Reconstruct Itinerary

Reconstructing an itinerary is an algorithmic problem that involves finding the correct order of flights to complete a journey given a set of flight data.

AIRLINE NAME BOARDING PASS

Passenger Name: JOHN DOE 1
Flight Number: AA 1387 4
Class: F 5
Date: 12 DEC22
Time: 0630
From: NEW YORK
To: PARIS
Seat: 54 J 7
Gate: S 9
PNR: 2 1234TH
Boarding Group: GROUP 4 8
SSSS S/O SPTC 6
©SOPHIA CA TRAVEL

Passenger Name: JOHN DOE
From: NEW YORK
To: PARIS
Flight Number AA 1387
Class: F
Seat: 54 J
Gate: S

Overview of the Problem

Imagine you're given a list of flights, each with a departure and arrival city. Your task is to reconstruct the optimal itinerary by determining the correct order of these flights to visit all cities exactly once, starting and ending at a specific city.

1 **Input**

A list of flights, each represented by a departure city and arrival city.

2 **Output**

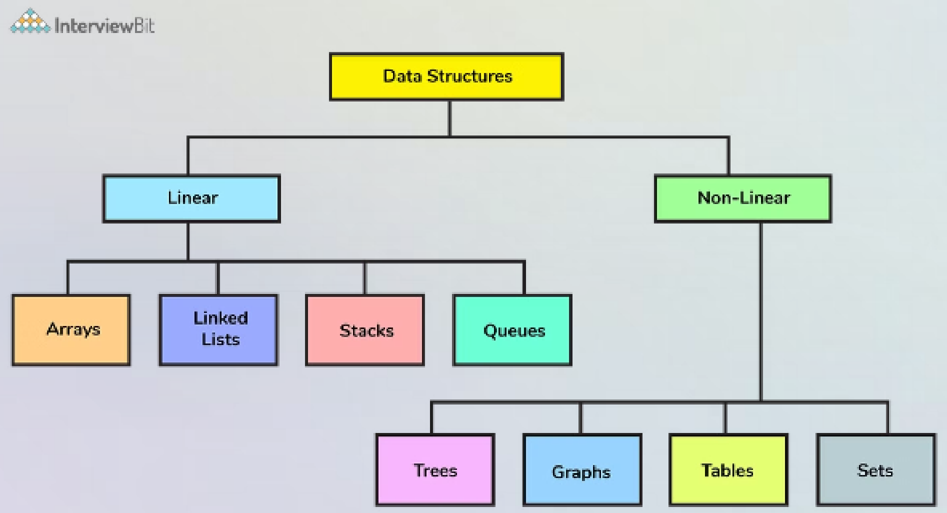
The reconstructed itinerary, a list of cities visited in the correct order.

3 **Constraints**

The itinerary must start and end at a specific city, and each city must be visited exactly once.

4 **Complexity**

The challenge lies in finding an efficient algorithm to reconstruct the itinerary while satisfying the constraints.



Input Data Structure

The input data for this problem can be represented using a graph data structure. Each city is a node, and each flight is an edge connecting two nodes.

Data Structure	Description
Graph	Represents the flights and cities. Nodes are cities, and edges are flights.
Edge	Represents a flight, with departure and arrival city information.

Approach to the Problem

The most common approach to solving this problem is using a Depth-First Search (DFS) algorithm. The idea is to traverse the graph starting from the departure city, exploring all possible paths until we find a path that visits all cities exactly once and ends at the destination city.

1 **Start at Departure City**

Begin the traversal from the specified departure city.

2 **Explore Flight Connections**

For each city, explore all available flights (edges) connecting to other cities.

3 **Backtrack and Continue**

If a path leads to a dead end (no further flights) or a city is visited twice, backtrack to the previous city and explore other flight options.

Algorithm Implementation

The algorithm uses recursion to traverse the graph. It keeps track of visited cities and the current path being explored. The algorithm backtracks when it reaches a dead end or a city is visited twice.

Initialization

Initialize the graph data structure, the starting city, and an empty list to store the reconstructed itinerary.

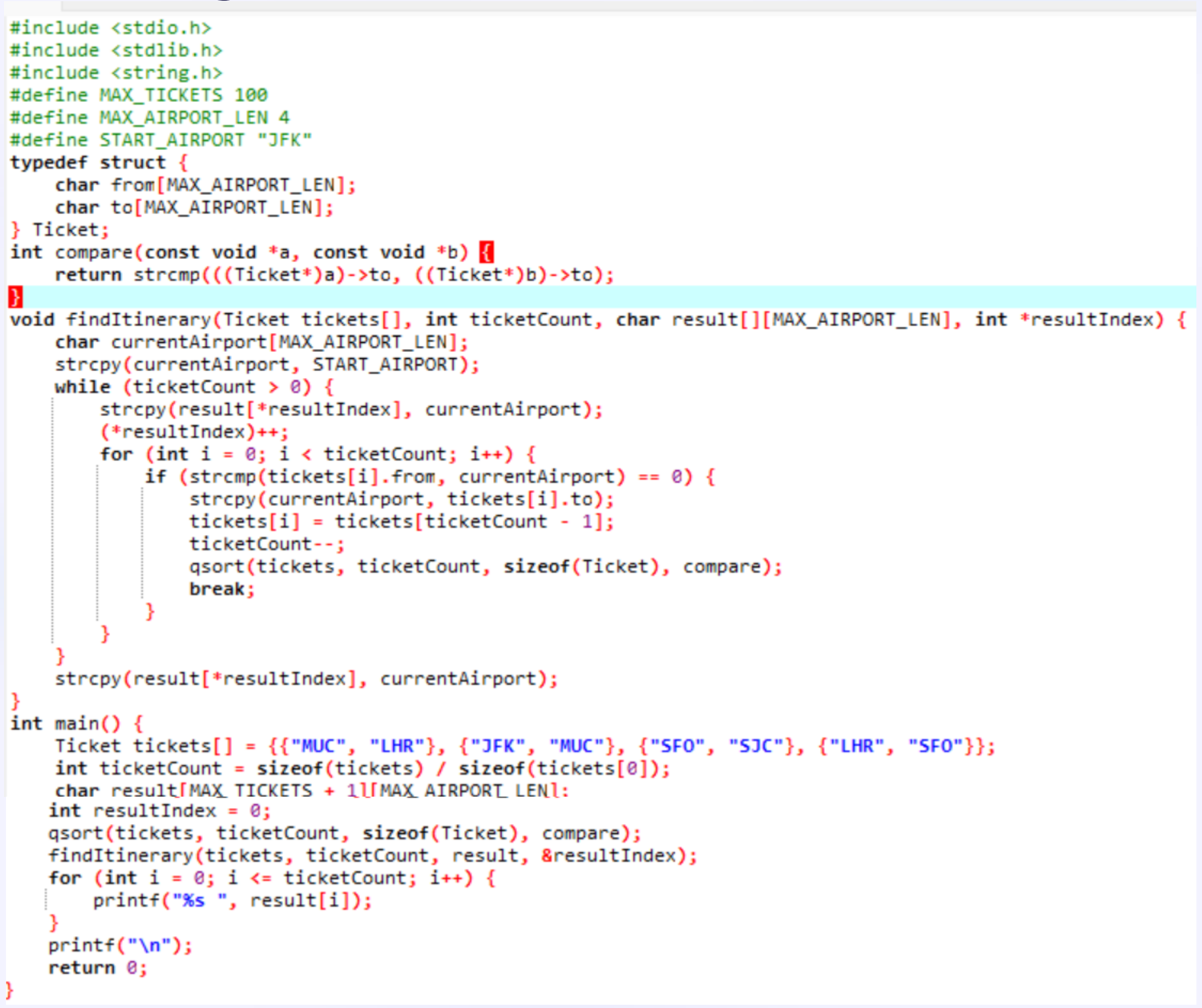
Depth-First Search (DFS)

Implement a recursive DFS function that takes the current city, visited cities, and the current path as input. The function explores all possible flights from the current city, and if a valid itinerary is found, it returns it.

Reconstruct Itinerary

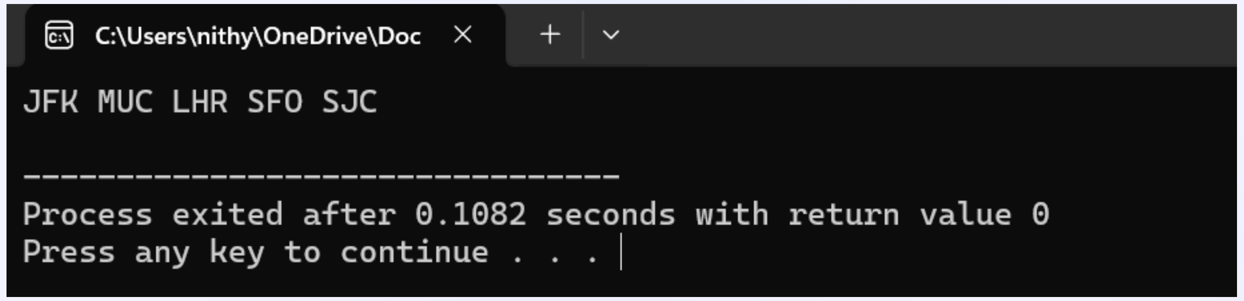
Call the DFS function with the starting city, an empty set of visited cities, and an empty path. The algorithm returns the reconstructed itinerary if it exists, otherwise, it returns an empty list.

Coding and Screenshot



```c
#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];
                ticketCount--;
                qsort(tickets, ticketCount, sizeof(Ticket), compare);
                break;
            }
        }
    }
    strcpy(result[*resultIndex], currentAirport);
}
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;
    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;
}
```

Output:



```
JFK MUC LHR SFO SJC

--------------------------------
Process exited after 0.1082 seconds with return value 0
Press any key to continue . . .
```

Time and Space Complexity

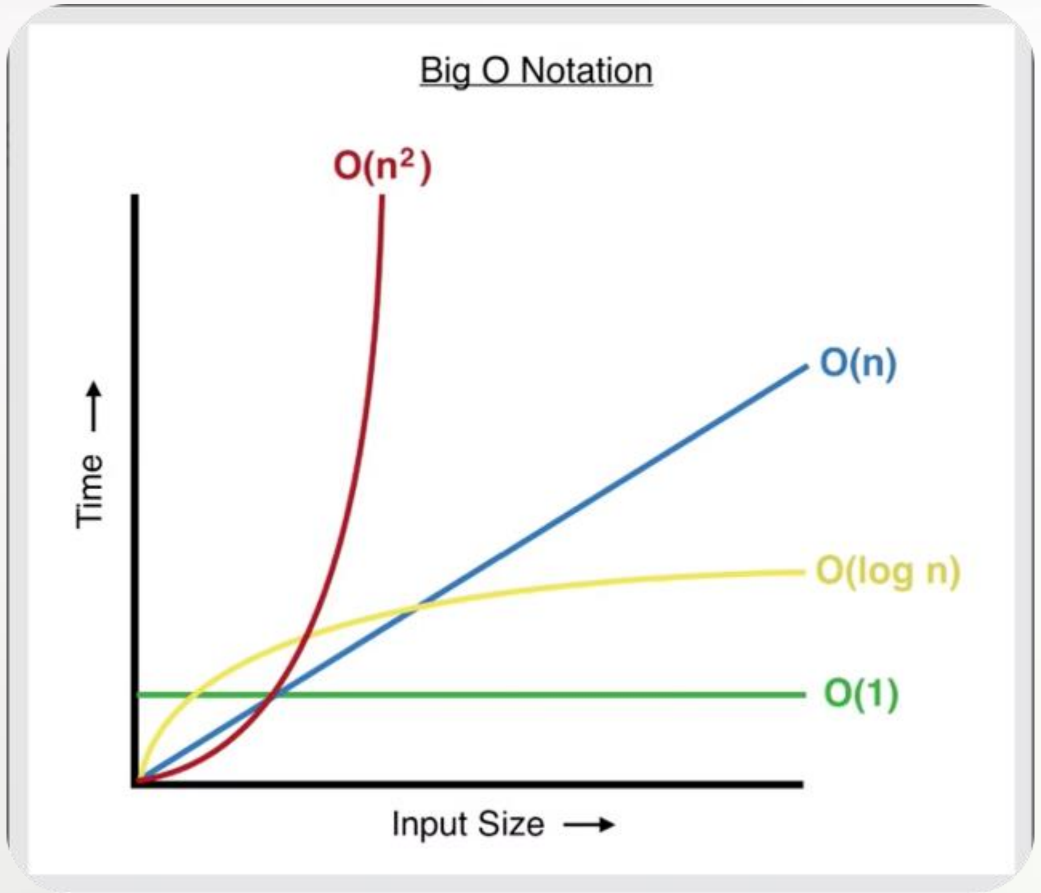
The time complexity of the Reconstruct Itinerary algorithm is O(V + E), where V is the number of cities and E is the number of flights. This is because the algorithm visits each city and edge at most once during the DFS traversal.

Time Complexity

O(n)

Space Complexity

$O(n^2)$



Edge Cases and Handling

The algorithm needs to handle edge cases such as invalid input, disconnected graphs, and cycles in the graph.

Invalid Input

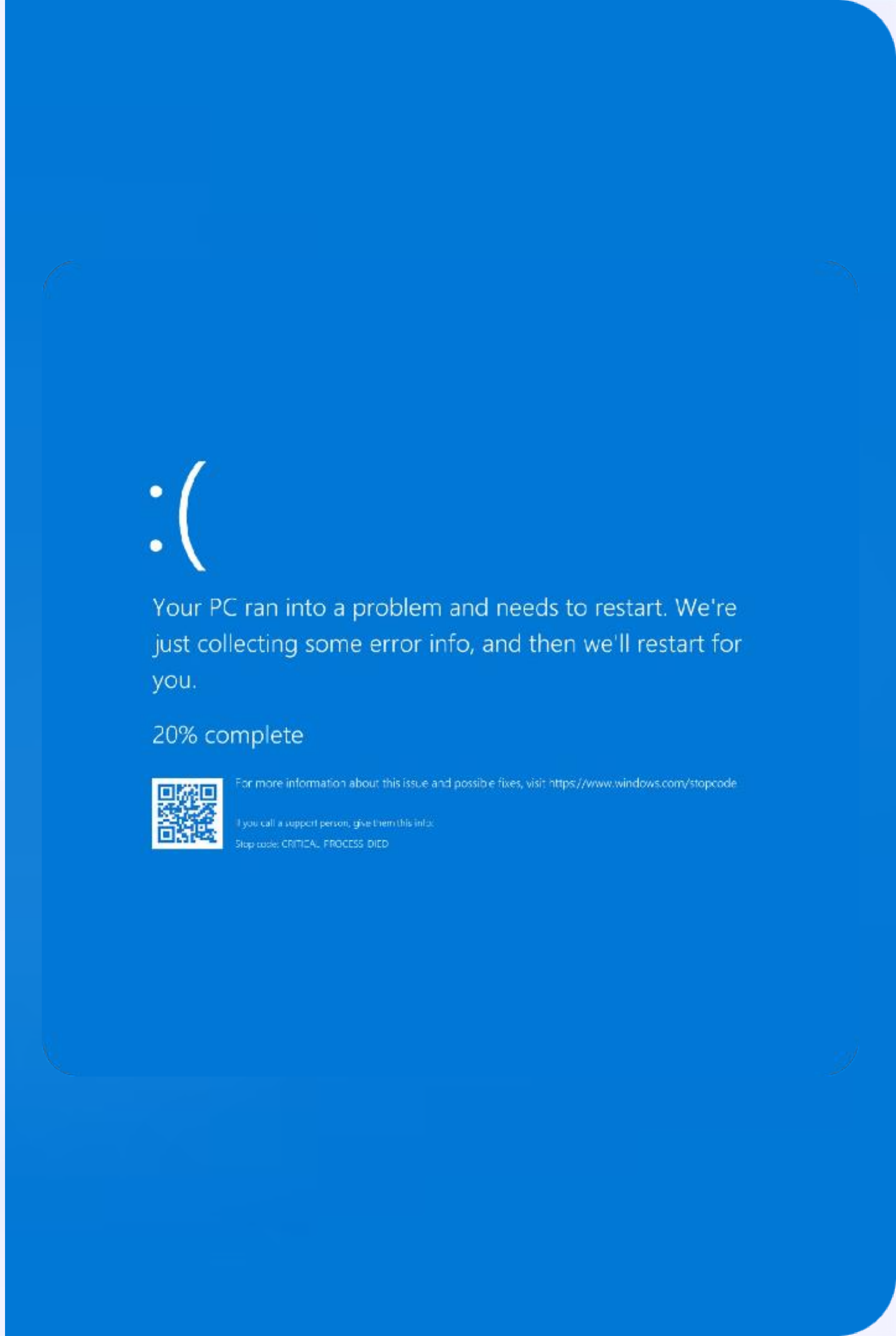
The algorithm should handle cases where the input data is invalid, such as missing flights or duplicate flights.

Disconnected Graph

If the graph is disconnected, the algorithm should not attempt to reconstruct an itinerary that visits all cities. It should return an empty itinerary.

Cycles

The algorithm should handle cases where the graph contains cycles. If a cycle exists, the algorithm may enter an infinite loop. To prevent this, it can check if a city has been visited twice during the traversal. If a city has been visited twice, the algorithm should backtrack to the previous city.



:(

Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you.

20% complete

For more information about this issue and possible fixes, visit https://www.windows.com/stopcode

If you call a support person, give them this info:

Stop code: CRITICAL PROCESS DIED

Practical Applications

This algorithm has practical applications in various fields, including:

1 **Airline Route Optimization**

Airlines can use this algorithm to optimize flight routes and create efficient itineraries for passengers.

2 **Travel Planning**

Travel agencies and websites can leverage this algorithm to generate personalized itineraries for travelers based on their preferences and constraints.

3 **Delivery Routing**

Delivery companies can use this algorithm to optimize delivery routes and ensure efficient delivery of packages.

Conclusion

The Reconstruct Itinerary problem is a classic algorithmic challenge with practical applications in various domains. By understanding the problem, the input data structure, and the DFS approach, we can develop efficient algorithms to solve this problem.