COMP3506/7505 Project – Part A

Due: 13th September 2021 @ 4:00 PM AEST Version: 1.0

Report Template

	Full Name	Student ID
Details	Alexander Frederick Viller	45375325

1. Overview

This document is the ***mandatory*** template which must be used to submit the report section of your project.

This template is automatically synced with Gradescope to identify the location of each section of the report; therefore, it is imperative that the overall format/layout of this document not be modified. Modification of the template **will** result in a penalty being applied.

You are permitted to make changes inside the purple boxes for each question provided however the overall size of the box cannot change. Your report should easily fit within the boxes provided however please be aware the minimum font size allowed is Arial 10pt. If you are exceeding the box size, then this may be a good indication your response is not succinct.

2. Submission

Once you have completed your report this document must be exported as a pdf and uploaded to the Gradescope Report Portal for Part A of this assignment. This document **should not** be uploaded to the autograder.

3. Marking

The report will be hand marked by the teaching team. Information regarding the rubrics used while marking will be made available after grades are released. While this report will indicate the relative weighting of each section of the report, should there be any discrepancy with the official assignment specification, the assignment specification shall take precedent.

4. Plagiarism

The University has strict policies regarding plagiarism. Penalties for engaging in unacceptable behaviour can range from cash fines or loss of grades in a course, through to expulsion from UQ. You are required to read and understand the policies on academic integrity and plagiarism in the course profile (Section 6.1).

If you have any questions regarding acceptable level of collaboration with your peers, please see either the lecturer or your tutor for guidance. Remember that ignorance is not a defence!

5. Task

You are required to complete all sections of this report in line with the programming tasks completed in the project.

Departure Display (20 Marks)

1. Provide a screenshot of the main sorting function for the partially sorted list (*DisplayPartiallySorted*).

```
@Override
Plane[] sort() {
    int length = (this.getSchedule().length + this.getExtraPlanes().length);
    Plane[] newSorted = (Plane[]) Array.newInstance(Plane.class, length);

    int counter = 0;
    for (Plane plane : this.getSchedule()) {
        newSorted[counter] = plane;
        counter++;
    }
    for (Plane plane : this.getExtraPlanes()) {
        newSorted[counter] = plane;
        counter++;
    }

    for (int i = 1; i < length; ++i) {
        for (int j = i; j > 0; --j) {
            if (newSorted[j - 1].compareTo(newSorted[j]) > 0) {
                Plane temp = newSorted[j-1];
                newSorted[j-1] = newSorted[j];
                newSorted[j] = temp;
            }
        }
    }
    return newSorted;
}
public static void quickSort(Plane[] array, int left, int right) {
    if (left < right) {
        int i = left;
        int j = right;
        Plane pivot = array[(int) Math.floor((i + j)/2)];

        do {
            while (array[i].compareTo(pivot) < 0) {
                i++;
            }
            while (pivot.compareTo(array[j]) < 0) {
                j--;
            }


            if (i <= j) {
                Plane temp = array[i];
                array[i] = array[j];
                array[j] = temp;
                i++;
                j--;
            }
```

2. Describe which sorting algorithm you have used for the unsorted list and state its time and space complexity in the best case, average case and worst case, with respect to the number of planes, **n**.

I used quick-sort which does have an absolute worst case running time of $O(2^n)$, however, it has the best case and also average case to sort in O(n log n) time. I used a set pivot (of the middle of the array) rather than a random one, however as the data is random this still leads to what is effectively a random pivot and this helps to avoid the absolute worst run time (n^2). Quick sort works by choosing a pivot, then sorting elements into either greater than the pivot, less than the pivot, or equal to the pivot, then it recursively calls itself on the less than and greater than arrays until everything is in its correct position.

3. Describe the modifications you have made to take advantage of the list being partially sorted. State the complexity of your algorithm with respect to **n** and **k**, where **n** is the number of sorted planes, **k** is the number of unsorted planes.

I used bubble sort, which is classically a slow sort (worst case run time of $O(n^2)$, however, has a best-case running time of O(n). This is only achieved when the input is sorted. We do have that though for our n (the sorted planes). So we end up with a run time complexity of $O(n + k^2)$. The extra planes are likely to be much smaller than the sorted planes as we will have a large database and then want to add a few new ones as they are introduced. This gives us a much faster sort time than the fastest sorts (e.g. Quick sort or merge sort) which typically run in O(n log n). The fastest way to do this, however, would be to have a very fast sorting algorithm like quick sort for the unsorted set, and then merge the two data sets. This would give a running time of O(1 + k log k) = O(k log k) which would be very fast. However, I couldn't work out a good way of implementing this without repeating significant portions of code so went for the far easier to implement bubble sort.

Flight Dispatcher (30 Marks)

1. Provide a screenshot of the functions *allocateLandingSlot/allocate_landing_slot* and *emergencyLanding/emergency_landing*.

```java
@Override
public String allocateLandingSlot(String currentTime) {
    //storing time variables
    String[] time = currentTime.split(":");
    String[] landingTime = data.peek().getTime().split(":");
    Integer[] times = new Integer[4];
    times[0] = Integer.parseInt(time[0]);
    times[1] = Integer.parseInt(time[1]);
    times[2] = Integer.parseInt(landingTime[0]);
    times[3] = Integer.parseInt(landingTime[1]);

    if (time[0].compareTo(landingTime[0]) > 0) { //if the current hour is after the landing time hour
        return data.removeMin().getPlaneNumber();
    } else if (time[0].compareTo(landingTime[0]) == 0) { //same hour for current time and landing time
        if (times[3] - times[1] <= 5 || time[1].compareTo(landingTime[1]) > 0) { //the current minute is after landing time or within 5 mins
            return data.removeMin().getPlaneNumber();
        }
    } else {
        Integer timeDifference = times[3] + (60 - times[1]);
        if (timeDifference <= 5) { //for case where the 5 minute time difference crosses an hour.
            return data.removeMin().getPlaneNumber();
        }
    }
    return null;
}
```

```java
@Override
public String emergencyLanding(String planeNumber) {
    return data.remove(planeNumber);
}       You, 2 hours ago • assignment done
```

```java
public Plane removeMin() {
    if (size == 0) {
        return null;
    }
    Plane temp = queue[size - 1];
    queue[size - 1] = null;
    size--;
    return temp;
}

public Plane peek() {
    if (size == 0) {
        return null;
    }
    return queue[size - 1];
}

public String remove(String planeNumber) {
    if (this.contains(planeNumber)) {
        int planeIndex;
        for (int i = 0; i < size; i++) {
            if (queue[i].getPlaneNumber() == planeNumber) {
                planeIndex = i;
                queue[planeIndex] = null;
                size--;
                sort(queue, 0, size - 1);
                return planeNumber;
            }
        }
    }
    return null;
}
```

```java
public boolean contains(String planeNumber) {
    if (size != 0){
        for (Plane plane : queue) {
            if (plane != null && plane.getPlaneNumber() == planeNumber) {
                return true;
            }
        }
    }
    return false;
}
```

2. Describe the data structure you used to represent a flight dispatcher.

I used a priority queue, sorted using quick sort and my comparator written within the plane class such that the planes are sorted in descending order (that is where the plane with the earliest time is at the item removed in removeMin()). The priority queue is implemented using an array that is sorted after every add and remove and stores its current size as a variable. The actual size of the array is initialised as 11 (copying the java 8 priority queue ADT) and is doubled every time the stored size is about to equal the max size of the array and would cause an index out of bounds error. This means that adding a new element takes worst case O(n + n log n) as the sort is implemented using quick sort. But this occurs once every time the size is doubled, meaning that initial n is only log n in amortised time. However, this is overrun by the n log n term anyway so is not very important.

3. State the best case, average case and worst case time complexities of the functions *allocateLandingSlot/allocate_landing_slot* and *emergencyLanding/emergency_landing* with respect to **n**. Briefly explain how you achieved these complexities given the data structure you have chosen.

allocateLandingSlot runs in constant time (Θ(1)) as it just performs some logic based on the time to check if the last thing in the queue has an appropriate time to be removed, then dequeues it. This is the main reason I chose a priority queue for this section as it provides instant removal of the most appropriate item and thus, the most likely item to be removed.
emergencyLanding runs in linear time (Θ(n)). This was achieved by simply calling the remove method from the queue, which initially checks that the item is in the queue (worst case O(n) time as it just iterates through the array checking for the item, could be optimized by being a binary search, but unnecessary as the other step takes O(n) time and must happen anyway). Then it sets the element at that index to the element after it and continues for the rest of the array and then returns the plane number. This step will always take O(n) has it has to iterate through the whole array in order to shuffle everything into its appropriate index.

Thus, for allocate landing slot I have a worst, best, and average running time of 1, and for emergency landing I have a worst, best and average running time of n.

4. If addPlane/add_plane had to run in constant time, which alternate data structure could be used? How would this affect the running time of *allocateLandingSlot/allocate_landing_slot* and *emergencyLanding/emergency_landing*?

A hash map would probably be the most effective way of storing this. If using a hash map, then emergency landing would run in constant time and allocateLandingSlot would take O(n) (linear search to find the lowest value). You would have amortised time of O(1) for insertion by doubling the array length every time max size was reached, or you could just have the initial size be so large that you never have to worry about it (e.g. since dispatcher is only for one day, set the initial size to 3 * 60 * 24 (gives space for 3 planes every minute for the whole day) = 4320), however this would likely be very space inefficient as the chance of having that many planes constantly is relatively low. The benefit of hashmap would be instant insertion and removals, which would be good for real world emergency situations.

END OF REPORT

ALIGNMENT TEST BOX
DO NOT EDIT