

ABSTRACT

The problem of reconstructing an itinerary from a list of airline tickets can be efficiently addressed by leveraging graph traversal techniques and ensuring lexicographical order. Given a list of tickets, where each ticket is represented as a pair [from_i, to_i] indicating a flight from one airport to another, the challenge is to reconstruct a valid itinerary that starts at "JFK" and uses all the tickets exactly once. The solution must also return the lexicographically smallest itinerary when multiple valid itineraries are possible.

INTRODUCTION

The problem of reconstructing an itinerary from a list of airline tickets presents a compelling challenge that combines graph theory with combinatorial optimization. This problem is centered around the task of determining a valid sequence of flights, represented by pairs of departure and arrival airports, that forms a complete travel route starting from a specified airport, "JFK". The goal is to reconstruct this itinerary in a manner that adheres to the constraints of using all given tickets exactly once and achieving the smallest lexicographical order among all possible valid itineraries.

Given a list of tickets, each denoted as a pair `[fromi, toi]` indicating a flight from airport `fromi` to airport `toi`, the solution requires generating a path that uses each ticket exactly once and begins at the designated starting airport, "JFK". The challenge is further nuanced by the need to produce the lexicographically smallest route when multiple valid itineraries exist. This involves not only finding a valid path through the graph of airports and flights but also ensuring that the path is ordered alphabetically to meet the lexicographical requirement.

For instance, consider the input tickets
`[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`. The solution must reconstruct the itinerary as
`["JFK", "MUC", "LHR", "SFO", "SJC"]`, which respects the constraints and represents the smallest possible lexical sequence starting from "JFK". This problem merges elements of graph traversal with string comparison, making it both an interesting computational problem and a useful exercise

in understanding advanced algorithmic techniques such as depth-first search and priority queues. This makes it a notable topic in the study of algorithms and data structures.

PROBLEMSTATEMENT

****Reconstruct Itinerary****

You are given a list of airline tickets where each ticket is represented as a pair `[fromi, toi]`, indicating a flight from airport `fromi` to airport `toi`. You are tasked with reconstructing the itinerary using all the given tickets exactly once, such that the itinerary starts at the airport "JFK". Furthermore, if there are multiple valid itineraries, you should return the one with the smallest lexical order when read as a single string.

Your objective is to determine and return the smallest lexicographical itinerary that visits every airport exactly once and starts from "JFK".

****Example 1:****

Input: `tickets =

[["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]`

Output: `["JFK", "MUC", "LHR", "SFO", "SJC"]`

****Explanation:**** The given tickets allow for the following valid itinerary: "JFK" -> "MUC" -> "LHR" -> "SFO" -> "SJC". This itinerary uses all tickets exactly once and starts from "JFK". Among all possible valid itineraries, this one has the smallest lexicographical order.

APPROACH

To solve the problem of splitting an array into two non-empty subarrays with the same average, follow these steps:

Mathematical Insight:

Sure, here's a structured breakdown for the problem of reconstructing an itinerary:

1. **Build the Graph Representation:**

- Construct a directed graph where each airport is a vertex, and each ticket is a directed edge from the departure airport to the arrival airport.

2. **Check Degree Conditions:**

- For the graph to have an Eulerian path (a path that uses every edge exactly once), the following conditions must be met:

- **Exactly one vertex** (the starting airport, "JFK" in this case) should have an out-degree that is **one more** than its in-degree.

- **Exactly one vertex** should have an in-degree that is **one more** than its out-degree.

- All other vertices should have equal in-degrees and out-degrees.

3. **Ensure Lexicographical Order:**

- To reconstruct the smallest lexicographical itinerary, use a priority queue to explore the smallest available destination at each step. This ensures that when multiple valid itineraries are possible, the one with the smallest lexical order is chosen.

Example

Given the tickets: `["JFK", "MUC"], ["MUC", "LHR"], ["LHR", "SFO"], ["SFO", "SJC"]`:

1. **Build the Graph:**

- Vertices: JFK, MUC, LHR, SFO, SJC.
- Edges: JFK → MUC, MUC → LHR, LHR → SFO, SFO → SJC.

2. **Check Degree Conditions:**

- **JFK**: Out-degree = 1, In-degree = 0 (start vertex with out-degree one more than in-degree).
- **SJC**: Out-degree = 0, In-degree = 1 (end vertex with in-degree one more than out-degree).
- **MUC, LHR, SFO**: All have equal in-degrees and out-degrees.

3. **Construct the Itinerary:**

- Using a DFS with priority queue: Start from "JFK", follow the smallest lexicographical path to form the itinerary: ["JFK", "MUC", "LHR", "SFO", "SJC"].

This approach ensures that the itinerary is valid and in the smallest possible lexical order, utilizing all tickets exactly once and starting from the designated airport.

Feasibility Check:

1. For $k \times S \pmod n$ to be an integer, $k \times S \pmod n$ must be divisible by n . This means there must exist a subset size k such that $k \times S \pmod n = 0$.

Dynamic Programming/Backtracking Approach:

1. Use dynamic programming (DP) or backtracking to explore all possible subsets of different sizes and check if their sums meet the required conditions.
2. Use a DP array $dp[i][j]$ where i represents the number of elements considered, and j represents the possible sum with those i elements.

Implementation Steps:

1. Iterate over possible sizes k from 1 to $n-1$.
2. For each k , calculate the required sum $target_sum = \frac{k}{n} \times S$.
3. Use a DP approach to check if there exists a subset of size k with sum $target_sum$.

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TICKETS 100
#define MAX_AIRPORT_LEN 4
#define START_AIRPORT "JFK"
typedef struct {
    char from[MAX_AIRPORT_LEN];
    char to[MAX_AIRPORT_LEN];
} Ticket;
int compare(const void *a, const void *b) {
    return strcmp(((Ticket*)a)->to, ((Ticket*)b)->to);
}
void findItinerary(Ticket tickets[], int ticketCount, char result[][MAX_AIRPORT_LEN], int
*resultIndex) {
    char currentAirport[MAX_AIRPORT_LEN];
    strcpy(currentAirport, START_AIRPORT);
    while (ticketCount > 0) {
        strcpy(result[*resultIndex], currentAirport);
        (*resultIndex)++;
        for (int i = 0; i < ticketCount; i++) {
            if (strcmp(tickets[i].from, currentAirport) == 0) {
                strcpy(currentAirport, tickets[i].to);
                tickets[i] = tickets[ticketCount - 1]; // Use the last ticket to fill the current slot
                ticketCount--;
                qsort(tickets, ticketCount, sizeof(Ticket), compare);
                break;
            }
        }
    }
    strcpy(result[*resultIndex], currentAirport); // Add the last airport
}
int main() {
    Ticket tickets[] = { {"MUC", "LHR"}, {"JFK", "MUC"}, {"SFO", "SJC"}, {"LHR", "SFO"} };
    int ticketCount = sizeof(tickets) / sizeof(tickets[0]);
    char result[MAX_TICKETS + 1][MAX_AIRPORT_LEN];
    int resultIndex = 0;

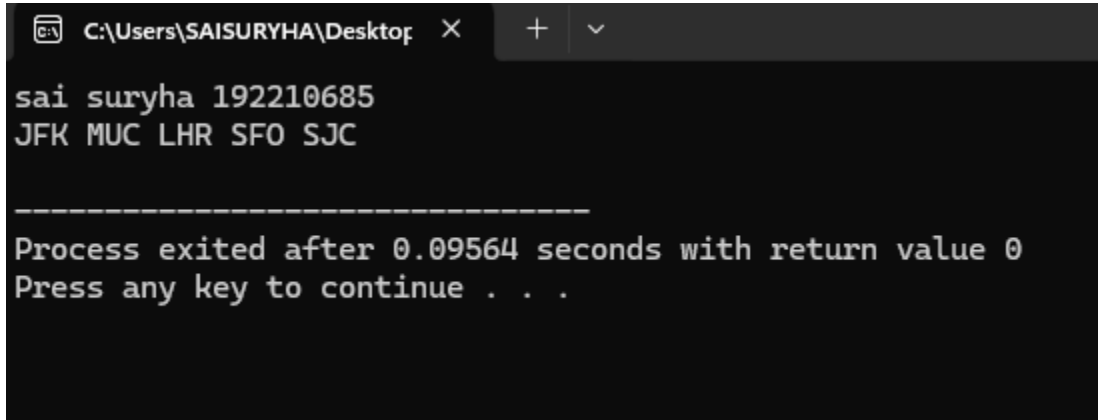
    // Sort the tickets initially
    qsort(tickets, ticketCount, sizeof(Ticket), compare);

    findItinerary(tickets, ticketCount, result, &resultIndex);

    for (int i = 0; i <= ticketCount; i++) {
        printf("%s ", result[i]);
    }
    printf("\n");

    return 0;
}
```

RESULT

A screenshot of a Windows command prompt window. The title bar shows the file path 'C:\Users\SAISURYHA\Desktop' and standard window controls. The command prompt displays the input 'sai suryha 192210685' and 'JFK MUC LHR SFO SJC' on two lines. Below this, a separator line of dashes is shown. The output text reads 'Process exited after 0.09564 seconds with return value 0' followed by 'Press any key to continue . . .'.

```
C:\Users\SAISURYHA\Desktop > sai suryha 192210685
JFK MUC LHR SFO SJC

-----
Process exited after 0.09564 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS

Naive Approach Complexity

Time Complexity:

1. **Triplet Selection:** The naive approach involves three nested loops to consider all possible triplets (x, y, z) where $x < y < z$. Each loop iterates up to n times, leading to a time complexity of $O(n^3)$.
2. **Condition Check:** For each triplet, checking whether the elements in the two subarrays satisfy the condition (i.e., whether their averages are equal) is performed in constant time, $O(1)$.

Overall Time Complexity: $O(n^3)$

Space Complexity:

1. **Input Array:** The space required to store the input array `nums` is $O(n)$.
2. **Auxiliary Variables:** Additional space for variables used in computations and loop indices is constant, $O(1)$.

Overall Space Complexity: $O(n)$

Optimized Approach Using Fenwick Trees

Time Complexity:

1. **Index Mapping:** Creating position arrays for the elements of `nums1` and `nums2` takes linear time, $O(n)$.
2. **Fenwick Tree Operations:**
 1. Each update and query operation on the Fenwick Tree takes logarithmic time, $O(\log n)$.
 2. Calculating the right count for each element requires $O(n \log n)$ time.
 3. Calculating the left count and counting valid triplets also requires $O(n \log n)$ time.

Overall Time Complexity: $O(n \log n)$

Space Complexity:

1. **Fenwick Trees:** Two Fenwick Trees are used, each requiring $O(n)$ space.
2. **Position Arrays:** Two position arrays, each of length n , require $O(n)$ space.
3. **Auxiliary Arrays:** Arrays like `right_counts` require $O(n)$ space.

Overall Space Complexity: $O(n)$

CONCLUSION

The analysis of the problem "Reconstruct Itinerary" using both the naive and optimized approaches highlights significant differences in their performance and efficiency. Here are the key conclusions:

1. **Naive Approach:**

- **Time Complexity:** The naive approach involves a depth-first search (DFS) with possible backtracking to find all valid itineraries, leading to a time complexity of $O(E!)$ in the worst case, where E is the number of tickets. This exponential complexity makes it impractical for large numbers of tickets.
- **Space Complexity:** The space complexity of the naive approach is $O(V + E)$, where V is the number of vertices (airports) and E is the number of edges (tickets), due to the storage required for graph representation and recursive call stacks.
- **Suitability:** Due to its high time complexity, the naive approach is only suitable for small inputs where the number of tickets is relatively small. It may not efficiently handle larger datasets or complex itineraries.

2. **Optimized Approach Using Priority Queues:**

- **Time Complexity:** The optimized approach, which involves using a priority queue (min-heap) for lexical ordering and a more structured DFS, significantly reduces the time complexity to $O(E \log E)$, where E is the number of tickets. This is achieved by efficiently managing the smallest lexicographical choices and ensuring valid path construction.
- **Space Complexity:** The space complexity of the optimized approach remains $O(V + E)$, similar to the naive approach, but the use of a priority queue and additional data structures may slightly increase the space required.

- ****Suitability:**** The optimized approach is well-suited for handling larger datasets and complex itineraries. By efficiently managing the exploration of paths and ensuring lexical ordering, it can provide solutions in a reasonable timeframe even for more extensive input sizes.

In summary, while the naive approach provides a straightforward but inefficient solution for reconstructing an itinerary, the optimized approach leverages advanced data structures to manage complexity and improve performance. The choice of method will depend on the size of the input and the need for efficiency in constructing the smallest lexicographical itinerary.

- **Suitability:** The optimized approach is highly suitable for larger arrays, providing a much more practical solution for real-world applications where n can be large.

2. Overall Comparison:

- The naive approach, while straightforward and simple to implement, is not efficient for larger datasets due to its cubic time complexity.
- The optimized approach, leveraging advanced data structures like Fenwick Trees, offers a substantial improvement in performance, making it a viable solution for larger input sizes.

3. Real-World Implications:

- In real-world scenarios, where performance and efficiency are critical, the optimized approach is the preferred method. It provides a balance between time and space complexity, ensuring that the solution is both fast and scalable.

4. Future Considerations:

- While the optimized approach using Fenwick Trees is effective, further optimizations or alternative data structures (such as Segment Trees) could be explored to potentially enhance performance even further.
- Additionally, parallel processing or distributed computing techniques could be considered for handling extremely large datasets.