Organizational Hierarchy Analysis

Programming Assignment

Introduction

In modern software systems, hierarchical relationships are common—from organizational structures to file systems to category trees in e-commerce platforms. This assignment focuses on analyzing an organizational hierarchy where employees report to supervisors in a tree structure.

You will implement algorithms using **recursion**, **depth-first search**, and **dynamic programming with memoization** to solve real-world queries about this hierarchical data.

Dataset

You are provided with employee data in JSON format. Each employee has a unique ID, and most employees have a `supervisor_id` pointing to their manager. The CEO has no supervisor (`supervisor_id: null`).

TypeScript Interface

```typescript
interface Employee {
  id: number;
  name: string;
  position:
  string;
  salary:
  number;
  supervisor_id: number | null;
}

interface EmployeeData {
  employees: Employee[];
}
```

JSON Structure

```json
{
  "employees": [
    {
      "id": 1,
      "name": "Alice Johnson",
      "position": "CEO",
      "salary": 250000,
      "supervisor_id": null
    },
    {
      "id": 2,
      "name": "Bob Smith",
```

```
  "position": "CTO",
  "salary": 180000,
  "supervisor_id": 1
},
{
  "id": 3,
  "name": "Carol White",
  "position": "CFO",
  "salary": 175000,
  "supervisor_id": 1
},
{
  "id": 4,
  "name": "David Brown",
  "position": "Engineering Manager",
  "salary": 140000,
  "supervisor_id": 2
},
{
  "id": 5,
  "name": "Eve Davis",
  "position": "QA Manager",
  "salary": 130000,
  "supervisor_id": 2
},
{
  "id": 6,
  "name": "Frank Wilson",
  "position": "Senior Accountant",
  "salary": 95000,
  "supervisor_id": 3
},
{
  "id": 7,
  "name": "Grace Lee",
  "position": "Senior Developer",
  "salary": 120000,
  "supervisor_id": 4
},
{
  "id": 8,
  "name": "Henry Martinez",
  "position": "Junior Developer",
  "salary": 85000,
  "supervisor_id": 4
},
{
  "id": 9,
  "name": "Ivy Chen",
  "position": "QA Engineer",
  "salary": 90000,
  "supervisor_id": 5
```

```
    },
    {
      "id": 10,
      "name": "Jack Thompson",
      "position": "DevOps Engineer",
      "salary": 110000,
      "supervisor_id": 4
    },
    {
      "id": 11,
      "name": "Kelly Anderson",
      "position": "Junior Accountant",
      "salary": 65000,
      "supervisor_id": 6
    },
    {
      "id": 12,
      "name": "Liam Garcia",
      "position": "Intern Developer",
      "salary": 50000,
      "supervisor_id": 7
    }
  ]
}
```

Organizational Tree Visualization

```
Alice Johnson (CEO, id=1)
 Bob Smith (CTO, id=2)
  David Brown (Engineering Manager, id=4)
      Grace Lee (Senior Developer, id=7)
         Liam Garcia (Intern Developer, id=12)
      Henry Martinez (Junior Developer, id=8)
      Jack Thompson (DevOps Engineer, id=10)
   Eve Davis (QA Manager, id=5)
       Ivy Chen (QA Engineer, id=9)
 Carol White (CFO, id=3)
     Frank Wilson (Senior Accountant, id=6)
         Kelly Anderson (Junior Accountant, id=11)
```

Problem 1: Calculate Employee Depth

Description

Determine the hierarchical depth (level) of any employee in the organization. The CEO is at depth 0, direct reports to the CEO are at depth 1, and so on.

Function Signature

```
function getEmployeeDepth(employees: Employee[], employeeId: number): number {
```

```
  /**
   * Calculate the depth of an employee in the organizational hierarchy.*
   * @param employees - Array of employee objects
   * @param employeeId - The ID of the employee to find depth for
   * @returns The depth level (CEO = 0), or -1 if employee not found
   */
  // Your implementation here
}
```

Input Format

- `employees`: A list of dictionaries, where each dictionary represents an employee
- `employee_id`: An integer representing the employee's unique ID

Output Format

Return an integer representing the depth level. Return -1 if the employee ID doesn't exist.

Example

```
getEmployeeDepth(employees, 1); // Returns: 0 (Alice - CEO)
getEmployeeDepth(employees, 2); // Returns: 1 (Bob - CTO)
getEmployeeDepth(employees, 7); // Returns: 3 (Grace - Senior Developer)
getEmployeeDepth(employees, 12); // Returns: 4 (Liam - Intern)
getEmployeeDepth(employees, 999); // Returns: -1 (Not found)
```

Requirements

- Implement using **recursion**
- Handle invalid employee IDs gracefully
- Detect circular references (cycles) and return -1 if found
- Time complexity should be O(h) where h is the height of the tree

---

Problem 2: Get All Subordinates

Description

Given a manager's employee ID, retrieve all employees who report to them directly or indirectly (the entire subtree). Include each subordinate's depth relative to the specified manager.

Function Signature

```
interface Subordinate {
  id: number;
  name: string;
  relative_depth: number;
}

function getAllSubordinates(employees: Employee[], managerId: number): Subordinate[] {
  /**
```

```
   *         Retrieve all subordinates under a given manager.
   *
   *         @param employees - Array of employee objects
   *         @param managerId - The ID of the manager
   *         @returns Array of subordinate objects sorted by relative_depth (ascending),
             then by id (ascending)
   *         Format: [{id: number, name: string, relative_depth: number}, ...]
   */
  // Your implementation here
}
```

Input Format

- `employees`: A list of dictionaries containing employee information
- `manager_id`: An integer representing the manager's unique ID

Output Format

Return a list of dictionaries, each containing: - `id`: The subordinate's employee ID - `name`: The subordinate's full name - `relative_depth`: How many levels below the manager (direct reports = 1)

Sort the results first by `relative_depth` (ascending), then by `id` (ascending).

Example

```
getAllSubordinates(employees,
2); // Returns:
[
    {id: 4, name: "David Brown", relative_depth: 1},
    {id: 5, name: "Eve Davis", relative_depth: 1},
    {id: 7, name: "Grace Lee", relative_depth: 2},
    {id: 8, name: "Henry Martinez", relative_depth: 2},
    {id: 9, name: "Ivy Chen", relative_depth: 2},
    {id: 10, name: "Jack Thompson", relative_depth: 2},
    {id: 12, name: "Liam Garcia", relative_depth: 3}
]

getAllSubordinates(employees,
6); // Returns:
[
    {id: 11, name: "Kelly Anderson", relative_depth: 1}
]

getAllSubordinates(employees,
12); // Returns: [] (No
subordinates)
```

Requirements

- Implement using **depth-first search (DFS) with recursion**
- Return an empty list if the manager has no subordinates
- Results must be sorted as specified

Problem 3: Find Path to CEO

Description

Construct the complete management chain from any employee up to the CEO. This represents the path through the tree from a leaf (or internal node) to the root.

Function Signature

```
function getPathToCeo(employees: Employee[], employeeId: number): number[] {
  /**
   *         Find the path from an employee to the CEO.
   *
   *         @param employees - Array of employee objects
   *         @param employeeId - The starting employee's ID
   *         @returns Array of employee IDs representing the path from employee to CEO
             (inclusive),
   *         or empty array if not found
   */
  // Your implementation here
}
```

Input Format

- `employees`: A list of dictionaries containing employee information
- `employee_id`: An integer representing the starting employee's ID

Output Format

Return a list of integers representing employee IDs from the starting employee to the CEO (inclusive).

Example

```
getPathToCeo(employees,
12); // Returns: [12, 7,
4, 2, 1]
// Path: Liam Garcia -> Grace Lee -> David Brown -> Bob Smith -> Alice Johnson

getPathToCeo(employees,
1); // Returns: [1]
// The CEO has no supervisor

getPathToCeo(employees, 5);
// Returns: [5, 2, 1]
// Path: Eve Davis -> Bob Smith -> Alice Johnson
```

Requirements

- Implement using recursion or iteration
- Return an empty list if the employee ID doesn't exist

- Handle the CEO case (path contains only the CEO)

Problem 4: Lowest Common Manager

Description

Given two employees, find their lowest common manager—the closest supervisor that both employees share in the organizational hierarchy. This is analogous to finding the Lowest Common Ancestor (LCA) in a tree.

Function Signature

```
interface CommonManager {
  manager_id: number;
  manager_name: string;
  distance_to_emp1: number;
  distance_to_emp2: number;
}

function findLowestCommonManager(
  employees:
  Employee[], empId1:
  number, empId2:
  number
): CommonManager | null
  { /**
   * Find the lowest common manager of two employees.
   *
   * @param employees - Array of employee objects
   * @param empId1 - First employee's ID
   * @param empId2 - Second employee's ID
   * @returns Object containing manager information or null if either employee doesn't
     exist */
  // Your implementation here
}
```

Input Format

- `employees`: A list of dictionaries containing employee information
- `emp_id_1`, `emp_id_2`: Integers representing the two employees' IDs

Output Format

Return an object containing: - `manager_id`: The ID of the lowest common manager - `manager_name`: The full name of the manager - `distance_to_emp1`: Number of steps from empId1 to the common manager - `distance_to_emp2`: Number of steps from empId2 to the common manager Return `null` if either employee doesn't exist.

Example

```
findLowestCommonManager(employees, 12,
9); // Returns:
```

```
{ manager_id: 2,
    manager_name: "Bob
    Smith",
    distance_to_emp1: 3, // Liam -> Grace -> David -> Bob
    distance_to_emp2: 2  // Ivy -> Eve -> Bob
}

findLowestCommonManager(employees, 8,
10); // Returns:
{ manager_id: 4,
    manager_name: "David
    Brown",
    distance_to_emp1:  //   Henry   ->
    1,                 David
    distance_to_emp2: 1//   Jack    ->
                       David
}

findLowestCommonManager(employees, 1,
12); // Returns:
{ manager_id:
    1,
    manager_name: "Alice Johnson",
    distance_to_emp1: 0, // Alice is already the manager
   distance_to_emp2: 4// Liam -> Grace -> David -> Bob -> Alice
}
```

Requirements

- Use the path-finding approach: find paths from both employees to CEO, then find intersection
- Consider edge cases: one employee is an ancestor of the other
- Optimize to avoid redundant path calculations if possible

Problem 5: Team Salary with Memoization

Description

Calculate the total salary budget for a manager's entire team, including the manager and all direct and indirect reports. Implement this using **dynamic programming with memoization** to optimize performance when the function is called multiple times.

Function Signature

```
function calculateTeamSalary(
  employees: Employee[],
  managerId: number,
  memo: Map<number, number> = new Map()
): number {
  /**
```

```
   * Calculate total salary for a manager and all subordinates using
     memoization.
   *
   * @param employees - Array of employee objects * @param managerId - The
     manager's employee ID
   * @param memo - Map to store computed results (optional, for memoization)
   * @returns Total salary for the entire team, or 0 if manager not found */
  // Your implementation here
}
```

Input Format

- `employees`: An array of employee objects
- `managerId`: An integer representing the manager's ID
- `memo`: Optional Map for storing cached results (initialize as empty Map if not provided) **Output Format**

Return an integer representing the total salary of the manager and all subordinates.

Example

```
calculateTeamSalary(employees, 1);
// Returns: 1490000
// (Sum of all 12 employees' salaries)

calculateTeamSalary(employees, 4);
// Returns: 505000
// (David: 140000 + Grace: 120000 + Henry: 85000 + Jack: 110000 + Liam: 50000)

calculateTeamSalary(employees, 7);
// Returns: 170000
// (Grace: 120000 + Liam: 50000)

calculateTeamSalary(employees, 12);
// Returns: 50000
// (Liam has no subordinates, only his own salary)
```

Requirements

- **Must implement memoization** (caching of computed subtree results) • Store results in the `memo` Map to avoid recalculating the same subtree • Demonstrate the performance benefit:
  - Add a global or class variable counter to track function calls
  - Show that calling the function multiple times reuses cached results
  - Include a test case that proves memoization is working
- Handle the case where managerId doesn't exist (return 0)

Memoization Demonstration Example

```
// Without memoization: may call function many times for same
manager let callCount = 0;

// With memoization: reuses previously calculated
results const memo = new Map<number, number>();
```

```
const salary1 = calculateTeamSalary(employees, 1,  // Calculates everything
memo);
const salary2 = calculateTeamSalary(employees, 2,  // Reuses cached result for
memo);                                             id=2
const salary4 = calculateTeamSalary(employees, 4,  // Reuses cached result for
memo);                                             id=4
// The memo Map should now contain results for multiple managers
```