordinate_id in employee.subordinates:
    total_importance += calculate_importance(subordinate_id)
# Return the total importance accumulated.
return total_importance
```

import java.util.HashMap;

import java.util.List;

public int id;

public int importance;

import java.util.Map;

class Employee {

Java

```
public List<Integer> subordinates;
class Solution {
    // A hashmap to store employee id and corresponding Employee object for quick access.
    private final Map<Integer, Employee> employeeMap = new HashMap<>();
    // Calculates the total importance value of an employee and all their subordinates.
    public int getImportance(List<Employee> employees, int id) {
       // Populate the employeeMap with the given list of employees.
        for (Employee employee : employees) {
            employeeMap.put(employee.id, employee);
       // Start the Depth-First Search (DFS) from the given employee id.
        return dfs(id);
    // Helper method to perform DFS on the employee hierarchy and accumulate importance.
    private int dfs(int employeeId) {
       // Retrieve the current Employee object using its id.
        Employee employee = employeeMap.get(employeeId);
       // Start with the importance of the current employee.
       int totalImportance = employee.importance;
       // Recursively accumulate the importance of each subordinate.
        for (Integer subordinateId : employee.subordinates) {
            totalImportance += dfs(subordinateId);
       // Return the total importance accumulated.
       return totalImportance;
C++
#include <vector>
#include <unordered_map>
// C++ struct definition for Employee.
struct Employee {
 int id;
 int importance;
  std::vector<int> subordinates;
```

```
Employee* employee = it->second;
// Start with the importance of the current employee.
int total_importance = employee->importance;
```

return dfs(id);

importance: number;

subordinates: number[];

employees.forEach(employee => {

if (!employee) {

Definition for Employee class.

self.importance = importance

self.id = id

class Employee:

return 0;

TypeScript

};

/**

*/

});

/**

return 0;

/**

```
// TypeScript type definitions for Employee.
type Employee = {
 id: number;
```

* Calculates the total importance value of an employee and their subordinates.

* @return The total importance value of the employee and their subordinates.

const getImportance = (employees: Employee[], id: number): number => {

// Map to store employee id as key and employee object as value.

// Fill the map with the employees for constant time look-ups.

const employeeMap: Map<number, Employee> = new Map();

* @param id - The employee id for which the importance value needs to be calculated.

* Calculates the total importance value of an employee and their subordinates.

* @return The total importance value of the employee and their subordinates.

// Recursive helper function to calculate importance using depth-first search.

// Include the importance of each subordinate using depth-first search.

// Kick off the recursive depth-first search process using the provided id.

// Usage of the function remains the same as provided by the user.

* @param id - The employee id for which the importance value needs to be calculated.

// Unordered map to store employee id as key and employee pointer as value for constant time look-ups.

// If the employee is not found in the map, there is no such employee with the employeeId.

* @param employees - Vector of Employee pointers.

std::unordered map<int, Employee*> employee map;

// Fill the map with the employees.

if (it == employee_map.end()) {

for (Employee* employees) {

employee_map[employee->id] = employee;

auto it = employee_map.find(employeeId);

for (int subId : employee->subordinates) {

* @param employees - Array of Employee objects.

employeeMap.set(employee.id, employee);

const employee = employeeMap.get(employeeId);

let totalImportance = employee.importance;

// Start with the importance of the current employee.

total_importance += dfs(subId);

return total_importance;

int getImportance(std::vector<Employee*> employees, int id) {

std::function<int(int)> dfs = [&](int employeeId) -> int {

// Retrieve the current employee from the map using their id.

```
* Recursive helper function to calculate importance using depth-first search.
* @param employeeId - The id of the current employee being processed.
* @return The total importance including all subordinates' importance.
*/
const dfs = (employeeId: number): number => {
```

// Retrieve the current employee from the map using their id.

```
// Include the importance of each subordinate (depth-first search).
  employee.subordinates.forEach(subId => {
    totalImportance += dfs(subId);
  });
  return totalImportance;
};
// Kick off the recursive process using the provided id.
return dfs(id);
```

// If employee is undefined, there is no such employee with the employeeId.

```
self.subordinates = subordinates
class Solution:
   def get_importance(self, employees: list['Employee'], id: int) -> int:
       # Create a dictionary where the key is the employee ID and the value is the employee object.
       # This allows for quick access to any employee object.
        employee_map = {employee.id: employee for employee in employees}
```

Get the employee object based on the employee ID.

def calculate_importance(emp_id: int) -> int:

Start with the employee's importance.

for subordinate_id in employee.subordinates:

Call the recursive function starting from the given ID.

Return the total importance accumulated.

total_importance = employee.importance

employee = employee_map[emp_id]

return total_importance

return calculate_importance(id)

def __init__(self, id: int, importance: int, subordinates: list[int]):

// Usage of the function remain the same as provided by the user.

```
The given code defines a Solution class with a method getImportance to calculate the total importance value of an employee
and all of their subordinates recursively.
```

Time and Space Complexity

Time Complexity The time complexity of the code is O(N), where N is the total number of employees. This is because the dfs function ensures that each employee is processed exactly once. The creation of the dictionary m is O(N) since it involves going through all the

employees once and inserting them into the m mapping, and the recursive dfs calls process each employee and their

Recursive function to calculate the total importance value starting from the given employee ID.

For each of the employee's subordinates, add their importance recursively.

total_importance += calculate_importance(subordinate_id)

Space Complexity

subordinates once.

The space complexity of the code is also O(N). This involves the space for the dictionary m which stores all the employees, corresponding to O(N). Additionally, the space complexity takes into account the recursion call stack, which in the worst case, when the organization chart forms a deep hierarchy, can go as deep as the number of employees N in the path from the top employee to the lowest-level employee.