Lab 7&8

Problem 1

Alex, a transportation planner at a city transport company, is tasked with reviewing the city's subway network to ensure it's designed as a tree. A tree is a connected graph with no cycles and exactly N−1 edges, where N is the number of stations (vertices). Each station in the subway system is connected to other stations by specific routes (edges).

The planning department has provided Alex with the number of stations (N) and the number of routes each station connects to (degree). Alex needs to determine whether the subway network can be considered a tree based on this information.

Input:

- The number of stations, N, in the subway network.
- A list of N integers, where each integer represents the number of routes (degree) each station is connected to.

Output:

Print "Yes" if the subway system forms a tree, otherwise print "No".

Questions:

1. **Describe the algorithm approach to solve this problem.**
   (Provide a step-by-step explanation of how Alex can determine if the subway system forms a tree based on the number of stations and the degree of each station.)
2. **Consider the following inputs:**
   a. N=4, Degrees = [1, 2, 1, 2]
   b. N=5, Degrees = [2, 3, 2, 2, 3]

   Do these subway networks represent a tree? Explain your reasoning.

Answer

A tree in graph theory has two main properties:

1. **It is connected:** All stations (vertices) are reachable from any other station.
2. **It has no cycles:** There are no loops or redundant routes in the system.
3. **It has exactly N−1 edges:** This is crucial. For a graph to be a tree, the number of edges (routes) must always be one less than the number of stations, because any more edges would create a cycle, and fewer would mean some stations are disconnected.

Given that we are only provided with the degree (number of edges connected to each station), we can check if these conditions hold true for the system. Specifically:

- The sum of the degrees of all stations should be **exactly twice the number of edges** in the graph (since each edge connects two stations).
- There should be **exactly N−1 edges** in total.
- If the total number of edges is less than N−1, we know that the graph is disconnected, so it's not a tree.
- If the total number of edges is more than N−1, we know that there's a cycle in the graph, which violates the acyclic property of trees.

With this in mind, the task becomes checking these two properties using the given degree information to determine if the system is a tree.

Algorithm Approach

1. **Check if the number of edges is correct:**
    - First, compute the total sum of the degrees of all stations.
      Each edge connects two stations, so the sum of the degrees is always twice the number of edges in the graph. That is, if the sum of the degrees is S, then the number of edges E should be E=S/2
    - If S is odd, the graph cannot form a valid structure, because the sum of degrees should always be even (since each edge contributes to two degrees).

2. **Check if the graph has exactly N−1 edges:**
    - After finding the total number of edges, check if it equals N−1. If it does, then the graph could potentially be a tree (i.e., it's a connected acyclic graph with the right number of edges). If not, the graph cannot be a tree.

3. **Final check:**
    - If the sum of the degrees is even and the number of edges is N−1, then the graph is a tree. If either of these conditions fails, then the graph is not a tree.

Step-by-Step Algorithm

1. **Input:**
    - Let N be the number of stations.
    - Let `degrees` be the list of degrees of all N stations.

2. **Calculate the sum of degrees:**
    - Let `sum_of_degrees = sum(degrees)`.

- If `sum_of_degrees` is odd, print "No" because a valid graph cannot have an odd sum of degrees.

3. **Calculate the number of edges:**

   - Let `E = sum_of_degrees / 2`. This is the number of edges in the graph.
   - If `E` is not equal to N−1, print "No" because the graph doesn't have the correct number of edges for a tree.

4. **Output:**

   - If both checks pass, print "Yes", as the graph forms a tree.
   - Otherwise, print "No".

Example Walkthrough:

Example 1

```
4
1 2 1 2
```

- Sum of degrees = 1 + 2 + 1 + 2 = 6
- Number of edges = 6/2=3
- Expected number of edges for a tree with 4 vertices = 4−1=3

Since the number of edges is correct, and the sum of degrees is even, the graph satisfies the conditions for being a tree. **The output is "Yes"**.

Example 2

```
5
2 3 2 2 3
```

- Sum of degrees = 2 + 3 + 2 + 2 + 3 = 12
- Number of edges = 12/2=6
- Expected number of edges for a tree with 5 vertices = 5−1=4

The number of edges is incorrect (6 instead of 4), so **the output is "No"**.

Problem 2

Sophie, a marine biologist, has arrived at an archipelago of N islands. Each island is connected to one or more others by bidirectional bridges. Sophie is currently at Island #1 and needs to

reach Island #N to conduct her research. However, she wants to minimize the number of bridges she has to cross because the bridges are old and require a lot of effort to cross.

Sophie is wondering how many bridges she will need to cross to get to Island #N, taking the optimal route. Can you help Sophie find the minimum number of bridges she must cross to reach her destination?

You are given:

- The total number of islands N and the total number of bridges connecting them.
- A list of connections between the islands in the form of bidirectional bridges.
- The index of the starting island, which is always Island #1.

You need to determine how many bridges Sophie will cross on the shortest route, or if it's even possible to reach Island #N.

Questions:

1. Describe the approach Sophie should use to find the minimum number of bridges to cross.
2. Given the following connections between the islands:
    - `Islands: 6`
    - `Bridges: 7`
    - `Connections: 1-2, 2-3, 3-4, 1-4, 5-4, 6-5, 4-6`
    - `Sophie starts at Island 1.`

    What is the optimal route Sophie should take to reach Island #6, and how many bridges will she cross? Explain your reasoning.

Answer

Question 1: Describe the approach Sophie should use to find the minimum number of bridges to cross.

To find the **minimum number of bridges** Sophie needs to cross, we can model the islands and their bridges as a **graph**. Here:

- **Islands** are represented as **nodes**.
- **Bridges** are represented as **edges** connecting these nodes.

The problem Sophie faces is essentially a **shortest path problem** in an unweighted graph, where the goal is to find the **minimum number of edges** (bridges) between the starting node (Island #1) and the destination node (Island #N).

To solve this efficiently, we use **Breadth-First Search (BFS)**. Here's why BFS is a suitable algorithm for this problem:

1. **Unweighted Graph**: Each bridge is considered to have the same weight (effort), so the shortest path in terms of the **number of bridges** is equivalent to finding the path with the fewest edges.

2. **Layered Exploration**: BFS explores all nodes level by level (or layer by layer), ensuring that the first time it reaches a node, it does so using the minimum number of edges. This makes BFS ideal for finding the shortest path in terms of the number of edges.

3. **Optimal Path**: By starting from the source island (Island #1), BFS will traverse through all possible routes. When BFS reaches Island #N, it will have traversed the fewest possible edges to get there, ensuring that Sophie crosses the minimum number of bridges.

The steps for this approach are:

1. Start BFS from Island #1.

2. Use a queue to explore all neighboring islands, marking each island as **visited** and tracking the number of bridges crossed.

3. If BFS reaches Island #N, the number of bridges crossed so far is the answer.

4. If BFS completes and Island #N isn't reached, it means there's no path to Island #N.

Question 2: Given the following connections between the islands:

- **Islands**: 6
- **Bridges**: 7
- **Connections**: 1-2, 2-3, 3-4, 1-4, 5-4, 6-5, 4-6
- **Starting Island**: Island #1
- **Destination Island**: Island #6

What is the optimal route Sophie should take to reach Island #6, and how many bridges will she cross? Explain your reasoning.

Let's apply the BFS algorithm to find the shortest path from **Island #1** to **Island #6**.

1. **Graph Representation**: The graph can be represented as an adjacency list:
    - `1: [2, 4]`
    - `2: [1, 3]`
    - `3: [2, 4]`
    - `4: [1, 3, 5, 6]`
    - `5: [4, 6]`
    - `6: [5, 4]`
2. **BFS Process**:
    - **Start at Island #1**:
        - Current Queue: [1]
        - Visited: {1}
        - Distance: {1: 0}
    - **Explore neighbors of Island #1**:
        - From Island #1, we can go to Island #2 and Island #4.
        - Update Queue: [2, 4]
        - Visited: {1, 2, 4}
        - Distance: {1: 0, 2: 1, 4: 1}
    - **Explore neighbors of Island #2**:
        - From Island #2, we can go to Island #1 and Island #3. But Island #1 is already visited, so we move to Island #3.
        - Update Queue: [4, 3]
        - Visited: {1, 2, 3, 4}
        - Distance: {1: 0, 2: 1, 4: 1, 3: 2}
    - **Explore neighbors of Island #4**:

        - From Island #4, we can go to Island #1, Island #3, Island #5, and Island #6.
        - Island #1 and Island #3 are already visited, so we move to Island #5 and Island #6.
        - Update Queue: [3, 5, 6]
        - Visited: {1, 2, 3, 4, 5, 6}
        - Distance: {1: 0, 2: 1, 4: 1, 3: 2, 5: 2, 6: 2}
3. **Reaching Island #6**:
    - At this point, we have reached **Island #6** after crossing **2 bridges**.
4. **Optimal Route**: The **optimal route** to Island #6 is **1 → 4 → 6** (2 bridges).

Answer: The optimal route is 1 → 4 → 6, and Sophie will cross 2 bridges.

Reasoning:

- BFS guarantees that the first time we reach a node (Island #6), we will do so with the minimum number of bridges, which in this case is 2.
- Since BFS explores all nodes level by level, we find the shortest path in terms of the number of bridges. The queue ensures that each node is explored with the minimum number of edges (bridges) from the starting point.

Problem 3

You are working as a network engineer for a large communication company. The company's data center is designed as a network of nodes, with each node representing a server. These servers are connected by edges, which represent communication links between them. Each server in the data center is identified by a node, and there is one special node called the **head node**, which is responsible for coordinating the network.

You need to check how many servers (nodes) are unreachable from the head node. Some servers might be isolated or disconnected from the head node due to network issues. The graph representing the network may have multiple edges between the same nodes or self-loops, and it's undirected. Your task is to identify how many nodes are unreachable from the head node.

Input:

- The total number of nodes (servers) in the network, followed by the total number of communication links (edges).
- A list of M communication links between servers. Each link is represented as a pair of nodes aaa and b, where there is a communication link between node a and node b.
- The index of the head node.

Output:

- Print a single integer, denoting the number of servers (nodes) that are unreachable from the head node.

Questions:

1. Describe the algorithm approach to solve this problem.
2. Given the following input:

   ```
   10 servers in total, and 10 communication links between them.
   Connections between the servers are as follows:
   8-1, 8-3, 7-4, 7-5, 2-6, 10-7, 2-8, 10-9, 2-10, 5-10
   The head node is 2.
   ```

   How many servers are unreachable from the head node (node 2)? What are these unreachable servers? Explain your reasoning.

Answer

Question 1: Describe the algorithm approach to solve this problem.

To solve the problem of determining the number of **unreachable servers** from the **head node** in a **network** of servers (represented as a graph), we can apply **Depth-First Search (DFS)**. Let's break this down step by step:

1. **Graph Representation**:
   - The servers are represented as **nodes** in a graph.
   - The communication links between servers are represented as **edges**.
   - The **head node** is the starting point from which we need to find all reachable nodes.
2. **DFS Approach**:
   - **DFS** is a natural choice for this problem because it explores nodes by going as deep as possible before backtracking. This ensures that we can explore all

reachable servers from the head node.

   - The reason for using **DFS** over **BFS** or other approaches is its ability to explore all nodes in a connected component in a recursive manner. When we reach a node, we can check if it is connected to other servers. If it is, we recursively explore those connections.

   - In DFS, we use a **visited list** to keep track of nodes that have already been explored. This prevents revisiting the same node and ensures we don't get stuck in cycles.

3. **Why DFS**:

   - **Graph traversal**: Since the problem involves finding all reachable servers from the head node, **DFS** efficiently solves this by visiting nodes along the depth of the graph. Once DFS finishes, we can identify which nodes are reachable and which are not.

4. **Algorithm Approach**:

   - Start by building the graph using an adjacency list.

   - Perform **DFS** starting from the head node. Mark each node that is visited as **reachable**.

   - After the DFS traversal, count the number of nodes that were not marked as visited. These are the **unreachable nodes**.

   - Finally, return the number of unreachable nodes and list them.

By applying **DFS**, we ensure that every node that is connected to the head node is explored, and we can easily identify and count the unreachable nodes.

Question 2: Given the following input, how many servers are unreachable from the head node (node 2)? What are these unreachable servers? Explain your reasoning.

Input:

- 10 servers (nodes) in total.

- 10 communication links (edges) between servers.

- The connections between servers are:
  `8-1, 8-3, 7-4, 7-5, 2-6, 10-7, 2-8, 10-9, 2-10, 5-10`

- The **head node** is **2**.

Step-by-Step Reasoning:

1. **Graph Representation**: We can construct an adjacency list from the provided connections:
   ```
   1: [8]
   2: [6, 8, 10]
   3: [8]
   4: [7]
   5: [7, 10]
   6: [2]
   7: [4, 5, 10]
   8: [1, 2, 3]
   9: [10]
   10: [2, 5, 7, 9]
   ```
2. **DFS Traversal**: We start DFS from the head node **2** and explore all reachable nodes:
   - From node **2**, we can go to nodes **6**, **8**, and **10**.
   - From node **6**, we can only go back to **2**, which we already visited.
   - From node **8**, we can go to nodes **1**, **2**, and **3**. Since **2** is already visited, we go to **1** and **3**.
   - From node **1**, we go to node **8**, which is already visited.
   - From node **3**, we go to node **8**, which is already visited.
   - From node **10**, we can go to nodes **2**, **5**, **7**, and **9**. Since **2** is visited, we explore **5**, **7**, and **9**.
   - From node **5**, we go to nodes **7** and **10**, but **7** is visited, and **10** is already explored.
   - From node **7**, we go to nodes **4**, **5**, and **10**, but **5** and **10** are already visited. We explore **4**.
   - From node **4**, we can only go to node **7**, which is already visited.

3. **Visited nodes**: {2, 6, 8, 1, 3, 10, 5, 7, 4, 9}
4. **Unreachable Nodes**:
    - All nodes are reachable from node **2**, so there are **no unreachable nodes**.

Answer:

- **Number of unreachable nodes**: **0**.
- **Unreachable nodes**: None.

Problem 4

In a small village, there are n villagers, each labeled from 1 to n. A rumor is spreading that one of the villagers is secretly the village mayor. The mayor has a special role in the village and is trusted by everyone except themselves. The mayor trusts no one.

You are tasked with identifying the mayor based on a series of trust relationships. A trust relationship between two villagers is represented as a pair [a,b], meaning that villager aaa trusts villager b. If a trust relationship does not exist between two villagers, no trust relationship exists between them.

The mayor must meet two criteria:

1. The mayor trusts nobody.
2. Everyone else (except for the mayor) trusts the mayor.

If a mayor exists, return the label of the mayor; otherwise, return -1.

Example:

Input:

```
n = 3, trust = [[1, 3], [2, 3]]
```

Output:

```
3
```

Explanation:
Villagers 1 and 2 both trust villager 3, and villager 3 trusts no one. Therefore, villager 3 is the mayor of the village.

Questions:

1. Describe the algorithm approach to solve this problem.
2. Given the following input:
   `n = 4, trust = [[1, 4], [2, 4], [3, 4], [4, 1]]`

   Can you identify the mayor of the village? If so, who is it? If not, explain why.

Answer

To identify the mayor in a village, we need to rely on two clear behaviors:

1. **The mayor trusts no one** – they don't rely on anyone else.

2. **Everyone else trusts the mayor** – because they believe the mayor is in charge.

This situation can be modeled using a **directed graph**, where each villager is a node, and a trust relationship is a directed edge from one node to another.

In graph theory:

- **Outdegree** of a person = number of people they trust (edges going out).

- **Indegree** of a person = number of people who trust them (edges coming in).

So, the mayor must have:

- **Outdegree = 0** (they trust no one).

- **Indegree = n - 1** (everyone else trusts them).

By tracking the difference between indegree and outdegree for each person, we can efficiently find the one who fits both conditions.

Algorithm Approach:

1. **Create a `count` array** of size `n + 1` (because people are labeled from 1 to n).
    - `count[i]` will track the net trust score: how many people trust person `i` minus how many they trust.
2. **Iterate through all trust relationships**:
    - For a pair `[a, b]`, person `a` trusts person `b`.
        - So, **decrease** `count[a]` by 1 (they trust someone).
        - And **increase** `count[b]` by 1 (they are trusted).
3. **Look for the mayor**:
    - The person who has `count[i] === n - 1` is the one who is trusted by everyone else and trusts no one.
4. **Return their label**, or return `-1` if no such person exists.

Step by step example

Input:

```
n = 4
trust = [[1, 4], [2, 4], [3, 4], [4, 1]]
```

Step-by-step reasoning:

- Let's compute the trust changes:

Person	Trusts Someone (Outdegree -1)	Is Trusted By (Indegree +1)	Net Trust
1	-1	0	-1
2	-1	0	-1
3	-1	0	-1
4	-1	+3	+2

- Now, check for the person with **net trust = n - 1 = 3**:
    - Person 4 has a net trust of +2, not +3, because they **trusted person 1**, violating the rule that the mayor trusts no one.

Answer:
No, there is **no mayor** in this village. Although person 4 is trusted by everyone else, they also **trust person 1**, which disqualifies them.

Problem 5

You are tasked with evaluating a system of interconnected devices in a smart home network. Each device is represented as a node in a directed graph, and connections between devices are represented by edges. The devices communicate with each other based on these connections, and some devices are terminal devices, meaning they do not communicate with any other device.

A device is considered **safe** if every possible communication starting from that device eventually leads to a terminal device or another safe device. Your task is to find all the safe devices in the network and return them in ascending order.

The system is represented by a 0-indexed 2D array called `graph`, where each index i represents a device, and `graph[i]` contains the list of devices that device i communicates with. A device is a **terminal device** if it does not communicate with any other devices.

Input:

- An integer array `graph` of size n, where `graph[i]` is an array representing the devices that device iii communicates with.
- Each device is represented by a node labeled from 0 to n−1n-1n−1.

Output:

- Return an array containing all the safe devices, sorted in ascending order.

Example:

Input:

```
graph = [[1, 2], [2, 3], [5], [0], [5], [], []]
```

Output:

```
[2, 4, 5, 6]
```

Explanation:

- Devices 5 and 6 are terminal devices because they do not communicate with any other devices.
- Every communication starting at devices 2, 4, 5, and 6 eventually leads to either device 5 or 6, which are terminal devices.
- Therefore, the safe devices are 2, 4, 5, and 6.

Questions:

1. **Describe the approach to solve the problem of identifying safe devices.** (Explain how you can determine which devices are safe based on the communication paths and terminal devices in the network.)
2. **Consider the following graph:** `graph = [[1, 2, 3, 4], [1, 2], [3, 4], [0, 4], []]`

   Which devices are safe in this network? Explain why.

Answer:

To identify safe devices in the given smart home network represented as a directed graph, we use **Depth-First Search (DFS)**. The main goal is to explore whether each device eventually leads to a **terminal device** (safe) or to a **cycle** (unsafe). DFS is particularly useful here because it allows us to traverse the graph deeply and detect cycles in the process.

Why DFS?

- **Cycle Detection**: DFS excels in detecting cycles in a directed graph, which is crucial for identifying unsafe devices. If a device leads to a cycle, it is considered unsafe because the communication from that device will not terminate — it will loop infinitely.

- **Path Exploration**: DFS explores each path starting from a device, allowing us to determine whether that path leads to a terminal device (safe) or an unsafe cycle. This depth-first exploration helps fully capture the communication chain of each device.

- **Backtracking**: One of the core features of DFS is backtracking — once all neighbors of a device have been explored, the search backtracks to the previous device to explore other paths. This makes it easy to mark devices as either safe or unsafe after fully exploring all potential communication routes.

State Tracking: The Key to Identifying Safe and Unsafe Devices

In DFS, state tracking is essential for determining the safety of each device. We use **two arrays** to manage the state of each device during the DFS traversal:

1. **`visited[]` Array**:
    - **Purpose**: The `visited[]` array tracks whether a device has been fully explored. This ensures that each device is processed only once.
    - **How it Works**:
        - Initially, all devices are marked as unvisited (`visited[i] = False`).
        - When we begin DFS on a device `i`, we mark it as visited (`visited[i] = True`).
        - Once all neighbors of the device are explored (i.e., the DFS path has been completely followed), we consider the device fully explored. If the device has been explored before, we skip further DFS on it.
    - **When it's Used**:
        - The `visited[]` array helps avoid **redundant DFS calls**. If a device has been visited and fully explored, we don't need to start DFS from it again, ensuring efficiency.
2. **`inStack[]` Array**:
    - **Purpose**: The `inStack[]` array keeps track of whether a device is part of the **current DFS recursion stack**. This is essential for detecting cycles. If we encounter a device that is already in the recursion stack, it means we have visited it during the current DFS path, and thus, a cycle has been detected.
    - **How it Works**:
        - When we start exploring a device `i`, we mark it as part of the recursion stack (`inStack[i] = True`).
        - As we recursively explore its neighbors, the device remains in the stack until all its neighbors have been fully explored.
        - Once we finish exploring all neighbors of a device, we mark it as **no longer part of the recursion stack** (`inStack[i] = False`) and

backtrack to explore other paths.

- **Cycle Detection**:
  - **Key role**: If we encounter a device that is already in the `inStack` during our DFS exploration, it means that device is involved in a cycle. This is because we're revisiting a device that's still being explored as part of the current path, indicating that we've looped back to it, forming a cycle.
  - **Unsafe Devices**: Any device that is part of a cycle or leads to a cycle is marked as **unsafe**.
- **When it's Used**:
  - The `inStack[]` array is crucial for **cycle detection** and helps us identify unsafe devices quickly.

Step-by-Step Example:

We will perform a Depth-First Search (DFS) for each device, and during the DFS, we will track:

- Whether a device is part of a cycle (`inStack[]`).
- Whether a device has been fully explored (`visited[]`).

1. Initialize `visited[]` and `inStack[]`:

- `visited = [False, False, False, False, False]` (no devices have been explored yet).
- `inStack = [False, False, False, False, False]` (no devices are in the DFS recursion stack).

2. DFS on Device 0:

- Mark device `0` as visited and part of the recursion stack:
  `visited[0] = True, inStack[0] = True`
- Explore its neighbors: `[1, 2, 3, 4]`.

3. DFS on Device 1 (Neighbor of Device 0):

- Mark device `1` as visited and part of the recursion stack: `visited[1] = True, inStack[1] = True`
- Explore its neighbors: `[1, 2]`.

4. DFS on Device 1 (Self-loop):

- **Cycle Detected**: Device `1` points to itself, so we detect a cycle.
- **Device 1 is unsafe** because it is part of a cycle.
- Backtrack from device `1`, marking it as no longer part of the recursion stack (`inStack[1] = False`).

5. DFS on Device 2 (Neighbor of Device 0):

- Mark device `2` as visited and part of the recursion stack: `visited[2] = True, inStack[2] = True`
- Explore its neighbors: `[3, 4]`.

6. DFS on Device 3 (Neighbor of Device 2):

- Mark device `3` as visited and part of the recursion stack: `visited[3] = True, inStack[3] = True`
- Explore its neighbors: `[0, 4]`.

7. DFS on Device 0 (Neighbor of Device 3):

- **Cycle Detected**: Device `0` is already in the recursion stack (`inStack[0] = True`), indicating a cycle.
- **Device 3 is unsafe** because it leads to a cycle (via device `0`).
- Backtrack from device `3`, marking it as no longer part of the recursion stack (`inStack[3] = False`).

8. DFS on Device 4 (Neighbor of Device 3):

- Mark device `4` as visited and part of the recursion stack: `visited[4] = True, inStack[4] = True`
- Device `4` is a **terminal device** (it has no outgoing edges).
- **Device 4 is safe** because it is a terminal device.
- Backtrack from device `4`, marking it as no longer part of the recursion stack (`inStack[4] = False`).

9. Backtrack to Device 2:

- No more neighbors to explore for device `2`.
- **Device 2 is unsafe** because it leads to device `3`, which is part of a cycle.
- Backtrack from device `2`, marking it as no longer part of the recursion stack (`inStack[2] = False`).

10. Backtrack to Device 0:

- No more neighbors to explore for device `0`.
- **Device 0 is unsafe** because it leads to a cycle (via device `3`).
- Backtrack from device `0`, marking it as no longer part of the recursion stack (`inStack[0] = False`).

Final Result:

- **Device 0**: Unsafe because it leads to a cycle (via device 3).
- **Device 1**: Unsafe because it has a self-loop (a cycle).
- **Device 2**: Unsafe because it leads to device 3, which is part of a cycle.
- **Device 3**: Unsafe because it leads to device 0, which is part of a cycle.
- **Device 4**: Safe because it is a terminal device and has no outgoing edges.

Safe Devices:

The only **safe device** is **device 4**.