Reconstruct Itinerary

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]`, reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK`. Thus, the itinerary must begin with `JFK`.

Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets may form at least one valid itinerary.

Example 1:
`tickets` = `[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`
Return `["JFK", "MUC", "LHR", "SFO", "SJC"]`.

Example 2:
`tickets` = `[["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]`
Return `["JFK","ATL","JFK","SFO","ATL","SFO"]`.
Another possible reconstruction is `["JFK","SFO","ATL","JFK","ATL","SFO"]`. But it is larger in lexical order.

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

Just Eulerian path. Greedy DFS, building the route backwards when retreating.

More explanation and example under the codes.

Iterative versions inspired by fangyang (I had only thought of recursion, d'oh).

Ruby

```ruby
def find_itinerary(tickets)
  tickets = tickets.sort.reverse.group_by(&:first)
  route = []
  visit = -> airport {
    visit[tickets[airport].pop()[1]] while (tickets[airport] || []).any?
    route << airport
  }
  visit["JFK"]
  route.reverse
end
```

Iterative version:

```ruby
def find_itinerary(tickets)
  tickets = tickets.sort.reverse.group_by(&:first)
  route, stack = [], ["JFK"]
  while stack.any?
    stack << tickets[stack[-1]].pop()[1] while (tickets[stack[-1]] || []).any?
    route << stack.pop()
  end
  route.reverse
end
```

Python

```python
def findItinerary(self, tickets):
    targets = collections.defaultdict(list)
    for a, b in sorted(tickets)[::-1]:
        targets[a] += b,
    route = []
    def visit(airport):
        while targets[airport]:
            visit(targets[airport].pop())
        route.append(airport)
    visit('JFK')
    return route[::-1]
```

Iterative version:

```
def findItinerary(self, tickets):
    targets = collections.defaultdict(list)
    for a, b in sorted(tickets)[::-1]:
        targets[a] += b,
    route, stack = [], ['JFK']
    while stack:
        while targets[stack[-1]]:
            stack += targets[stack[-1]].pop(),
        route += stack.pop(),
    return route[::-1]
```

Java

```
public List<String> findItinerary(String[][] tickets) {
    for (String[] ticket : tickets)
        targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1
]);
    visit("JFK");
    return route;
}

Map<String, PriorityQueue<String>> targets = new HashMap<>();
List<String> route = new LinkedList();

void visit(String airport) {
    while(targets.containsKey(airport) && !targets.get(airport).isEmpty())
        visit(targets.get(airport).poll());
    route.add(0, airport);
}
```

Iterative version:

```
public List<String> findItinerary(String[][] tickets) {
    Map<String, PriorityQueue<String>> targets = new HashMap<>();
    for (String[] ticket : tickets)
        targets.computeIfAbsent(ticket[0], k -> new PriorityQueue()).add(ticket[1
]);
    List<String> route = new LinkedList();
    Stack<String> stack = new Stack<>();
    stack.push("JFK");
    while (!stack.empty()) {
        while (targets.containsKey(stack.peek()) && !targets.get(stack.peek()).is
Empty())
            stack.push(targets.get(stack.peek()).poll());
        route.add(0, stack.pop());
    }
    return route;
}
```

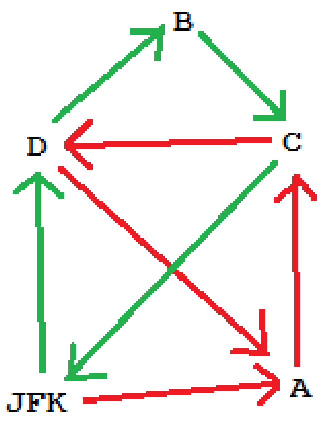
C++

```
vector<string> findItinerary(vector<pair<string, string>> tickets) {
    for (auto ticket : tickets)
        targets[ticket.first].insert(ticket.second);
    visit("JFK");
    return vector<string>(route.rbegin(), route.rend());
}

map<string, multiset<string>> targets;
vector<string> route;

void visit(string airport) {
    while (targets[airport].size()) {
        string next = *targets[airport].begin();
        targets[airport].erase(targets[airport].begin());
        visit(next);
    }
    route.push_back(airport);
}
```

Explanation

First keep going forward until you get stuck. That's a good main path already. Remaining tickets form cycles which are found on the way back and get merged into that main path. By writing down the path backwards when retreating from recursion, merging the cycles into the main path is easy - the end part of the path has already been written, the start part of the path hasn't been written yet, so just write down the cycle now and then keep backwards-writing the path.

Example:



From JFK we first visit JFK -> A -> C -> D -> A. There we're stuck, so we write down A as the end of the route and retreat back to D. There we see the unused ticket to B and follow it: D -> B -> C -> JFK -> D. Then we're stuck again, retreat and write down the airports while doing so: Write down D before B, then JFK before D, etc. When we're back from our cycle at D, the written route is D -> B -> C -> JFK -> D -> A. Then we retreat further along the original path, prepending C, A and finally JFK to the route, ending up with the route JFK -> A -> C -> D -> B -> C -> JFK -> D ->

A.

written by StefanPochmann original link here

Solution 2

```
class Solution {
public:
    vector<string> findItinerary(vector<pair<string, string>> tickets) {
        // Each node (airport) contains a set of outgoing edges (destination).
        unordered_map<string, multiset<string>> graph;
        // We are always appending the deepest node to the itinerary,
        // so will need to reverse the itinerary in the end.
        vector<string> itinerary;
        if (tickets.size() == 0){
            return itinerary;
        }
        // Construct the node and assign outgoing edges
        for (pair<string, string> eachTicket : tickets){
            graph[eachTicket.first].insert(eachTicket.second);
        }
        stack<string> dfs;
        dfs.push("JFK");
        while (!dfs.empty()){
            string topAirport = dfs.top();
            if (graph[topAirport].empty()){
                // If there is no more outgoing edges, append to itinerary
                // Two cases:
                // 1. If it searchs the terminal end first, it will simply get
                //    added to the itinerary first as it should, and the proper r
oute
                //    will still be traversed since its entry is still on the sta
ck.
                // 2. If it search the proper route first, the dead end route wil
l also
                //    get added to the itinerary first.
                itinerary.push_back(topAirport);
                dfs.pop();
            }
            else {
                // Otherwise push the outgoing edge to the dfs stack and
                // remove it from the node.
                dfs.push(*(graph[topAirport].begin()));
                graph[topAirport].erase(graph[topAirport].begin());
            }
        }
        // Reverse the itinerary.
        reverse(itinerary.begin(), itinerary.end());
        return itinerary;
    }
};
```

written by frederick2 original link here

Solution 3

Noticed some folks are using Hierholzer's algorithm to find a Eulerian path.

My solution is similar, considering this passenger has to be physically in one place before move to another airport, we are considering using up all tickets and choose lexicographically smaller solution if in tie as two constraints.

Thinking as that passenger, the passenger choose his/her flight greedy as the lexicographical order, once he/she figures out go to an airport without departure with more tickets at hand. the passenger will push current ticket in a stack and look at whether it is possible for him/her to travel to other places from the airport on his/her way.

Please let me know if you have any suggestions.

```
    public List<String> findItinerary(String[][] tickets) {
        List<String> ans = new ArrayList<String>();
        if(tickets == null || tickets.length == 0) return ans;
        Map<String, PriorityQueue<String>> ticketsMap = new HashMap<>();
        for(int i = 0; i < tickets.length; i++) {
            if(!ticketsMap.containsKey(tickets[i][0])) ticketsMap.put(tickets[i][
0], new PriorityQueue<String>());
            ticketsMap.get(tickets[i][0]).add(tickets[i][1]);
        }

        String curr = "JFK";
        Stack<String> drawBack = new Stack<String>();
        for(int i = 0; i < tickets.length; i++) {
            while(!ticketsMap.containsKey(curr) || ticketsMap.get(curr).isEmpty()
) {
                drawBack.push(curr);
                curr = ans.remove(ans.size()-1);
            }
            ans.add(curr);
            curr = ticketsMap.get(curr).poll();
        }
        ans.add(curr);
        while(!drawBack.isEmpty()) ans.add(drawBack.pop());
        return ans;
    }
```

written by fangyang original link here