Time and space complexity analysis

Problem solving application.

Airline

Members:

- Alexis Jaramillo (A00395655)
- David Molte (A00368867)
- Juan Daniel Reina (A00394352)

Temporal Complexity

Method: exitOrder

```
public String exitOrder(){
    int order = 1;
    String message ="-----------------------\n"+
                    "Exit order: \n"+
                    "-----------------------\n"+
                    "Name                              ||    Boarding Order    ||    Seat    \n"+
                    "-------------------------------------------------------------------------\n";
    PriorityQueue<Passenger> exitOrder = new PriorityQueue<>();
    exitOrder = exitOrder(exitOrder);
    while(!exitOrder.isEmpty()){
        message += order + ". "+ exitOrder.dequeue().getBoardingInformation()+"\n";
        order++;
    }
    return message;
}
```

The time complexity level of the method is O(n), where n is the total number of passengers on board. This is because the method iterates over three queues (vipBoardingQueue, specialBoardingQueue and boardingQueue) and at each iteration unqueues a passenger from one of the queues, which implies a constant time. Since the length of the queues is equal to the total number of passengers, the time complexity of the method is linear in terms of the number of passengers on board. Therefore, the method is efficient and presents no performance problems in terms of time complexity.

Method: PriorityQueue exitOrder

```
    private PriorityQueue exitOrder(PriorityQueue<Passenger> exitOrder){
        for(int i = 1; i < ids.length; i++){
            if(passengers.get(ids[i-1])!=null && ids[i] != 0){
                exitOrder.enqueue(passengers.get(ids[i]), passengers.get(ids[i]).compareTo(passengers.get(ids[i-1])));
            }
        }
        return exitOrder;
    }
}
```

The time complexity level of this method is O (n log n), where n is the number of passengers. This is because the method uses a for loop to iterate through the passenger identifiers, and within the loop, a comparison of each passenger with its immediate predecessor is performed using the compareTo () method. Then, each passenger is added to a priority queue, which has a logarithmic complexity at each insertion. Therefore, the for loop has a linear complexity of O(n), while the insertion operations on the priority queue have a logarithmic complexity of O (log n), resulting in a total complexity of O (n log n).

Space Complexity

Method: printBoardingOrder:

```
public String printBoardingOrder(){
    int order = 1;
    Passenger passenger;
    String messageBoardingOrder = "-----------------------------\n"+
                                  "Boarding order: \n"+
                                  "-----------------------------\n";
    while (!vipBoardingQueue.isEmpty()){
        passenger = passengers.get(vipBoardingQueue.dequeue());
        messageBoardingOrder += order+". "+ passenger.getName() + "\n";
        order++;
    }
    while (!specialBoardingQueue.isEmpty()){
        passenger = passengers.get(specialBoardingQueue.dequeue());
        messageBoardingOrder += order+". "+passenger.getName() + "\n";
        order++;
    }
    while (!boardingQueue.isEmpty()){
        passenger = passengers.get(boardingQueue.dequeue());
        messageBoardingOrder += order+". "+passenger.getName() + "\n";
        order++;
    }
    return messageBoardingOrder;
}
```

The level of spatial complexity of the printBoardingOrder method is determined by the number of passengers in the vipBoardingQueue, specialBoardingQueue and boardingQueue queues, as well as by the length of the strings being concatenated in the messageBoardingOrder variable.

The space complexity of this method is linear in relation to the number of passengers in the boarding queues. That is, if the number of passengers in the queues is N, then the space used by the method will be of the order of O(N).

Method: printBoardingOrder:

```
public void loadPlaneData() throws IOException {
    File file = new File(planePath);
    if (file.exists()) {
        FileInputStream fis = new FileInputStream(file);
        BufferedReader br = new BufferedReader(new InputStreamReader(fis));
        String line;
        while ((line = br.readLine()) != null) {
            String[] data = line.split(",");
            String flightNumber = data[0];
            String originAirport = data[1];
            String destinationAirport = data[2];
            int rows = Integer.parseInt(data[3]);
            int column = Integer.parseInt(data[4]);
            flight = new Flight(flightNumber, originAirport, destinationAirport, rows, column);
            ids = new int[rows*column*2];
            passengers = new HashTable<>(rows*column*2);
        }
    }
}
```

The level of spatial complexity of this method depends on the number of rows and columns of the aircraft, since a hash table is created to store the passenger data, and an array of integers for the passenger IDs. An instance of the Flight class is also created. In addition, a file is read and processed line by line, so the size of the file may influence memory usage.

In terms of memory, the method can be costly since data structures are created to store the aircraft and passenger information, and data is loaded from a file, which could consume more memory depending on the size of the file. However, memory usage is not directly related to the execution time of the method.