Intuition

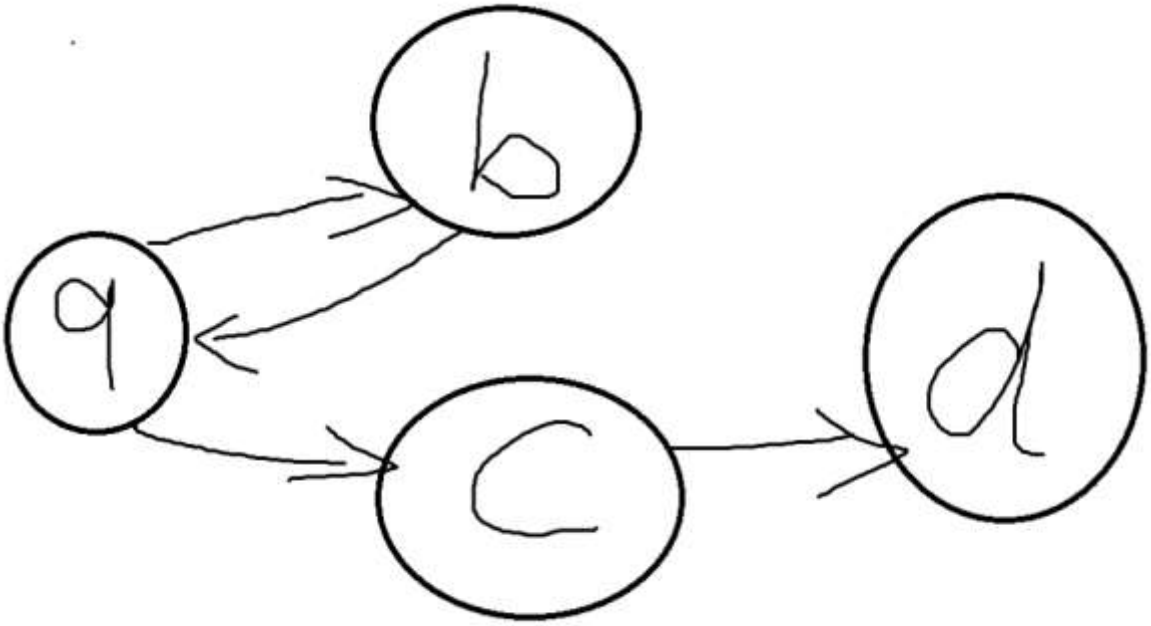
For our airports problem we have two **crucial points** we must put in mind: We must use each ticket once and only once, there is atleast one valid itinerary as a solution of the problem.

What can we **conclude** from that? 🤔
The graph that we will build out of tickets list is something called **Semi Euler Graph** and the path that we want is called **Euler Path**.
I won't talk about them in depth because they are advanced topics even for me. 😂 but I will give a simple Intuition.

Let's call some graph termenolgies that we want later.

- **In-degree** for a node: number of edges that points to a specific node.
- **Out-degree** for anode: number of edges coming out from a node.

Let's see an **exmaple**:



Node	In-degree	Out-degree
a	1	2
b	1	1
c	1	1

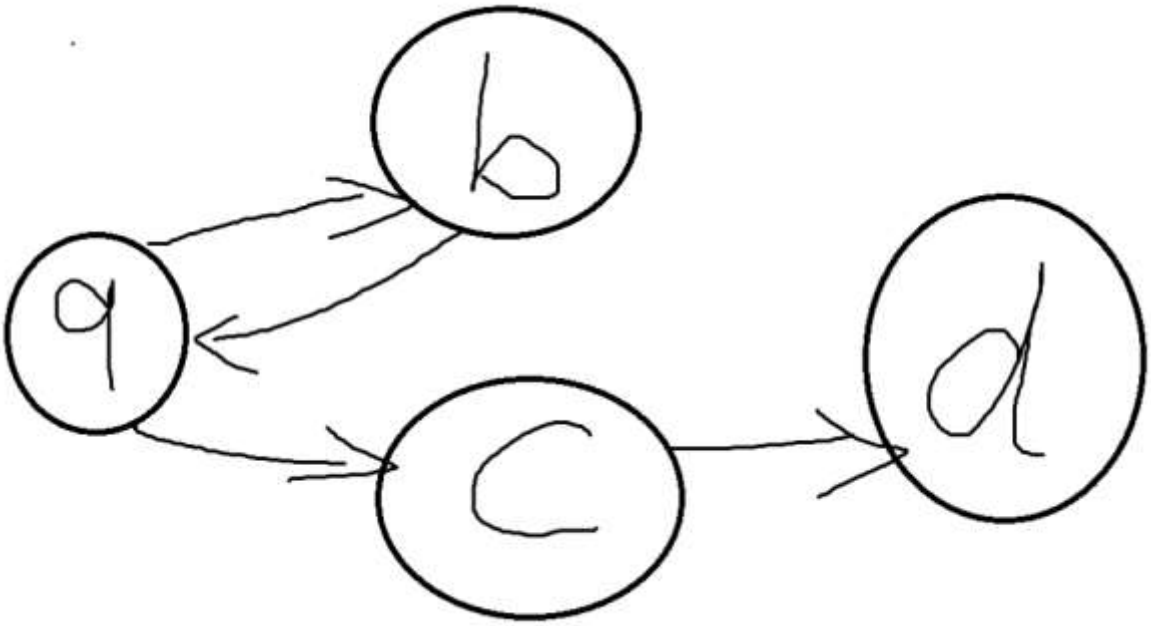
Node	In-degree	Out-degree
d	1	0

a **Semi Euler Graph** is a graph that has **Euler Path** which mean there is a path from starting node to an ending node and we will use all edges once and only once.
Huh looks familiar 🧐 ... Yes ITS OUR PROBLEM!! 🤯

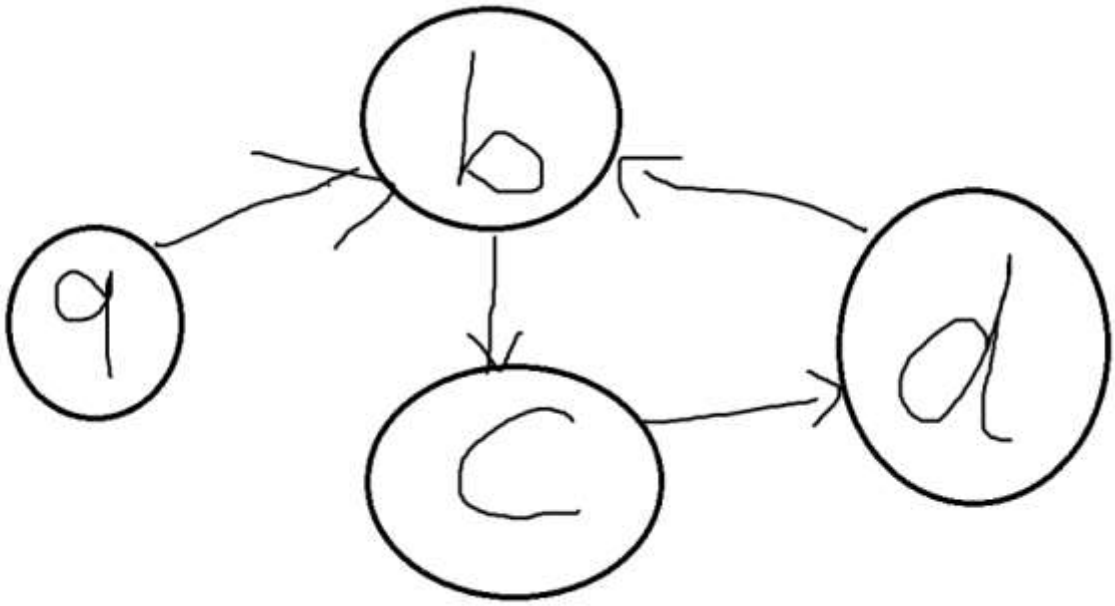
For a **directed graph** to be a **Semi Euler Graph** it must meet some conditions:

- **In degree** for all nodes = **Out degree** for all nodes
- Except for two node:
    - Starting node should have Out-degree = In degree + 1
    - Ending node should have In-degree = Out-degree + 1

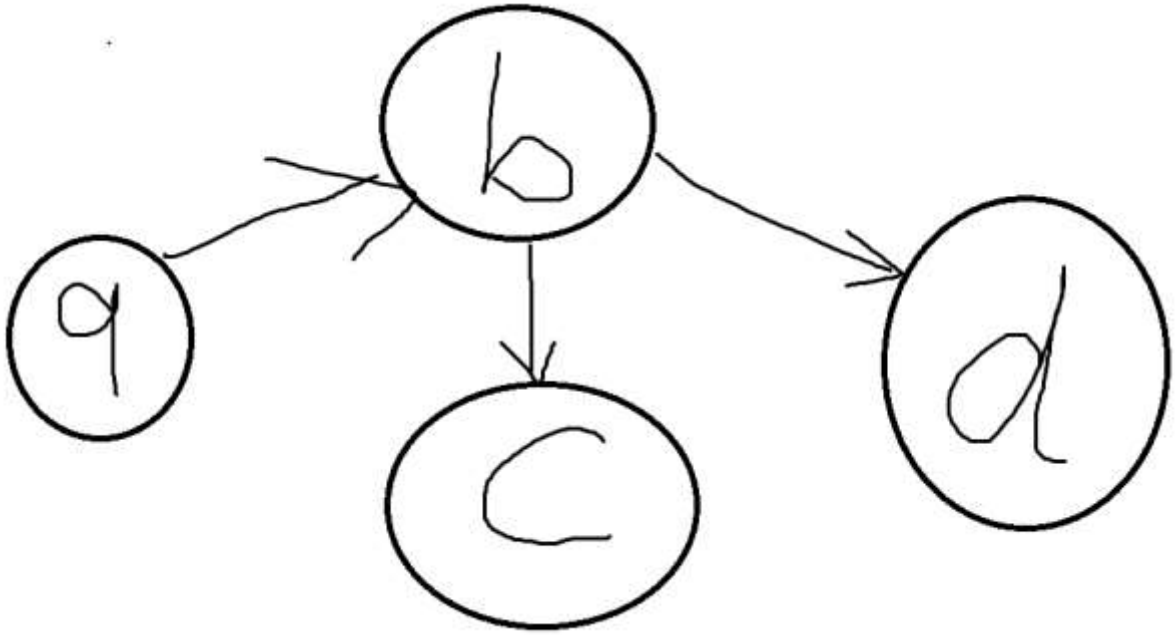
Let's see some examples for our original problem.



In this example, we can see that d airport meets the condition for an ending node and a meets the condition for a starting node.
An answer will be (a, b, c, d)



In this example, we can see that b airport meets the condition for an ending node and a meets the condition for a starting node.
An answer will be also (a, b, c, d, b)



In this example, we can see that the nodes didn't meet the condition to be an**Semi Euler Graph**.
And if you tried to put it as a test case it will tell you invalid input.

Okay, I think we have something here 🤯
The Pseudo Code for finding Euler Path is easy the only editing we will do it to sort the edges for the graph to be in the smallest lexical order.

and for this problem, we will find for each test case that JFK meets the conditions to be a starting node.

And that's it the solution for our problem only a **Semi Euler Graph** 😂

Approach

1. DFS Recursive

Steps

- Initialize a flightGraph as a dictionary (map) to represent flights and an itinerary list to store the final travel sequence.
- Iterate through each ticket and populate the flightGraph dictionary.
- Sort the destinations for each airport in reverse order to visit **lexical smaller** destinations first.
- Start the DFS traversal from the JFK airport.
- Using the **depth-first search** (DFS) method called dfs that takes an airport as input and recursively explores its destinations while **maintaining lexical order**. It adds the visited airports to the itinerary list.
- **Reverse** the itinerary list to get the correct travel order.
- **Return** the itinerary list as the final result.

Complexity

- **Time complexity:** O(N2log(N))O(N^2log(N))$O(N2log(N))$
  Since we loop over lists of destinations in the flight graph and sorts them. Sorting each list has a time complexity of O(E * log(E)), where E is the total number of edges (tickets). Since E can be at most N, this step has a time complexity of O(N * log(N)). and since we loop over N city then the total time complexity is O(N^2log(N)) where N is the number of airports.
- **Space complexity:** O(N+E)O(N+E)$O(N+E)$
  We are storing the Flight Graph which is represented using map of lists, which will have at most N keys (airports) and a total of E values (destinations). Therefore, the space complexity is O(N + E).

2. DFS Iterative

Steps

- Initialize a flightGraph as a dictionary (map) to represent flights and an itinerary list to store the final travel sequence.
- Iterate through each ticket and populate the flightGraph dictionary.
- Sort the destinations for each airport in reverse order to visit **lexical smaller** destinations first.
- **Start** with JFK as the initial airport and create a stack.
- While the stack is not empty:
    - **Explore** destinations:
        - Push the next destination onto the stack.
    - **Backtrack**:
        - Add the current airport to the travel itinerary.
        - Pop the current airport from the stack.
- **Reverse** the travel itinerary to get the correct order.
- **Return** the travel itinerary as the final result.

Complexity

- **Time complexity:** O(N2log(N))O(N^2log(N))$O(N2log(N))$
  Since we loop over lists of destinations in the flight graph and sorts them. Sorting each list has a time complexity of O(E * log(E)), where E is the total number of edges (tickets). Since E can be at most N, this step has a time complexity of O(N * log(N)). and since we loop over N city then the total time complexity is O(N^2log(N)) where N is the number of airports.
- **Space complexity:** O(N+E)O(N+E)$O(N+E)$
  We are storing the Flight Graph which is represented using map of lists, which will have at most N keys (airports) and a total of E values (destinations). Therefore, the space complexity is O(N + E).