332. Reconstruct Itinerary

Hard Depth-First Search Graph **Eulerian Circuit** Leetcode Link

Problem Description

The problem provides you with a list of flight tickets represented as pairs of airports, where each pair `[from_i, to_i]` indicates a flight departing from `from_i` and arriving at `to_i`. Your task is to use these tickets to reconstruct a trip itinerary starting from the airport "JFK". Since the man's journey starts from "JFK", the itinerary must also start with "JFK".

You should find an itinerary that uses all the given tickets exactly once. If there are multiple itineraries that satisfy the ticket usage, you need to choose the itinerary that is lexicographically smallest when the sequence of airports is considered as a single string. For example, an itinerary that begins with `[“JFK”, “LGA”]` is considered smaller than `[“JFK”, “LGB”]`.

The key points to consider for this problem are:

- The trip must start from "JFK".
- You must use each ticket exactly once.
- If multiple itineraries are possible, return the one that is lexicographically smallest.

Intuition

To solve this problem, we take inspiration from graph theory. We can represent the list of tickets as a directed graph where each airport is a vertex and each ticket is a directed edge from `from_i` to `to_i`. The objective is to find an Eulerian path through the graph, which is a path that visits every edge exactly once.

To ensure that we visit the smallest lexicographical path first, we sort the outgoing flights (edges) for each airport (vertex) in descending lexical order. We use a recursive depth-first search (DFS) algorithm, starting from "JFK", to build the itinerary. Whenever we reach an airport with no further outgoing flights, we add that airport to the itinerary. Since we're using a recursive approach, the itinerary is constructed in reverse, beginning with the last airport visited.

During the DFS, if there is a dead end (and because the graph has at least one valid itinerary, this dead end is not the starting airport "JFK"), we backtrack to the previous airport and continue the search. This process ensures that we use all tickets. After the DFS is completed, we reverse the constructed itinerary to get the correct order of travel.

This solution utilizes a greedy approach to always take the smallest lexicographical path first while ensuring all edges are visited, fitting the requirements of an Eulerian path where all tickets are used once.

Solution Approach

The implementation of the solution follows a depth-first search (DFS) methodology using a stack that operates on a recursive call stack. The data structure and pattern used are as follows:

- **Graph Representation**: We use a `defaultdict` of lists to represent the directed graph. Each key in this dictionary is an airport, and its value is a list of airports to which there are outbound flights.
- **Sorting**: Before we start the DFS, we sort the tickets in reverse order. This ensures that while we add destinations to the graph, they are stored in reverse lexicographical order. Since Python lists function as stacks and we use `.pop()` to retrieve the last element, we will get the smallest element first due to this initial reverse sorting.
- **Depth-First Search**: We create a recursive function `dfs` that takes an airport as input and explores all possible itineraries from that airport using DFS. If the current airport has no more destinations to visit (`graph[airport]` is empty), it means we've hit a dead end or finished one valid path, and we add this airport to the itinerary.
- **Building Itinerary**: As we are backtracking (when the recursion unwinds), we append the airports to the `itinerary`. This constructs the itinerary in reverse, as the last airport in our journey will be the first to be added to the itinerary once all its tickets have been used up.
- **Reversing Itinerary**: After the DFS completes, we get an itinerary that is in reverse order. To correct this, we simply reverse the itinerary before returning it to get the itinerary in the order of travel.

In this approach, we leverage the call stack of the recursive function to serve as a temporal data structure to hold our path. We guarantee that the path we're constructing is valid and that we are visiting the flights in the smallest lexicographical order as required.

Here's the relevant code breakdown:

- `graph = defaultdict(list)` is used to construct the graph.
- `for src, dst in sorted(tickets, reverse=True): graph[src].append(dst)` is used for populating the graph with sorted tickets.
- `itinerary = []` is the list that will eventually contain the itinerary, once reversed.
- `dfs("JFK")` initiates the DFS from "JFK".
- `itinerary[::-1]` reverses the itinerary to return the final solution.

The elegance of the solution lies in how the graph is constructed and traversed, ensuring all tickets are used exactly once and the lexicographically smallest path is found without the need for post-processing or backtracking beyond what's natural to DFS.

Example Walkthrough

Consider the following set of tickets representing flights:

```
[["JFK","KUL"],["JFK","NRT"],["NRT","JFK"]]
```

We start by representing these tickets as a directed graph:

```
JFK -> KUL
JFK -> NRT
NRT -> JFK
```

Firstly, we need to sort the destinations in descending lexicographical order for each airport.

1. For "JFK", the destinations "KUL" and "NRT" are sorted as ["NRT", "KUL"].
2. "NRT" only has one destination: "JFK".

Our graph now looks like this, with destinations sorted:

```
JFK -> ["NRT", "KUL"]
NRT -> ["JFK"]
```

Now, we apply the depth-first search (DFS) approach starting from "JFK".

- Begin at "JFK". Available destinations are "NRT" and "KUL", we choose "NRT" as it is lexicographically smaller.
- At "NRT", the only destination is "JFK". We go there.
- Back at "JFK", the only remaining destination is "KUL". We go there.
- Now, we are at "KUL" with no further destinations to explore.

As we reach each "end" of a path, we add the airport to the itinerary. Since "KUL" has no further destinations, "KUL" is added to the itinerary first.

The constructed itinerary in reverse is:

```
["KUL", "JFK", "NRT", "JFK"]
```

Finally, we reverse the itinerary to get the correct travel order:

```
["JFK", "NRT", "JFK", "KUL"]
```

This itinerary has utilized all tickets exactly once and is lexicographically the smallest, fulfilling all the problem requirements.

Python Solution

```python
from collections import defaultdict

class Solution:
    def findItinerary(self, tickets: List[List[str]]) -> List[str]:
        # Create a default dictionary to represent the graph where each
        # key is a departure airport and values are lists of arrival airports
        graph = defaultdict(list)

        # Sort tickets in reverse order, since we will later pop destinations from the end of the list
        # which is more efficient than pop(0), and we need the smallest lexical order for equal paths.
        for departure, arrival in sorted(tickets, reverse=True):
            graph[departure].append(arrival)

        # Initialize a list to keep track of the itinerary
        itinerary = []

        # Recursive Depth First Search function
        def dfs(airport):
            # While there are destinations to visit from current airport
            while graph[airport]:
                # Visit the destination by doing DFS on the last airport in the list
                # (since it's reversed sorted, smallest lexical goes last)
                dfs(graph[airport].pop())
            # Append the airport to itinerary after visiting all destinations
            itinerary.append(airport)

        # Begin DFS from 'JFK' which is the starting airport
        dfs("JFK")

        # The itinerary is in reverse order due to the nature of DFS, so reverse it before returning
        return itinerary[::-1]

# Note: List needs to be imported from typing to use List type annotations.
from typing import List

```

Java Solution

```java
import java.util.*;

public class Solution {

    // Performs Depth-First Search recursively on the graph.
    private void dfs(Map<String, PriorityQueue<String>> graph, LinkedList<String> itinerary, String currentNode) {
        // Retrieve the neighbors (destinations) for the current node
        PriorityQueue<String> neighbors = graph.get(currentNode);

        // Process all neighbors
        while (neighbors != null && !neighbors.isEmpty()) {
            // Remove the top neighbor (next destination) from the queue and perform dfs
            String nextNode = neighbors.poll();
            dfs(graph, itinerary, nextNode);
        }

        // All flights from the current node have been used; add to the itinerary
        itinerary.addFirst(currentNode);
    }

    public List<String> findItinerary(List<List<String>> tickets) {
        // Create a graph from the list of tickets. Each node maps to a priority queue of destinations
        Map<String, PriorityQueue<String>> graph = new HashMap<>();
        for (List<String> ticket : tickets) {
            String src = ticket.get(0);  // Source airport
            String dest = ticket.get(1); // Destination airport

            // Initialize the priority queue if it does not already exist
            graph.computeIfAbsent(src, k -> new PriorityQueue<>()).add(dest);
        }

        // This linked list will store the itinerary in reverse order
        LinkedList<String> itinerary = new LinkedList<>();

        // Start DFS from the "JFK" node
        dfs(graph, itinerary, "JFK");

        // The LinkedList itinerary already contains the nodes in reverse due to the use of addFirst().
        // Therefore, there's no need to reverse the itinerary at the end.
        return itinerary;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
#include <queue>
using namespace std;

class Solution {
public:
    // Function to find the itinerary from a list of tickets
    vector<string> findItinerary(vector<vector<string>>& tickets) {
        // Graph to hold nodes and edges where the edges are sorted in a min-heap (priority queue)
        unordered_map<string, priority_queue<string, vector<string>, greater<string>>> graph;
        vector<string> result; // This will hold the final itinerary

        // Build the graph
        for (const auto& ticket : tickets) {
            graph[ticket[0]].push(ticket[1]);
        }

        // DFS to build the itinerary
        findItineraryDFS(graph, result, "JFK");

        // The DFS result is in reverse order, reverse it to get the correct itinerary
        reverse(result.begin(), result.end());

        return result;
    }

    // Helper DFS function to add the itinerary in reverse order
    void findItineraryDFS(unordered_map<string, priority_queue<string, vector<string>, greater<string>>>& graph, vector<string>& resu
        // Iterate as long as there are destinations from the current node
        while (!graph[currentNode].empty()) {
            // Get the next destination in lexical order
            string nextDestination = graph[currentNode].top();
            graph[currentNode].pop(); // Remove the edge to the next destination
            findItineraryDFS(graph, result, nextDestination); // DFS with the next node
        }
        // Add the current node to the result list
        result.push_back(currentNode);
    }
};

```

Typescript Solution

```typescript
import { PriorityQueue } from 'typescript-collections';

// The graph will map nodes to a priority queue of destinations.
const graph: Record<string, PriorityQueue<string>> = {};
// An array to hold the final itinerary.
let itinerary: string[] = [];

// Function to find the itinerary from a list of tickets.
function findItinerary(tickets: [string, string][]): string[] {
    // Build the graph with destinations sorted in lexical order.
    tickets.forEach(([from, to]) => {
        if (!graph[from]) {
            graph[from] = new PriorityQueue<string>((a, b) => a < b ? -1 : (a > b ? 1 : 0));
        }
        graph[from].enqueue(to);
    });

    // Perform the depth-first search starting from "JFK".
    dfs("JFK");

    // The DFS result is in reverse order, so it's reversed to get the correct itinerary.
    itinerary.reverse();

    return itinerary;
}

// Helper DFS function to recursively build the itinerary.
function dfs(node: string): void {
    // Continue until all destinations from the current node have been visited.
    let destinations = graph[node];
    while (destinations && !destinations.isEmpty()) {
        // Visit the next destination in lexical order.
        const nextDestination = destinations.dequeue();
        dfs(nextDestination);
    }
    // Add the current node to the result list.
    itinerary.push(node);
}

// Please note, the PriorityQueue class used in this sample is hypothetical and assumed to be from
// an npm package like 'typescript-collections'. The actual implementation may differ.

```

Time and Space Complexity

Time Complexity

The time complexity of the algorithm can be broken down into the following parts:

1. Sorting the `tickets` list, which has a complexity of `O(T log T)`, where `T` is the number of tickets.
2. Building the graph takes `O(T)` time, as each ticket (`src`, `dst` pair) is considered exactly once when adding to the graph.
3. The depth-first search (DFS) uses recursion and, in the worst case, visits each vertex and edge once. Since each ticket represents a directed edge in the graph, the complexity for the DFS part is `O(T)`.

Combining these gives us a total time complexity of `T(sorting) + T(building graph) + T(DFS) -> O(T log T) + O(T) + O(T)`, which simplifies to `O(T log T)` due to the dominance of the sorting complexity.

Space Complexity

The space complexity is determined by:

1. The space needed for the `graph` data structure, which can hold at most `2T` entries (because each ticket is a directed edge and might introduce up to two vertices).
2. The `itinerary` list that, in the worst-case, holds `O(T)` entries, as each ticket will be converted into a step in the itinerary.
3. The depth of the call stack for the recursive DFS function, which, in the worst case, could be `O(T)` if the graph forms a straight line.

The `graph` size dominates and offers a space complexity of `O(T)`. The call stack and the final itinerary list also contribute `O(T)` each. Thus, the total space complexity is `O(T) + O(T) + O(T)`, which simplifies to `O(T)`, since all are linear with respect to the number of tickets.