Name: Christopher Chan

Student No: 20143087

Assignment: Algorithms and Advanced Programming Continuous Assessment

Class: HDSDEV_SEP

Part 1: Sorting and Searching: Algorithm Analysis

Because this is an Algorithm analysis project and not an Object Orientated Software Engineering one, I decided to make separate classes for the sorting and searching algorithms for ease of viewing.

Q1. Write a sorting algorithm (algorithm A) that sorts the employee data using the first_name field. You may choose any efficient sorting algorithm we covered in this module.

Merge Sort Algorithm (*MergeSort.java*) was implemented for generic types <T>. My implementation is based off the article(*ref1*). My implementation Accepts a List of objects (type T) that inherits from the Comparable class. In the given Employee class, it implements the comparable interface, the method compareTo() only compares first name of employees. If this function was altered to accept objects of type T we could define on what values to compare by and thus the sorting algorithm could sort in that order.

Q2. Experimentally analyse the time complexity of your sorting algorithm you write for q1 above. Show your results by taking the average elapsed time for 10, 100, 1000, 5000 and 10000 records.

Asymptotic Analysis (Merge Sort):

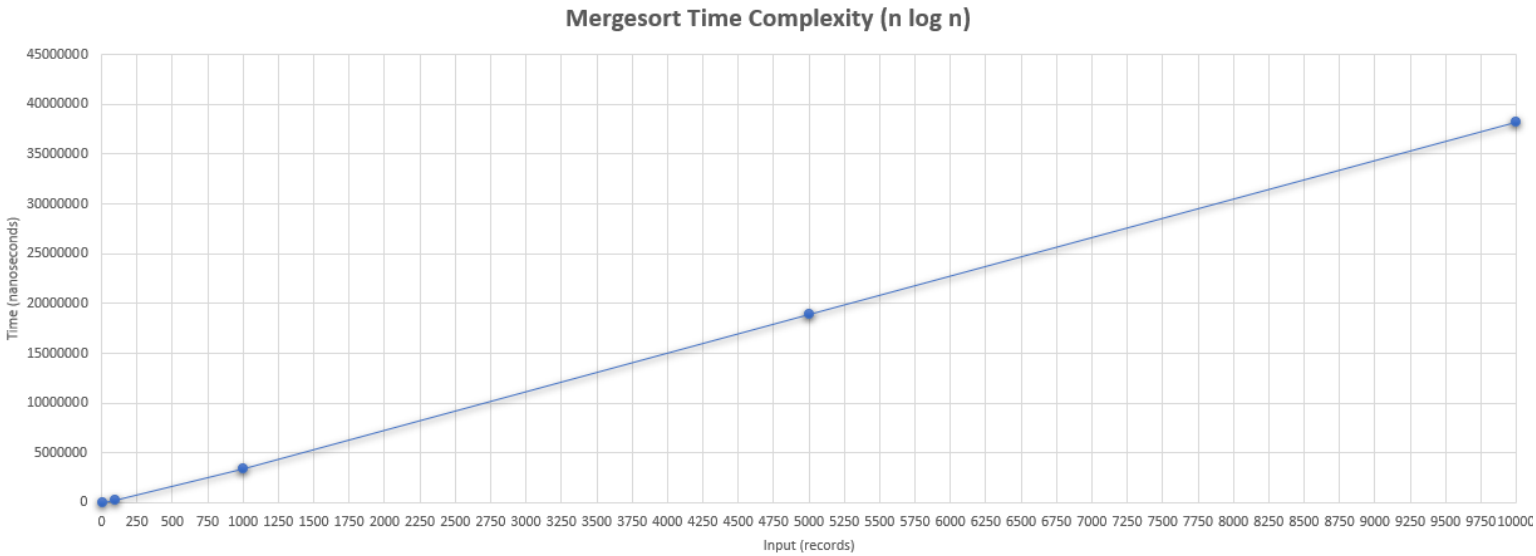
Theoretically Merge Sort has a time complexity of O(n log n) in Big O notation. Let's try prove this with results from our tests. I tested merge sort for 5 times on sample inputs of 10, 100, 1000, 5000 and 10000 records.

```
For input=10 The time taken in nanoseconds is: 18129
For input=100 The time taken in nanoseconds is: 230280
For input=1000 The time taken in nanoseconds is: 3494724
For input=5000 The time taken in nanoseconds is: 21015726
For input=10000 The time taken in nanoseconds is: 41078091
```

Example of one instance of Merge Sort Test

So taking an average time for the 5 tests (*Main.java - line 37*) on each of the 10, 100, 1000, 5000 and 10000 records we can plot of graph and show the distribution of the times we received.

Input Size (records)	Avg time of 5 Tests (nanoseconds)
10	19541
100	208354
1000	3375015
5000	18912991
10000	38269701



mergeSort() Plotted Results

I plotted the x-axis with intervals of 250 records and the y-axis with intervals of 5000000 nano-seconds. The initial 10 and 100 records are difficult to see as the time required is dwarfed by the rapidly increasing time it takes to sort 1000 records and upwards. The distribution plotted is in keeping with O(n log n) distribution on the graph(*ref2*). The line curve is not as steep as a quadratic function but it is steeper than a linear function.

Q3. Write a searching algorithm that accept the first_name of an employee and searches the employee record from the dataset. There could be multiple employees with the same first_name. If no employee is found, display "Not an employee!" message.

To accept a name from the command line I used a scanner object to accept String arguments (*Main.java - line 94*). The name input is case checked and then the searching algorithm is called. A Binary Search algorithm (*BinarySearch.java*) was implemented from the given class notes. It accepts an Arraylist<Employee> only. For our purposes of algorithm analysis this is fine. This function only returns one instance of first name. The upperLowerBound function then searches the indexes of the searched name allowing us a way for printing all the employees with a give first name.

```
/* ********************** Q3 Search and Return Names ****************** */

Scanner scanner = new Scanner(System.in);
System.out.println("Please input a name to search: ");
String name = scanner.next();
// Checks case
name = Character.toUpperCase(name.charAt(0)) + name.substring(1).toLowerCase();

BinarySearch searcher1 = new BinarySearch();
int indexfound = searcher1.binarySearch(employees, name);
int[] upperLowerBound;

// // check if employee exisits
if (indexfound != -1) {
    upperLowerBound = searcher1.upperLowerBound(employees, indexfound, name);
    System.out.println((upperLowerBound[1] - upperLowerBound[0] + 1) + " Employee/Employees found!");
    for (int i = upperLowerBound[0]; i <= upperLowerBound[1]; i++) {
        System.out.println(employees.get(i));
    }
} else {
    System.out.println("Is " + name + " an employee?");
    System.out.println("Not an employee!");
}
```

Code to accept name from command line and print out the results (Main.java - line 94)

```
********** Search an employee from the record **************
Please input a name to search:
wayne
8 Employee/Employees found!
16033 1954-08-06 Wayne Spelt F 1986-07-14
18909 1956-04-08 Wayne Frijda F 1992-09-17
10603 1955-08-02 Wayne Restivo M 1995-07-20
13067 1955-08-01 Wayne Kornyak F 1993-04-08
14818 1952-04-01 Wayne Falco M 1990-07-21
17908 1959-10-08 Wayne Borovoy M 1993-10-14
18376 1952-07-04 Wayne Stranks M 1986-02-10
14576 1962-04-29 Wayne Pelc M 1998-02-14
```

Output when employee is found in the records

```
********** Search an employee from the record **************
Please input a name to search:
bill
Is Bill an employee?
Not an employee!
```

Output when employee is not found in the records

Q4. Analyse and determine the time complexity of the searching algorithm you write in q3.

Asymptotic Analysis (Binary Search):

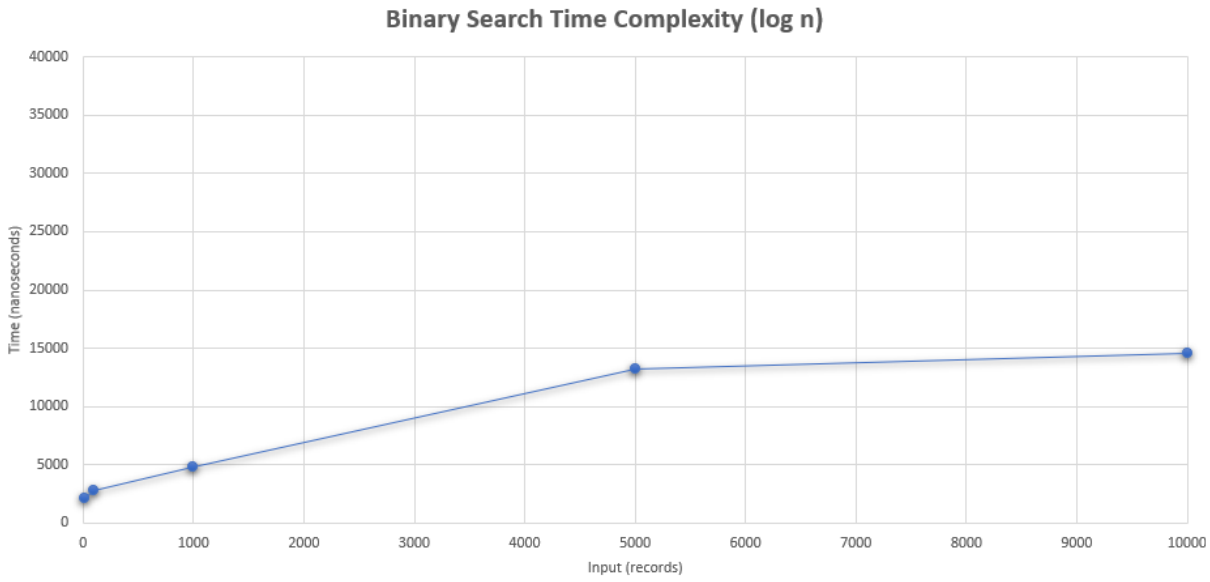
Theoretically Binary Search has a time complexity of O(log n) in Big O notation. Let's plot a graph and see if our implemented algorithm follows a similar distribution. I tested the binary-search() (*BinarySearch.java - line 15*) 5 times to retrieve one instance of a name in the records when the input records are 10, 100, 1000, 5000, 10000.

```
For Input = 10 the time taken to search one instance(index) of the name Duangkaew took 1409 nanoseconds.
For Input = 100 the time taken to search one instance(index) of the name Georgy took 2221 nanoseconds.
For Input = 1000 the time taken to search one instance(index) of the name Theirry took 5593 nanoseconds.
For Input = 5000 the time taken to search one instance(index) of the name Tadahiko took 13709 nanoseconds.
For Input = 10000 the time taken to search one instance(index) of the name Wayne took 7940 nanoseconds.
```

Example of one instance of Binary Search Test

So taking an average time for the 5 tests (*Main.java - line 118*) on each of the 10, 100, 1000, 5000 and 10000 records we can plot a graph and show the distribution of the times we received.

Input Size (records)	Avg time of 5 Tests (nanoseconds)
10	2078
100	2784
1000	4862
5000	13187
10000	14621



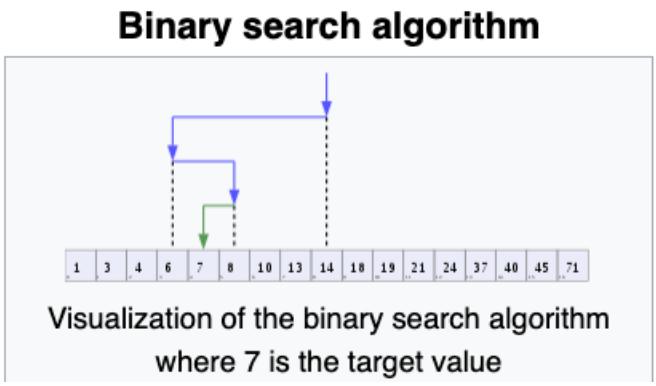
binarySearch() Plotted Results

The distribution plotted is in keeping with O(log n) distribution on the graph(*ref2*).

Mathematical Analysis(Binary Search)

Binary Search works on the idea of: in an ordered set of data, for a given search key finding the midpoint of that set we can eliminate one half of the dataset depending on whether the mid-

point is higher or lower than the search key. If we keep on halving the dataset we will finally arrive at our searched key. This is the concept on which my BinarySearch() (*BinarySearch.java - line 15*) algorithm is based on.



Visualisation of Binary Search(ref3)

For a given set of records, how many times do we have to divide N by 2 until we arrive at 1(our search key)? Writing this into an expression would look like:

$$1 = N / 2^x$$

$$2^x = N$$

$$\log_2(2^x) = \log_2 N$$

$$x * \log_2(2) = \log_2 N$$

$$x * 1 = \log_2 N$$

$$x = \log_2 N$$

So we have to divide $\log_2 N$ times until the algorithm finds the search key.

upperLowerBound

Once the index of one instance of the first name is retrieved using binarySearch() the function upperLowerBound() (*BinarySearch.java - line 38*) iteratively tries to find the upper and lower bound indexes corresponding to the name.This is a linear function searching both indexes iteratively. Because the dataset we are working with is very limited (not more than 15 people with

the same first name) doing time complexity analysis for this function with the given code it is difficult to get sufficiently accurate test results. Theoretically the worst case scenario for this method would be if everyone in our 10000 records had the same first name, then the function would iteratively search through every record before finding the first and last entries.

```java
// linear search upper and lower bound index search algorithim
public int[] upperLowerBound(ArrayList<Employee> employees, int searchIndex, String searchKey)

    // find the lower and upperbound indexes of the searched item
    int upperbound, lowerbound;
    upperbound = lowerbound = searchIndex;
    if (searchIndex != -1) {
        try {
            // iterate to find the upperbound index
            while (employees.get(upperbound + 1).getFirstName().equals(searchKey)) {
                upperbound++;
            }
            // iterate to find the lowerbound index
            while (employees.get(lowerbound - 1).getFirstName().equals(searchKey)) {
                lowerbound--;
            }
        } catch (ArrayIndexOutOfBoundsException e) {
        }
    }
    // copy and return arrays of the upper and lower bound indexes
    int[] upperLowerBound = new int[2];
    upperLowerBound[0] = lowerbound;
    upperLowerBound[1] = upperbound;
    return upperLowerBound;
}
```

UpperLowerBound() (BinarySearch.java - line 38)

If we look at the code we can see 2 while loops, these are the dominant operations in the algorithm. If the Big O time complexity of a standard loop is O(n), then some expression: 2n + other constants can be deduced from this method. The other constants such as assigning variables can be ignored as we are only interested in dominant operations. O(2n) = O(n), neither is “faster” than the other in regards to asymptotic complexity(*ref4*). The growth rates here are linear thus we can deduce the time complexity is O(n).

Part 2: Defensive Programming and Exception Handling

Q1. Write a Java program that accept new employee record (with all the six fields) and add it at the end of the record with a new consecutive emp_no (after the last employee).

The newEmployee() (*Main.java - line 252*) is a static method when called allows user input to add a new employee from the command line with all the fields required. Date of Birth and Naming, business rules from Q2 and Q3 are also taken into account when adding an employee. This method is called at(*Main.java - line 208*).

```
// Q2 Add an employee record to the end of the array

// Calls a static method adding employee from user input - line 261
Employee newEmployee = newEmployee();
List<Employee> newlist = new ArrayList<>();

// lets parse the original list for 10000 employee records - line 245
newlist = parseArrayList(list, 10000);

// add employee too the oringial list - line 331
addEmployee(newlist, newEmployee);

// print all the employees
for (Employee employee : newlist) {
    System.out.println(employee);
}
```

Main.java - line 198

Once we have correctly made a new employee the addEmployee() (*Main.java - line 329*) adds it to the end of the records/List with consecutive employee number, the program then prints the entire list of employee records.

```
// add employee with consecutive employee number
public static void addEmployee(List<Employee> employees, Employee employee) {
    Employee temp = employees.get(employees.size() - 1);
    employee.setEmpNo(temp.getEmpNo() + 1);
    employees.add(employee);
}
```

addEmployee() Main.java - line 329

```
19999 1953-10-16 Jahangir Speer F 1989-09-29
20000 1961-09-14 Jenwei Matzke F 1990-11-29
20001 1950-12-12 Christopher Chan M 2020-12-12
Christobels-MacBook-Pro:Continuous Assesment christobel$
```

Sample output when the records are printed

Q2. Write a Java method that checks the correct values of the birth_date . The birth_date should not be less than 01-01-1950. Also, the company should not higher employees younger than 18 years (current year –18).

Validating Date of Birth - Employee Class

Validating the birthday of an employee is contained within the setter method of the Employee class setDateOfBirth() (*Employee.java - line 43*). This method uses a SimpleDateFormat Obj, to define the date format and a Date Obj to compare the entered birthdays once the dates are parsed. If the date entered does not satisfy our date of birth business rule an IllegalArgument-Exception will be thrown.

```
// setter with specified business rules
public void setDateOfBirth(String dateOfBirth) throws ParseException {
    // date format obj
    SimpleDateFormat sdfObj = new SimpleDateFormat("yyyy-MM-dd");

    // declaring the date objects
    Date lowerDate = sdfObj.parse("1950-01-01");
    Date upperDate = sdfObj.parse("2003-01-01");
    Date checking = sdfObj.parse(dateOfBirth);

    if (checking.compareTo(lowerDate) < 0 || checking.compareTo(upperDate) > 0) {
        throw new IllegalArgumentException("Employees between 1950-01-01 and 2003-01-01 are ONLY allowed.")
    } else {
        this.dateOfBirth = dateOfBirth;
    }
}
```

setDateOfBirth() (*Employee.java - line 43*)

The date of birth business rule is also present when trying to create an employee through the overloaded constructor.

```
// overloaded constructor
public Employee(int empNo, String dateOfBirth, String firstName,
String lastName, char gender, String hireDate)
        throws ParseException {
    this.empNo = empNo;
    this.setDateOfBirth(dateOfBirth);
```

Business rule applies in constructor *(Employee.java - line24)*

Validating Date of Birth - Command line User Entry

This validateDateOfBirth()(*Main.java - line 351*) method is called when we are adding a new Employee newEmployee() *(Main.java - line 261*) from the command line. The validateDateOfBirth() is very similar to setDateOfBirth() in the Employee class using a SimpleDateFormat Obj for validation. This time when reading in the user input the code checks if the expression entered if of the correct format.

```
// input format for RegEx -> dddd-dd-dd (d = digit)
if (stringInput.matches("(\\d{4})-(\\d{2})-(\\d{2})")) {
```

Validating the date input format - Main.java line 303

Once the date inputted satisfies the format constraints the validateDateOfBirth() is called. Below is some sample user input and how the program responds.

```
Employees between 1950-01-01 and 2003-01-01 are ONLY allowed.
194212-21-12
Enter Date of Birth (YYYY-MM-DD):
Employees between 1950-01-01 and 2003-01-01 are ONLY allowed.
1949-01-13
Enter Date of Birth (YYYY-MM-DD):
Employees between 1950-01-01 and 2003-01-01 are ONLY allowed.
1950-01-01
Enter Date of Hire (YYYY-MM-DD):
```

Sample user input - how the program responds

Q3. Write a Java Exception that handles special cases and communicate to users to correct the cases. A typical special case is that the first_name field cannot be empty or cannot contain digits only. The exception should generate "Employee first_name cannot be empty. It cannot have only digits! Please correct this" message.

Validating First Name - Employee.java

Validation of first name for our business rule is applied in the code snippet below (*Employee.java - line 65*). If the rule is not satisfied it throws an IllegalArgumentException informing the user why that first name cannot be entered.

```java
// setter with specified business rules
public void setFirstName(String firstName) {
    if (firstName.equals("") || firstName.matches("[0-9]+")) {
        throw new IllegalArgumentException(
                "Employee first_name cannot be empty. It cannot have only digits! Please correct this");
    }
    this.firstName = firstName;
}
```

setFirstName() (*Employee.java - line 65*)

The same rule also needs to be satisfied when adding an Employee through the overloaded constructor.

```
this.setFirstName(firstName);
```

Setting first name in constructor Employee.java line - 25

Validating First Name - Command Line User Input

ValidateFirstName() (*Main.java - line 366*) is a very similar method to setFirstName() as they achieve validation throwing the same exception. The inputted name cannot be an empty string nor can it be only digits.

```
//defines the business rule name cant be empty string or contain only digits
public static boolean validateFirstName(String firstName) {
    if (firstName.equals("") || firstName.matches("[0-9]+")) {
        return false;
    }
    return true;
}
```

validateFirstName() (*Main.java - line 367*)

When reading in a new employee's details from the console by calling newEmployee() (*Main.java - line 273*), the method handles input errors of this specific business rule by throwing an IllegalArgumentException() (*Main.java - line 279*) and prints out to the user what is a valid input.

```
while (!flag) {
    System.out.println("Enter First Name: ");
    stringInput = scanner.nextLine();
    try {
        if (!validateFirstName(stringInput)) {
            throw new IllegalArgumentException();
        } else {
            addNewEmployee.setFirstName(stringInput);
            flag = true;
        }
    } catch (IllegalArgumentException e) {
        System.out.println(
                "Employee first_name cannot be empty. It
                cannot have only digits! Please correct this");
    }
}
```

Code snippet from newEmployee() (Main.java - line 269)

The program responds not by exiting but by telling the user how to rectify the mistaken input.

```
Enter First Name:

Employee first_name cannot be empty. It cannot have only digits! Please correct this
Enter First Name:
12345678
Employee first_name cannot be empty. It cannot have only digits! Please correct this
Enter First Name:
Christopher
Enter Surname Name:
```

How the program reacts to certain inputs

References:

1. Merge Sort (Generic): https://stackoverflow.com/questions/34783815/java-recursive-mergesort-for-arraylists/34825244
2. BigOCheatsheet: https://www.bigocheatsheet.com/
3. Binary Search: https://en.wikipedia.org/wiki/Binary_search_algorithm
4. O(n) = O(n2): https://stackoverflow.com/questions/25777714/which-algorithm-is-faster-on-or-o2n