Part 1: Sorting and Searching: Algorithm Analysis (70 marks)

Task 1

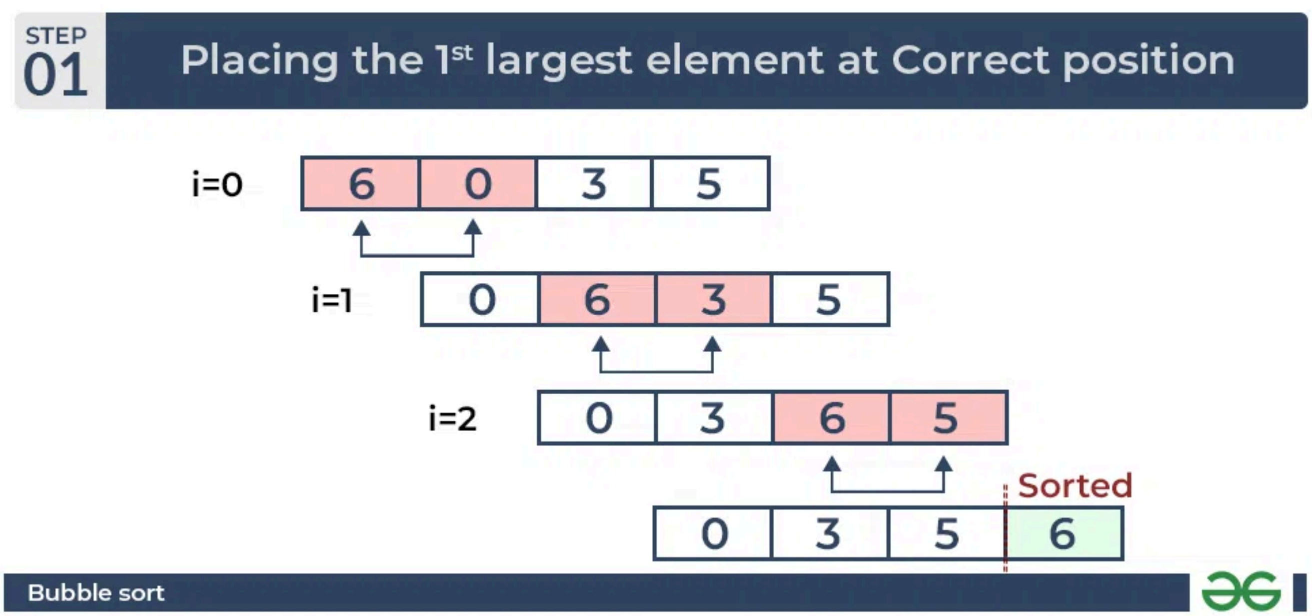
1. Write a Bubble Sort algorithm that sorts the data using a column based on your student number. If two items have the same value sort based on column 1.
   You will receive higher marks for optimal (low run-time) solutions. **Highlight in the submission the reason why you chose your sorting algorithm with reference to the run-time complexity.** The sorting algorithm must be your own implementation. You will receive 0 marks for using an imported library to complete this task.

   Reason why I chose this sorting algorithm with reference to the run-time complexity

   First of all I would like to say that a Bubble Sort is a sorting algorithm. In algorithm terminology,sorting means to arrange a list of elements into a particular order according to a well-defined ordering rule (ascending or descending).

   A bubble sort is an algorithm based on comparing adjacent elements, and exchanging or swapping adjacent elements if they are out of order.

   The values are sorted into passes. For clarity purposes: This is one pass. Source https://www.geeksforgeeks.org/bubble-sort-algorithm/



   After the first pass through the collection, the maximum value will 'bubble' to the end/top of the collection. The passes are repeated until the collection is sorted.

   Visuals and gifs are a very efficient way to understand graphically how an algorithm works. In this sense, this W3C resource is great to see a bubble algorithm work in an animated way.

Bubble sort is one of the oldest algorithms and not the most efficient algorithm, so in terms of Big O represents an O(N2) complexity, meaning that the time needed is quadratic to the number of inputs. Not an ideal scenario.

When filtering a dataset with 10,000 records on my AsusVivo Pro, I found bubble sort to be particularly slow. Despite my computer not being the fastest, it served as a good test for understanding the performance of O(N2) algorithms when running locally and a good exercise also to imagine how it can perform on the user experience on live products.

In spite of being one of the less efficient algorithms, in class we mentioned a way to optimize the algorithm. For a quick recap see the code snippets formatted for legibility.

Simple implementation

```
public void simpleBubbleSort() {
    // write the simpleBubbleSort() method
      //it simply compares neighbours repeatedly until there are no more
swaps
    boolean bMoreSwaps = true;

    while (bMoreSwaps == true) {
        int iCount;
        bMoreSwaps = false;
        for (iCount = 0; iCount < size() - 1; iCount++) {
            Comparable elementAtiCount = (Comparable) get(iCount);
            Comparable elementAtiCountPlus = (Comparable) get(iCount + 1);

            if (elementAtiCount.compareTo(elementAtiCountPlus) > 0) {
                swap(iCount, iCount + 1);
                bMoreSwaps = true;
            }
        }
    }
}
}
```

Versus optimized

```
public void bubbleSort() {
    int iCount, jCount;
    Comparable elementAtjCount, elementAtjCountPlus;
    for (iCount = 0; iCount < size(); iCount++) {
        for (jCount = 0; jCount < size() - 1 - iCount; jCount++) {
            elementAtjCount = (Comparable) get(jCount);
            elementAtjCountPlus = (Comparable) get(jCount + 1);
```

```
            if (elementAtjCount.compareTo(elementAtjCountPlus) > 0) {
                  //swap element on postion jCount with element on position
jCount + 1
                swap(jCount, jCount + 1);
            }
        }
    }
}

public void swap(int inPos1, int inPos2) {
    //Create two objects that will store the info from the two positions
    ElementType objPos1 = get(inPos1);
    ElementType objPos2 = get(inPos2);

    //Remove element from position 1
    remove(inPos1);

    //Insert objPos2 into position 1
    add(inPos1, objPos2);

    //Remove element from position 2
    remove(inPos2);

    // Insert objPos1 into position 2
    add(inPos2, objPos1);
}
```

The optimized bubble sort is more efficient because it reduces the number of comparisons by decreasing the range of the inner loop with each pass. And as checked in class, efficiency of a sorting algorithm depends on the number of comparisons and the number of swaps.

```
for (jCount = 0; jCount < size() - 1 - iCount; jCount++)
```

After each full pass (e.g., first pass, second pass), the algorithm excludes the last sorted element because it is already in its correct position. This avoids unnecessary comparisons and swaps, increasing efficiency

On the other hand the simple buble sort checks the entire list for swaps in every single interaction until no more swaps are needed. bMoreSwaps is used to detect if more swaps were made. If no more swaps are made, the list is sorted. If not, it continues.

In conclusion, while bubble sort isn't the most efficient sorting algorithm, choosing the optimized version over the simple one is a logical decision for having better performance. The optimized version performs fewer comparisons, making it faster and more suitable for larger datasets.

Code implemented -

I added comments on the adjustments done over the readPeopleData and People classes. I did not use insatiable classes, but generics.

Important to note that I had issues with the relative path to read the excel file. So I used the absolute path. Please change that part of the code to make it work, it should work as I could test 100 times before the submission date.

Another consideration is that for this deliverability I used NetBeans. However, to make it more readable in this doc, I pasted here the formatted code in VSC (Comments and code can be read easier).

```
package ca1;

import java.io.File;
import java.util.*;

/**
 * @author Ruben
 */
public class ReadPeopleData {
    public static void main(String[] args) throws Exception {
        // Parsing and reading the CSV file data into the object array
          // I am not using the relative path provided because I had some
issues running
        // the code with it
        // String name = directory.getAbsolutePath() + "//people.csv";
        File directory = new File(

"C:/Users/Ruben/OneDrive/escritorio/programacion/nci/algorithms/CA1/CA1-alg
orithms/src/ca1");
        String name = directory.getAbsolutePath() + "/people.csv";

        // Array to store People objects - changed it a bit. In here people
variable
        // refers to an a new array of people containing
        // objects with a length of 10,000
        People[] people = new People[10];

        try (Scanner scanner = new Scanner(new File(name))) {
            // this will just print the header in CSV file
            scanner.nextLine();
            int i = 0;
            String sGetData;
```

```
            while (scanner.hasNextLine() && i < 10) { // && i < n - limit
loop to read only x records, change also line
                                                        // 19
                sGetData = scanner.nextLine();
                String[] data = sGetData.split(",");
                       people[i++] = new People(Integer.parseInt(data[0]),
data[1], data[2], data[3],
                                                 Integer.parseInt(data[4]),
Long.parseLong(data[5]));
            }

            // closes the scanner
        }

           // Convert the people array (containing People Objects) into a
MyArrayList
                          MyArrayList<People>   sortedPeopleList   =   new
MyArrayList<>(Arrays.asList(people));

        // Record start time in milliseconds
        long startTime = System.currentTimeMillis();

        // Call the bubbleSort() method
        sortedPeopleList.bubbleSort();

        // Record end time
        long endTime = System.currentTimeMillis();

        // Calculate the duration
        long duration = endTime - startTime;

        // Print the time taken
            System.out.println("Time taken for sorting: " + duration + "
milliseconds");

        // Print all entries in the sorted people list
        for (People person : sortedPeopleList) {
            System.out.println(person);
        }
    }
}

// People class
class People implements Comparable<People> {
    private int iId;
    private String sName;
```

```java
    private String sSurname;
    private String sJob;
    private int iAge;
    private long lCredit;

    // People Constructor
      public  People(int  iInId,  String  sInName,  String  sInSurname,  String
sInJob, int iInAge, long lInCredit) {
        this.iId = iInId;
        this.sName = sInName;
        this.sSurname = sInSurname;
        this.sJob = sInJob;
        this.iAge = iInAge;
        this.lCredit = lInCredit;
    }

    @Override
    public int compareTo(People myPeople) {
        // Compare by lCredit as per requirement
        // This section compares the column of the excel
        // my column is iCredit (6th column)
        return Long.compare(this.lCredit, myPeople.lCredit);
    }

    @Override
    public String toString() {
        return "Person [ID= " + iId + ", Name= " + sName + ", Surname= "
                + sSurname + ", Job= " + sJob + ", Age= "
                + iAge + ", Credit= " + lCredit + "]";
    }

    // Getters and setters
    public int getiId() {
        return iId;
    }

    public void setiId(int iId) {
        this.iId = iId;
    }

    public String getsName() {
        return sName;
    }

    public void setsName(String sName) {
        this.sName = sName;
```

```java
    }

    public String getsSurname() {
        return sSurname;
    }

    public void setsSurname(String sSurname) {
        this.sSurname = sSurname;
    }

    public String getsJob() {
        return sJob;
    }

    public void setsJob(String sJob) {
        this.sJob = sJob;
    }

    public int getiAge() {
        return iAge;
    }

    public void setiAge(int iAge) {
        this.iAge = iAge;
    }

    public long getlCredit() {
        return lCredit;
    }

    public void setlCredit(long lCredit) {
        this.lCredit = lCredit;
    }
}

// MyArrayList --> this implements the bubbleAlgorithm - followed example
in
// class
// Array class extends arrayList and adds a bubbleSort method
class MyArrayList<ElementType extends Comparable<ElementType>> extends
ArrayList<ElementType> {

    // default constructor
    public MyArrayList() {
        super();
    }
```

```
    // Constructor that accepts a collection
    public MyArrayList(Collection<? extends ElementType> c) {
        super(c);
    }

    // Method to sort the list using the bubble sort algorithm
    public void bubbleSort() {
        int iCount, jCount;
        // Outer loop to traverse through all elements
        for (iCount = 0; iCount < size(); iCount++) { // O(N) * 1 = O(N)
            // Inner loop for comparing adjacent elements
             for (jCount = 0; jCount < size() - 1 - iCount; jCount++) { //
O(M) * O(1) = O(M)
                // Get the current element and the next element
                ElementType elementAtjCount = get(jCount); // O(1) + O(1) =
O(2) = O(1) simplified
                 ElementType elementAtjCountPlus = get(jCount + 1); // O(1)
+ O(1) = O(2) = O(1) simplified

                // Compare the current element with the next element
                // If the current element is greater, swap them
                  if (elementAtjCount.compareTo(elementAtjCountPlus) > 0) {
// O(1)
                        // Swap element at position jCount with element at
position jCount + 1
                    swap(jCount, jCount + 1); // O(1)
                }
            }
        }
    }

    // Method to swap elements at two positions
    public void swap(int inPos1, int inPos2) {
            // Create two objects that will store the info from the two
positions
          ElementType objPos1 = get(inPos1); // O(1) + O(1) = O(2) = O(1)
simplified
          ElementType objPos2 = get(inPos2); // O(1) + O(1) = O(2) = O(1)
simplified

        // Remove element from position 1
        remove(inPos1); // O(N)

        // Insert objPos2 into position 1
        add(inPos1, objPos2); // O(N)
```

```
        // Remove element from position 2
        remove(inPos2); // O(N)

        // Insert objPos1 into position 2
        add(inPos2, objPos1); // O(N)
    }
}
```

Task 2

2. Experimentally analyse the time complexity of your sorting algorithm that you wrote for question 1 above. **Show your results by taking the average elapsed time for 10, 100, 1000, 5000 and 10000 records.**

When running the algorithm in NeatBeans, I received an output in seconds, which is not ideal for visualizing the data in an excel graph. Milliseconds is a more accurate approach.

```
BUILD SUCCESSFUL (total time: 1 minute 28 seconds)
```

To change that I implemented the following code snippet (highlighted the important parts)

```
public class ReadPeopleData {
    public static void main(String[] args) throws Exception {
        // Parsing and reading the CSV file data into the object array
         // I am not using the relative path provided because I had some issues running the code with it
        // String name = directory.getAbsolutePath() + "//people.csv";
        File directory = new File("C:/Users/Ruben/OneDrive/escritorio/programacion/nci/algorithms/CA1/CA1-algorithms/src/ca1");
        String name = directory.getAbsolutePath() + "/people.csv";

         // Array to store People objects - changed it a bit. In here people variable refers to an a new array of people containing
        // objects with a lenght of 10.000
        People[] people = new People[10];

        try (Scanner scanner = new Scanner(new File(name))) {
           // this will just print the header in CSV file
           scanner.nextLine();
           int i = 0;
           String sGetData;
```

```
            while (scanner.hasNextLine() && i < 10) { // && i < n - limit loop to read only x records, change also line 19

            sGetData = scanner.nextLine();
            String[] data = sGetData.split(",");
                people[i++] = new People(Integer.parseInt(data[0]), data[1], data[2], data[3], Integer.parseInt(data[4]), Long.parseLong(data[5]));
        }

        // closes the scanner

    }

// Convert the people array (containing People Objects) into a MyArrayList
        MyArrayList<People> sortedPeopleList = new MyArrayList<>(Arrays.asList(people));

    // Record start time in milliseconds
    long startTime = System.currentTimeMillis();

    // Call the bubbleSort() method
    sortedPeopleList.bubbleSort();

    // Record end time
    long endTime = System.currentTimeMillis();

    // Calculate the duration
    long duration = endTime - startTime;

    // Print the time taken
    System.out.println("Time taken for sorting: " + duration + " milliseconds");

    // Print all entries in the sorted people list
    for (People person : sortedPeopleList) {
        System.out.println(person);
    }
  }
}
```

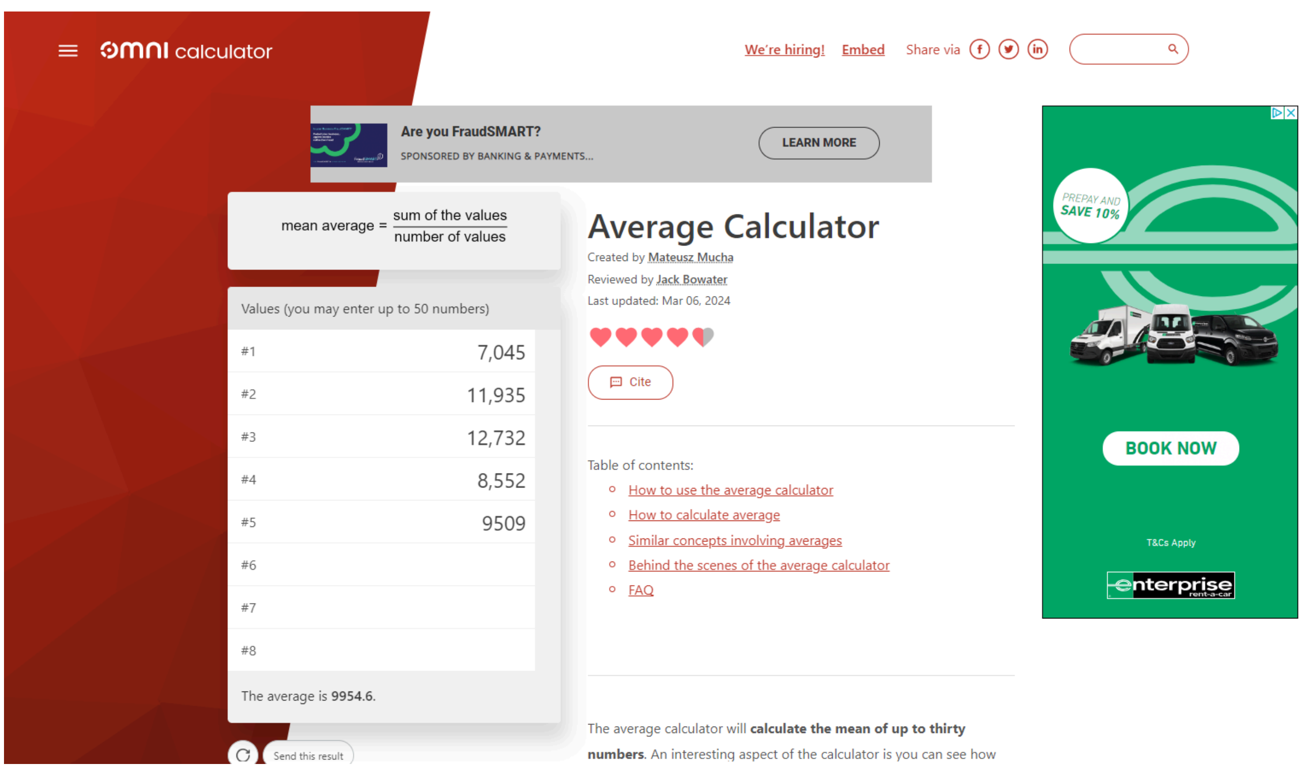
With the changes I applied I can visualize the time the algorithm takes to run with different inputs, and therefore calculate the algorithm complexity (see screenshot below)

```java
        }

// Convert the people array (containing People Objects) into a MyArrayList
        MyArrayList<People> sortedPeopleList = new MyArrayList<>(Arrays.asList(people));

        // Record start time in milliseconds
        long startTime = System.currentTimeMillis();

        // Call the bubbleSort() method
        sortedPeopleList.bubbleSort();

        // Record end time
```

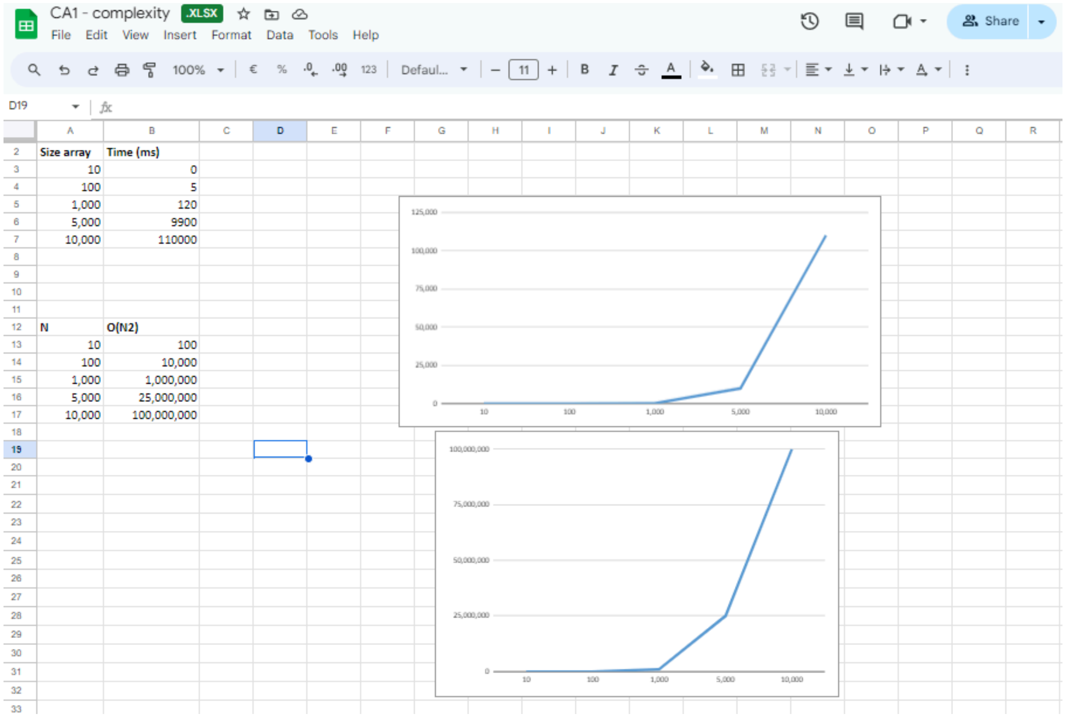
```
Time taken for sorting: 0 milliseconds
Person [ID= 2, Name= Guadalupe, Surname= Runolfsdottir, Job= Chief Identity Agent, Age= 84, Credit= 14484382]
Person [ID= 6, Name= Felicia, Surname= Robel, Job= Direct Metrics Planner, Age= 9, Credit= 83456613]
Person [ID= 3, Name= Justin, Surname= Cronin, Job= Product Creative Executive, Age= 33, Credit= 101804299]
Person [ID= 5, Name= Johnnie, Surname= Hauck-Maggio, Job= Investor Integration Specialist, Age= 91, Credit= 126052246]
Person [ID= 9, Name= Lee, Surname= Krajcik, Job= Senior Identity Producer, Age= 35, Credit= 508599181]
Person [ID= 10, Name= Kristie, Surname= Borer, Job= Central Response Manager, Age= 89, Credit= 598325640]
Person [ID= 8, Name= Josh, Surname= Gorczany, Job= Product Brand Coordinator, Age= 15, Credit= 639610120]
Person [ID= 1, Name= Inez, Surname= Kirlin, Job= Internal Marketing Producer, Age= 69, Credit= 822749358]
Person [ID= 7, Name= Rafael, Surname= Heaney, Job= Customer Division Assistant, Age= 83, Credit= 844468895]
Person [ID= 4, Name= Shari, Surname= Treutel, Job= Forward Applications Representative, Age= 107, Credit= 983429779]
```

Next step is to run the code with the different input sizes. As the exercise demands 10, 100, 1000, 5000 and 10000. I display in an excel template as the one shown below the average of the time taken for the algorithm when using a different number of inputs.

I used the time displayed by Netbeans and calculated the average using this tool https://www.omnicalculator.com/math/average, here an image of the average time complexity for 5 records of the bubbleSort algorithm when it takes 5000 inputs



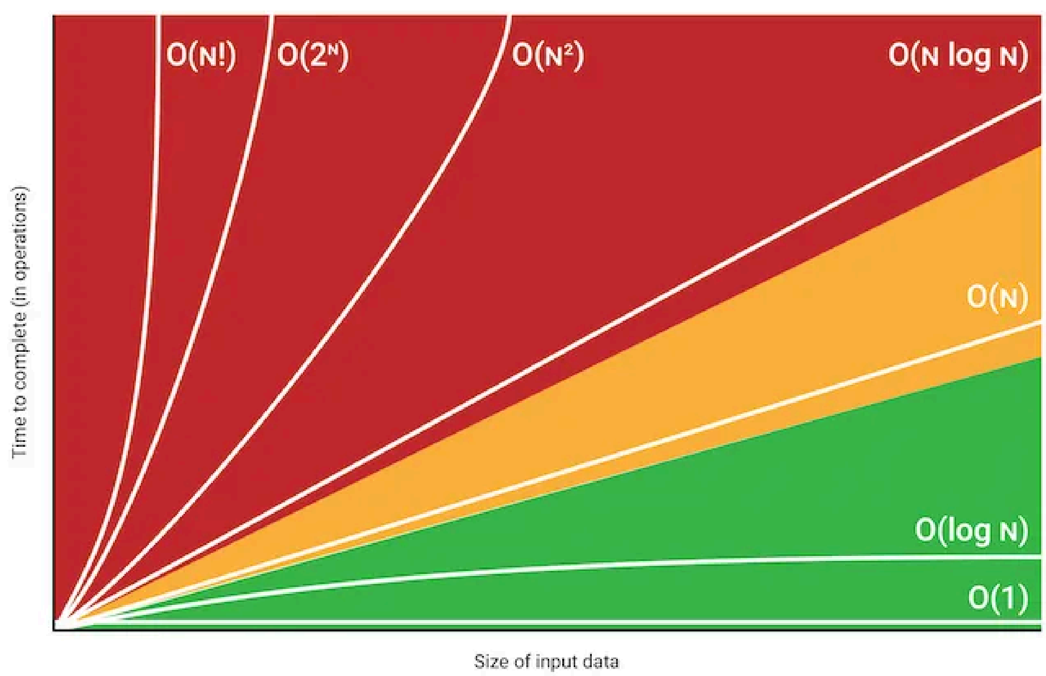
Based on the data, I plot the excel file (attached in the deliverables)



The graphic can be seen as O(N2), because it increases exponentially, and represents an in danger time performance as it can be seen in the following graphic.

I work in a Sass company called Qualtrics, and I see datasets with thousands of data. Imagine waiting 20 seconds or more just to see the data sorted: UX would be poor, increasing tickets being opened with support analysts and increasing product churn ratio. Here is where I realized why BigO and experimental analysis are important in a business context.

Here is a graph from this [source](), that visually explains how the algorithms affects depending on the size of input data. O(n2) is in a danger zone.



The graphic plot in excel performs as O(N2) and by theory we know that a BubbleSort algorithm has O(N2) complexity, but it is also necessary to add the code lines that explains why

```
    //Method to sort the list using the bubble sort algorithm
    public void bubbleSort() {
        int iCount, jCount;
        // Outer loop to traverse through all elements
        for (iCount = 0; iCount < size(); iCount++) { //O(N) * 1 = O(N)
            //Inner loop for comapring adjacent elements
             for (jCount = 0; jCount < size() - 1 - iCount; jCount++) { //O(M) *
O(1) = O(M)

                //Get the current element and the next element
                ElementType elementAtjCount = get(jCount); // O(1)
                ElementType elementAtjCountPlus = get(jCount + 1); //O(1)

                //Comapre the current element with the next element
                // If the current element is greater, swap them
                if (elementAtjCount.compareTo(elementAtjCountPlus) > 0) { //O(1)
                      // Swap element at position jCount with element at position
jCount + 1
                    swap(jCount, jCount + 1); //O(1)
                }
            }
        }
    }

    //Method to swap elements at two positions
    public void swap(int inPos1, int inPos2) {
        // Create two objects that will store the info from the two positions
        ElementType objPos1 = get(inPos1); //O(1)
        ElementType objPos2 = get(inPos2); //O(1)

        // Remove element from position 1
        remove(inPos1); //O(N)

        // Insert objPos2 into position 1
        add(inPos1, objPos2); //O(N)

        // Remove element from position 2
        remove(inPos2); //O(N)

        // Insert objPos1 into position 2
        add(inPos2, objPos1); //O(N)
```

```
    }
}
```

And here a more detailed explanation for each of the elements present in the algorithm:

Initial loop

- for(iCount = 0; iCount <size(); iCount++) // O(N) ⇒ Outer Loop runs from 0 to size(). It iterates N times where N is the size of the list
- for (jCount = 0; jCount < size() - 1 - iCount; jCount // O(M) → Inner loop: For each iteration of the outer loop, it runs from 0 to size() - 1 - iCount. The numeb rof iterations decreases with each pass of the outer loop, making the complexity O(N - iCount). Although it could be approx as O(N/2) for the inner loop. For simplicity sake is O(N)
- ElementType objPost1 = get(inPos1) // O(1)
- ElementType objPos1 = get(inPos2) // O(1)

Overall complexity of the bubbleSort: as it can be appreciated, the structure of the bubbleSort contains a nested loop structure. The outer loop runs 'N' times and the inner loop runs approx N times on each iteration of the OuterLoop.

Therefore, complexity if O(N) x O(N) = O(N2)

Swap method is O(N)

- ElementType objPos1 = get(inPos1); // O(1) ElementType objPos2 = get(inPos2); // O(1). Get retrieves the element at the specified position and this is O(1) because it involves direct-index based access.
- remove(inPos1); // O(N): Remove element is O(N) because involves shifting all subsequent elements to fill the gap. This operation is O(N), where 'N' is the number of elements in the list
- add(inPos1, objPos2); // O(N). Inserting an element at a specific position in an 'ArrayList' requires shifting elements to accommodate the new element, making this operation O(n)
- remove(inPos2); // O(N) add(inPos2, objPos1); // O(N) - are similar to the previous remove and add operations and also have O(N) complexity

Total complexity of the swap method is

Retrieve Elements - O(1) + O(1) = O(2) = O(1)
Remove elements O(N) + O(N) = O(2N), simplified O(N)
Add elements = O(N) + O(N) = O(2N), simplified O(N)

Overall complexity of the bubbleSort:

As it can be appreciated, the structure of the bubbleSort contains a nested loop structure. The outer loop runs 'N' times and the inner loop runs approx N times on each iteration of the OuterLoop. Therefore, complexity if O(N) x O(N) = O(N2)

```
    //Method to sort the list using the bubble sort algorithm
```

Final code (repeated)

```
package ca1;

import java.io.File;
import java.util.*;

/**
 * @author Ruben
 */
public class ReadPeopleData {
    public static void main(String[] args) throws Exception {
        // Parsing and reading the CSV file data into the object array
          // I am not using the relative path provided because I had some
issues running
        // the code with it
        // String name = directory.getAbsolutePath() + "//people.csv";
        File directory = new File(

"C:/Users/Ruben/OneDrive/escritorio/programacion/nci/algorithms/CA1/CA1-alg
orithms/src/ca1");
        String name = directory.getAbsolutePath() + "/people.csv";

        // Array to store People objects - changed it a bit. In here people
variable
        // refers to an a new array of people containing
        // objects with a length of 10,000
        People[] people = new People[10];

        try (Scanner scanner = new Scanner(new File(name))) {
            // this will just print the header in CSV file
            scanner.nextLine();
            int i = 0;
            String sGetData;

              while (scanner.hasNextLine() && i < 10) { // && i < n - limit
loop to read only x records, change also line
                                                        // 19
                sGetData = scanner.nextLine();
                String[] data = sGetData.split(",");
                       people[i++] = new People(Integer.parseInt(data[0]),
data[1], data[2], data[3],
```

```
                                                        Integer.parseInt(data[4]),
Long.parseLong(data[5]));
              }

              // closes the scanner
          }

            // Convert the people array (containing People Objects) into a
MyArrayList
                              MyArrayList<People>    sortedPeopleList    =    new
MyArrayList<>(Arrays.asList(people));

        // Record start time in milliseconds
        long startTime = System.currentTimeMillis();

        // Call the bubbleSort() method
        sortedPeopleList.bubbleSort();

        // Record end time
        long endTime = System.currentTimeMillis();

        // Calculate the duration
        long duration = endTime - startTime;

        // Print the time taken
            System.out.println("Time taken for sorting: " + duration + "
milliseconds");

        // Print all entries in the sorted people list
        for (People person : sortedPeopleList) {
            System.out.println(person);
        }
    }
}

// People class
class People implements Comparable<People> {
    private int iId;
    private String sName;
    private String sSurname;
    private String sJob;
    private int iAge;
    private long lCredit;

    // People Constructor
```

```
    public People(int iInId, String sInName, String sInSurname, String
sInJob, int iInAge, long lInCredit) {
        this.iId = iInId;
        this.sName = sInName;
        this.sSurname = sInSurname;
        this.sJob = sInJob;
        this.iAge = iInAge;
        this.lCredit = lInCredit;
    }

    @Override
    public int compareTo(People myPeople) {
        // Compare by lCredit as per requirement
        // This section compares the column of the excel
        // my column is iCredit (6th column)
        return Long.compare(this.lCredit, myPeople.lCredit);
    }

    @Override
    public String toString() {
        return "Person [ID= " + iId + ", Name= " + sName + ", Surname= "
                + sSurname + ", Job= " + sJob + ", Age= "
                + iAge + ", Credit= " + lCredit + "]";
    }

    // Getters and setters
    public int getiId() {
        return iId;
    }

    public void setiId(int iId) {
        this.iId = iId;
    }

    public String getsName() {
        return sName;
    }

    public void setsName(String sName) {
        this.sName = sName;
    }

    public String getsSurname() {
        return sSurname;
    }
```

```java
    public void setsSurname(String sSurname) {
        this.sSurname = sSurname;
    }

    public String getsJob() {
        return sJob;
    }

    public void setsJob(String sJob) {
        this.sJob = sJob;
    }

    public int getiAge() {
        return iAge;
    }

    public void setiAge(int iAge) {
        this.iAge = iAge;
    }

    public long getlCredit() {
        return lCredit;
    }

    public void setlCredit(long lCredit) {
        this.lCredit = lCredit;
    }
}

// MyArrayList --> this implements the bubbleAlgorithm - followed example
in
// class
// Array class extends arrayList and adds a bubbleSort method
class MyArrayList<ElementType extends Comparable<ElementType>> extends
ArrayList<ElementType> {

    // default constructor
    public MyArrayList() {
        super();
    }

    // Constructor that accepts a collection
    public MyArrayList(Collection<? extends ElementType> c) {
        super(c);
    }
```

```
    // Method to sort the list using the bubble sort algorithm
    public void bubbleSort() {
        int iCount, jCount;
        // Outer loop to traverse through all elements
        for (iCount = 0; iCount < size(); iCount++) { // O(N) * 1 = O(N)
            // Inner loop for comparing adjacent elements
             for (jCount = 0; jCount < size() - 1 - iCount; jCount++) { //
O(M) * O(1) = O(M)
                // Get the current element and the next element
                ElementType elementAtjCount = get(jCount); // O(1) + O(1) =
O(2) = O(1) simplified
                 ElementType elementAtjCountPlus = get(jCount + 1); // O(1)
+ O(1) = O(2) = O(1) simplified

                // Compare the current element with the next element
                // If the current element is greater, swap them
                  if (elementAtjCount.compareTo(elementAtjCountPlus) > 0) {
// O(1)
                        // Swap element at position jCount with element at
position jCount + 1
                    swap(jCount, jCount + 1); // O(1)
                }
            }
        }
    }

    // Method to swap elements at two positions
    public void swap(int inPos1, int inPos2) {
            // Create two objects that will store the info from the two
positions
          ElementType objPos1 = get(inPos1); // O(1) + O(1) = O(2) = O(1)
simplified
          ElementType objPos2 = get(inPos2); // O(1) + O(1) = O(2) = O(1)
simplified

        // Remove element from position 1
        remove(inPos1); // O(N)

        // Insert objPos2 into position 1
        add(inPos1, objPos2); // O(N)

        // Remove element from position 2
        remove(inPos2); // O(N)

        // Insert objPos1 into position 2
        add(inPos2, objPos1); // O(N)
```

```
    }
}
```

Task 3

3. Write a Quick Sort algorithm that sorts the data using a column based on your student number. If two items have the same value sort based on column 1.
   You will receive higher marks for optimal (low run-time) solutions. **Highlight in the submission the reason why you chose your sorting algorithm with reference to the run-time complexity.** The sorting algorithm must be your own implementation. You will receive 0 marks for using an imported library to complete this task.

The quick sort algorithm is a highly efficient algorithm that uses the divide and conquer paradigm. It typically has an average time complexity of O(n log n), which makes it faster than other algorithms such as the bubble short.. However, the worst-case time complexity can degrade to O(N2), especially when the array is already sorted or nearly sorted, and the pivot selection is poor. Therefore the key points affecting complexity are the pivot selection and how sorted is the array.

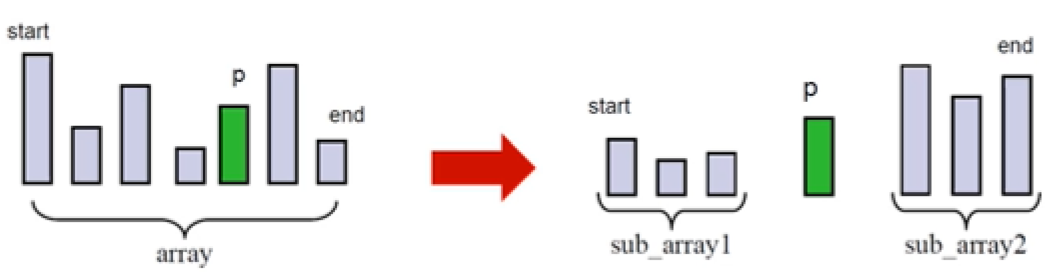
Quick sort has two main phases

- Partioning phase
- Sorting phase

Partioning phase

With the quick sort we should choose a pivot, in the sense that will arange the collection to the left and to the right in such a way that the collection to the left of the pivot to contain elements less greater than the pivot and to the right greater than the pivot.

But the subarray one and the subarray two, that elements are not sorted, so they are not in the right position.



Source: class slides.

The Quick algorithm is faster when the median value of the array is chosen as the pivot element, that is because the resulting partitions are of very similar size and each partition splits itself in two and thus the base case is reached very quickly.

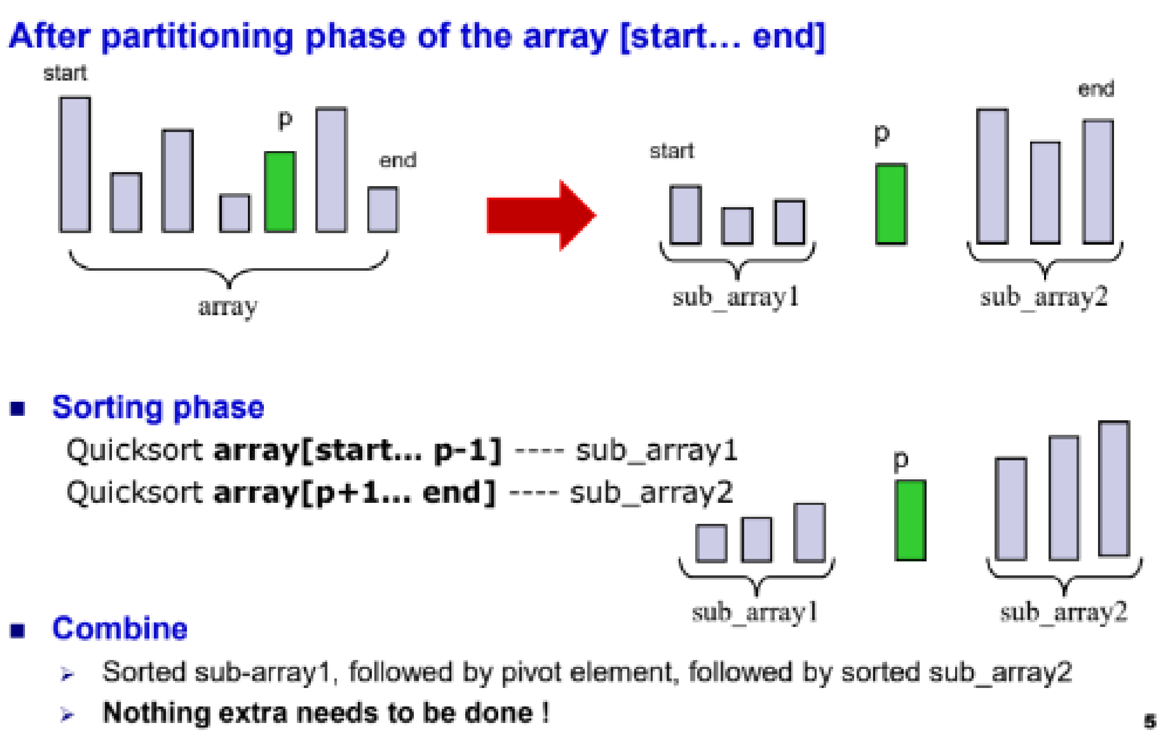
However, the Quick Sort algorithm becomes very slow when the array is already or close to being sorted. There is no efficient way for the computer to find the median element to use as the pivot. It's a contratuitive reasoning for a human, but not for an algorithm and therefore important to note. The more random the arrangement of the element in the array is, the faster the Quick Sort algorithm finishes. The performance of QuickSort ranges from O(N* logN) with the best pivot picked, to O(N2) with the worst pivot.

Therefore can be four options to choose a pivot:

- **Median Value (Ideal):** Results in partitions of nearly equal size, leading to optimal performance.
- **First Element:** Simple but can lead to poor performance with sorted or nearly sorted data.
- **Median of Three:** Chooses the median of the first, middle, and last elements, balancing simplicity and performance.
- **Random Pivot:** Reduces the likelihood of worst-case performance by randomizing the pivot choice.

Sorting phase

Each subarray is sorted independently. The process continues until the base case (subarrays of size 1 or 0) is reached.



Regarding the time complexity I added in the code some comments about the Big O complexity. Also here for reference:

Variable initialization:

int iUp, iDown // O(1)
ElementType pivot //O(1)

Reading the pivot element

pivot = get(iStart) // O(1)

Setting up and Down indexes

iUp = iStart //O(1)
iDown = iEnd //O(1)

Outer while loop (runs until iUp is less than iDown
while (iUp < iDown) - O(n) in total because in the worst case, iUp, and iDown move towards each other with each iteration

Inner while loops:

while (iUp < iEnd && iEnd && get(iUp).compareTo(pivot) <= 0 //O(1) for get and compareTo, runs o(n) times in total

while (iDown > iStart && get(iDown).compareTo(pivot) >= 0 //O(1) for get compareTo, runs O(n) time

Conditional swap inside the outer loop:

If (iUp < iDown) //O(1)

ElementType elementUp = get(iUp) //O(1)

set(iUp, get(iDown)) //O(1)

set (iDown, elementUp) //O(1)

Finally swap of pivot with element at iDown

set (iStart, get(iDown) //O(1)

set(iDown, pivot) //O(1)

return statement

return iDown //O(1)

Code implemented

```
package ca1;
```

```java
import java.io.File;
import java.util.*;

/**
 * @author Ruben
 */
public class readPeopleDataQuickSort {
    public static void main(String[] args) throws Exception {
        // Parsing and reading the CSV file data into the object array
        // I am not using the relative path provided because I had some issues
running
        // the code with it
        // String name = directory.getAbsolutePath() + "//people.csv";
        File directory = new File(

"C:/Users/Ruben/OneDrive/escritorio/programacion/nci/algorithms/CA1/CA1-algorith
ms/src/ca1");
        String name = directory.getAbsolutePath() + "/people.csv";

        // Array to store People objects - changed it a bit. In here people
variable
        // refers to an a new array of people containing
        // objects with a length of 10,000
        People[] people = new People[10000];

        try (Scanner scanner = new Scanner(new File(name))) {
            // this will just print the header in CSV file
            scanner.nextLine();
            int i = 0;
            String sGetData;

            while (scanner.hasNextLine() && i < 10000) { // Limit loop to read
only x records
                sGetData = scanner.nextLine();
                String[] data = sGetData.split(",");
                people[i++] = new People(Integer.parseInt(data[0]), data[1],
data[2], data[3],
                        Integer.parseInt(data[4]), Long.parseLong(data[5]));
            }
            // closes the scanner
        }

        // Convert the people array (containing People Objects) into a
MyArrayList
        MyArrayList<People> sortedPeopleList = new
MyArrayList<>(Arrays.asList(people));
```

```
        // Record start time in milliseconds
        long startTime = System.currentTimeMillis();

        // Call the quickSort() method
        sortedPeopleList.quickSort(0, sortedPeopleList.size() - 1);

        // Record end time
        long endTime = System.currentTimeMillis();

        // Calculate the duration
        long duration = endTime - startTime;

        // Print the time taken
        System.out.println("Time taken for sorting: " + duration + "
milliseconds");

        // Print all entries in the sorted people list
        for (People person : sortedPeopleList) {
            System.out.println(person);
        }
    }
}

// People class
class People implements Comparable<People> {
    private int iId;
    private String sName;
    private String sSurname;
    private String sJob;
    private int iAge;
    private long lCredit;

    // People Constructor
    public People(int iInId, String sInName, String sInSurname, String sInJob,
int iInAge, long lInCredit) {
        this.iId = iInId;
        this.sName = sInName;
        this.sSurname = sInSurname;
        this.sJob = sInJob;
        this.iAge = iInAge;
        this.lCredit = lInCredit;
    }

    @Override
    public int compareTo(People myPeople) {
```

```java
        // Compare by lCredit as per requirement
        // This section compares the column of the excel
        // my column is lCredit (6th column)
        return Long.compare(this.lCredit, myPeople.lCredit);
    }

    @Override
    public String toString() {
        return "Person [ID= " + iId + ", Name= " + sName + ", Surname= "
                + sSurname + ", Job= " + sJob + ", Age= "
                + iAge + ", Credit= " + lCredit + "]";
    }

    // Getters and setters
    public int getiId() {
        return iId;
    }

    public void setiId(int iId) {
        this.iId = iId;
    }

    public String getsName() {
        return sName;
    }

    public void setsName(String sName) {
        this.sName = sName;
    }

    public String getsSurname() {
        return sSurname;
    }

    public void setsSurname(String sSurname) {
        this.sSurname = sSurname;
    }

    public String getsJob() {
        return sJob;
    }

    public void setsJob(String sJob) {
        this.sJob = sJob;
    }
```

```java
    public int getiAge() {
        return iAge;
    }

    public void setiAge(int iAge) {
        this.iAge = iAge;
    }

    public long getlCredit() {
        return lCredit;
    }

    public void setlCredit(long lCredit) {
        this.lCredit = lCredit;
    }
}

// MyArrayList implements QuickSort algorithm
class MyArrayList<ElementType extends Comparable<ElementType>> extends
ArrayList<ElementType> {

    // default constructor
    public MyArrayList() {
        super();
    }

    // Constructor that accepts a collection
    public MyArrayList(Collection<? extends ElementType> c) {
        super(c);
    }

    // Method to sort the list using the QuickSort algorithm
    public void quickSort(int iStart, int iEnd) {
        int iPivotIndex; // declare pivot of the index // O(1)
        if (iStart < iEnd) { // as long as the start is less than the end
            // we will select the pivot, rearrange elements in the partition,
            // such as as all elements from the left and all elements to the
            // right // O(1)
            /*
             * select pivot and re-arrange elements in two partitions such as
             * all array[start … p-1] are less than pivot = array [p] and
             * all array[p+1 … end] are >= pivot
             */
            iPivotIndex = partition(iStart, iEnd); // 0(N)

            // sort first partition of the array (from start to pivot_index-1)
```

```
            // We call recursively this function quickSort again from the start,
            // what we are doing here is move elements smaller than the pivot to
            // the left
            // and values higher than 1 to the right
            quickSort(iStart, iPivotIndex - 1); // T(n/2)
            quickSort(iPivotIndex + 1, iEnd); // T(n/2)
        } else // do nothing, the array has one element, so it's sorted
        {
            return; // o(1) this return statement is not really neccessary, but
it is there for
                    // clarity reasons
        }
    }

    // Second part, where we increment up or increment down until we find the
value
    // less or greater
    // than the pivot and then we swap up and down. If they cross each other, we
    // swap them down with the pivot
    // and we repeat all of this
    public int partition(int iStart, int iEnd) {
        int iUp, iDown; // O(1)
        ElementType pivot;

        // we read the element from the start, we pivot as the pivot value
        pivot = get(iStart); // O(1)

        // we swap the labels up and down to be start and end in the
        // beginning

        // set the UP and DOWN indexes
        iUp = iStart; // O(1)
        iDown = iEnd; // O(1)

        // We increment up as long as up is less than down and up is less that
end
        // and the value at up is less or equal to the pivot.
        // if true, increment up
        while (iUp < iDown) { // O(n) in total for the loop
            // increment UP index until found first element higher than pivot
            while (iUp < iEnd && get(iUp).compareTo(pivot) <= 0) { // O(1) *
O(n) in total
                iUp = iUp + 1; // O(1)
            }

            // decrement DOWN as long as we did not reach the beginning of
```

```
            // the array and the value at down is greater than the value of the
            // pivot
            while (iDown > iStart && get(iDown).compareTo(pivot) >= 0) { // O(1)
* O(n) in total
                iDown = iDown - 1; // O(1)
            }

            // if UP and DOWN indexes did not cross each other
            if (iUp < iDown) { // O(1)
                ElementType elementUp = get(iUp); // we take the element in iup
and store into elementUp
                // O(1)

                // we take the value from up and put it out element up
                // We declare a temporary value to swap the up and down, because
if
                // we simply take up and put it at down, then we would not have
a value
                // remaining at down. We need to temporary store in var value
between before
                // we overwrite it.
                set(iUp, get(iDown)); // element in down stored in iUp - so for
a split of second there at up and down
                                      // would have the same value
                // The value from down would be duplicated at up, so that's why
we save value
                // from up in line 219, so we get that value and put it that
down //O(1)
                set(iDown, elementUp); // element in up stored in iDown //O(1)
                // At up and down, we now have the down and up, up at down
            }
        }

        // all this block is repeated until up is less than down
        // when they did pass each other, or they cross each other, then we swap
the
        // pivot with the value up down
        // we read the value at down and put it at the start index where we know
we will
        // find the pivot and we get the
        // pivot and put it at the value down
        set(iStart, get(iDown)); // O(1)
        set(iDown, pivot); // O(1)
        return iDown; // O(1)
    }
}
```

```
// Time complexity O(n log n), worst O(N2)
```

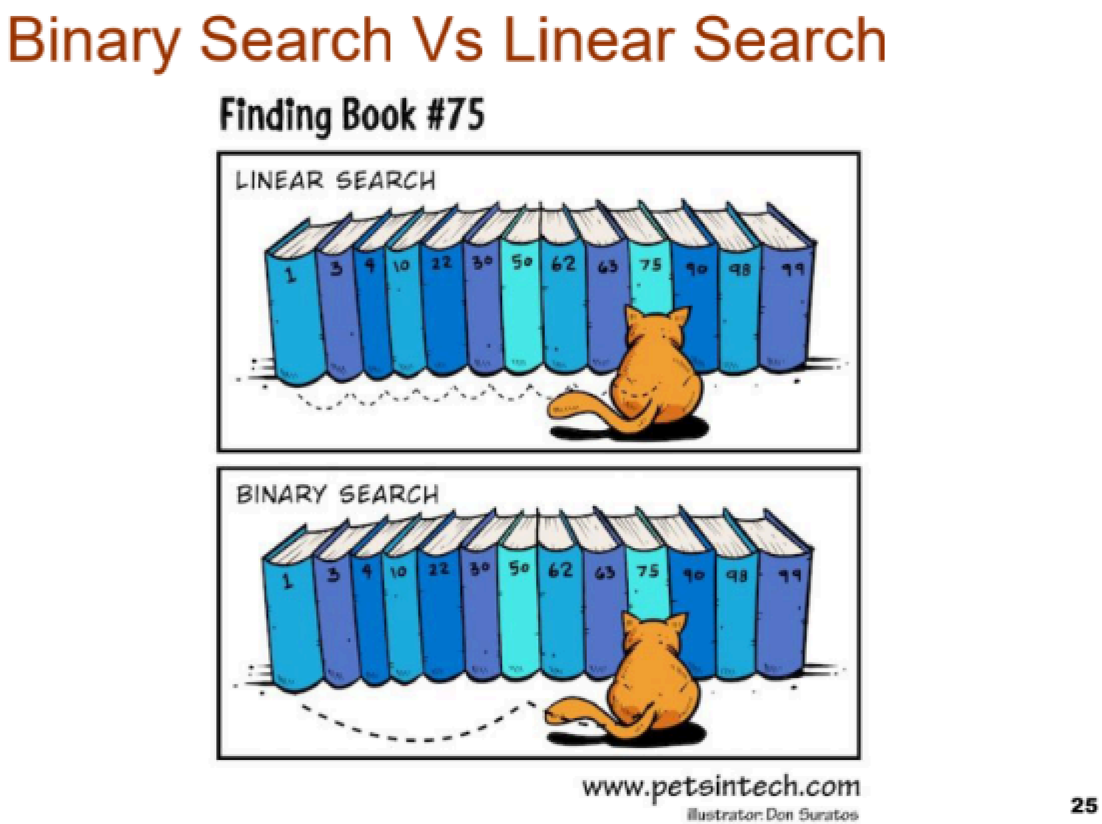
Task 4

4. Write a Binary Search algorithm that accepts a sorted column type and searches the data record from the dataset. For this you can use the sort() Java method to sort the elements in that column (can be any column between 2 and 4). If an element X is not found, display "X was not found in the Y list", where Y is the title of the chosen sorted column (e.g., Name, Country, Location, Age, etc.). If the element was found in the list, display "X was found in the Y list".

The Binary Search algorithm locates a target value within an array by repeatedly dividing the search interval in half. If the array is unsorted, it must be sorted first to apply binary search.

Binary Search operates in O(logN) time, making it suitable for large datasets due to its good time performance.

This image from the course slides can be easy to understand



Approaches used are basically two: with recursion and without recursion

Withour recursion pseudocode

```
int binarySearch (array, key, start, end)
{
      found = false;
while ((start <= end) and (found == false))
{
middle = (start + end) / 2;
if (array_ElemAt (middle) == key)
```

```
found = true;
else if (array_ElemAt (middle) < key)
start = middle +1;
else
end = middle - 1;
}
if (found == true)
return middle ;
else
return -1;
```

With recursion pseudocode

```
int binarySearch (array, key, start, end){
    middle = (start + end) / 2;
    if (array_ElemAt (middle) == key){ //base case
    result = middle;
    }else if (start == end){ //base case
    result = -1;
    }else{ //recursive case
    if (array_ElemAt (middle) > key){
    result = binarySearch(array, key, start, middle-1);
    }else{
    result = binarySearch(array, key, middle+1, end);
    }
    return result;
    }
```

How the algorithm works

- The algorithm starts with a pointer pointing to the beginning of the dataset(start), and the other at the end.
- The algorithm calculates the middle index of the current search segment
- The middle value is compared with the targeted value also called key
- If the middle value = key, the target is found
- If target is less than middle value, the algorithm moves to the left
- If target is more than middle value, algorithm moves to the right
- search end when the target is found.

Complexity

It's a very efficient dataset for large datasets, it already ran very quick in my local machine. This happens because in each step the algorithm divides the space, this reduction makes it run smoothly in a O(logN).

Here the notes of the big(O) for each line of code

```
// MyArrayList implements QuickSort algorithm

class MyArrayList<ElementType extends Comparable<ElementType>> extends
ArrayList<ElementType> {
```

```java
    // default constructor
    public MyArrayList() {
        super();
    }

    // Constructor that accepts a collection
    public MyArrayList(Collection<? extends ElementType> c) {
        super(c);
    }

    // Binary search method to search for a name in a sorted list of People
    public static int binarySearchByName(MyArrayList<People> list, String key,
int start, int end) {
        // Calculate the middle index of the current segment
        while (start <= end) { // O(log N)
            int middle = (start + end) / 2; // O(1)

            // Directly access and compare the name at the middle index with the
search key
            if (list.get(middle).getsName().equalsIgnoreCase(key)) { // O(1)
Base case: found the key
                return middle; // O(1) Return the index where the key was found
            } else if (list.get(middle).getsName().compareToIgnoreCase(key) > 0)
{ // O(1) Compare and search left half
                end = middle - 1; // O(1)
            } else { // Search right half
                start = middle + 1; // O(1)
            }
        }
        return -1; // Key not found
    }
```

```
}
```

Code implementation

```
package ca1;


import java.io.File;

import java.util.*;


/**

 * Author: Ruben

 */

public class readPeopleDataBinarySearch {

    // Parsing and reading the CSV file data into the object array

    // I am not using the relative path provided because I had some issues
running

    // the code with it

    // String name = directory.getAbsolutePath() + "//people.csv";

    public static void main(String[] args) throws Exception {

        // Parsing and reading the CSV file data into the object array

        File directory = new File(

"C:/Users/Ruben/OneDrive/escritorio/programacion/nci/algorithms/CA1/CA1-algorith
ms/src/ca1");

        String name = directory.getAbsolutePath() + "/people.csv";


        // List to store People objects - changed it a bit. In here peopleList
variable

        // refers to a new list of people

        List<People> peopleList = new ArrayList<>();


        // Use try-with-resources for the Scanner to ensure it is closed
properly

        try (Scanner scanner = new Scanner(new File(name))) {
```

```
            // this will just print the header in CSV file
            scanner.nextLine();
            String sGetData;

            while (scanner.hasNextLine()) { // Read all records
                sGetData = scanner.nextLine();
                String[] data = sGetData.split(",");
                peopleList.add(new People(Integer.parseInt(data[0]), data[1],
data[2], data[3],
                        Integer.parseInt(data[4]), Long.parseLong(data[5])));
            }
        } // closes the scanner

        // Convert the people list (containing People Objects) into a
MyArrayList
        MyArrayList<People> sortedPeopleList = new MyArrayList<>(peopleList);

        // Sort by Name
        sortedPeopleList.sort(Comparator.comparing(People::getsName));
        // sortedPeopleList - instance of MyArrayList<People>
        // sort - recommended for binary search algorithms
        // People:getsName = method reference for getting name

        // Get user input for the name to search
        try (Scanner inputScanner = new Scanner(System.in)) {
            System.out.print("Enter the name to search: "); // user type a name
            String searchName = inputScanner.nextLine();

            // Perform binary search on the sorted list for the input name
            int index = MyArrayList.binarySearchByName(sortedPeopleList,
searchName, 0, sortedPeopleList.size() - 1);
```

```
            // Output result based on the search
            if (index >= 0) {
                System.out.println("'" + searchName + "' was found in the Name list!");
            } else {
                System.out.println("'" + searchName + "' was not found in the Name list!");
            }

            // Print all entries in the sorted people list
            for (int i = 0; i < sortedPeopleList.size(); i++) {
                System.out.println(sortedPeopleList.get(i));
            }
        }
    }
}

// People class
class People implements Comparable<People> {
    private int iId;
    private String sName;
    private String sSurname;
    private String sJob;
    private int iAge;
    private long lCredit;

    public People(int iInId, String sInName, String sInSurname, String sInJob, int iInAge, long lInCredit) {
        this.iId = iInId;
        this.sName = sInName;
        this.sSurname = sInSurname;
        this.sJob = sInJob;
```

```
        this.iAge = iInAge;

        this.lCredit = lInCredit;

    }


    @Override

    public int compareTo(People myPeople) {

        return Long.compare(this.lCredit, myPeople.lCredit);

    }


    @Override

    public String toString() {

        return "Person [ID= " + iId + ", Name= " + sName + ", Surname= "

                + sSurname + ", Job= " + sJob + ", Age= "

                + iAge + ", Credit= " + lCredit + "]";

    }


    public int getiId() {

        return iId;

    }


    public void setiId(int iId) {

        this.iId = iId;

    }


    public String getsName() {

        return sName;

    }


    public void setsName(String sName) {

        this.sName = sName;

    }
```

```java
public String getsSurname() {
    return sSurname;
}

public void setsSurname(String sSurname) {
    this.sSurname = sSurname;
}

public String getsJob() {
    return sJob;
}

public void setsJob(String sJob) {
    this.sJob = sJob;
}

public int getiAge() {
    return iAge;
}

public void setiAge(int iAge) {
    this.iAge = iAge;
}

public long getlCredit() {
    return lCredit;
}

public void setlCredit(long lCredit) {
    this.lCredit = lCredit;
```

```
    }
}

// MyArrayList implements QuickSort algorithm
class MyArrayList<ElementType extends Comparable<ElementType>> extends
ArrayList<ElementType> {

    // default constructor
    public MyArrayList() {
        super();
    }

    // Constructor that accepts a collection
    public MyArrayList(Collection<? extends ElementType> c) {
        super(c);
    }

    // Binary search method to search for a name in a sorted list of People
    public static int binarySearchByName(MyArrayList<People> list, String key,
int start, int end) {
        // Calculate the middle index of the current segment
        while (start <= end) { // O(log N)
            int middle = (start + end) / 2; // O(1)

            // Directly access and compare the name at the middle index with the
search key
            if (list.get(middle).getsName().equalsIgnoreCase(key)) { // O(1)
Base case: found the key
                return middle; // O(1) Return the index where the key was found
            } else if (list.get(middle).getsName().compareToIgnoreCase(key) > 0)
{ // O(1) Compare and search left half
                end = middle - 1; // O(1)
```

```
            } else { // Search right half
                start = middle + 1; // O(1)
            }
        }
        return -1; // Key not found
    }
}
```

Part 2: Defensive Programming and Exception Handling (30 marks)

Task 5

5 Write a Java program that accepts a new record (with all the six fields) and adds it at the end of the record array, with a new consecutive ID number. (15 Marks)

Example input for people dataset:

(10001,Mark,Grant,Manager,33,472132554)

Example input for vehicles dataset:

(10001,Fiat,Petrol,Dublin,1634,942)

Example input for companies dataset:

(1001;Mango ltd.;Ireland;Euro;575325719;159)

Code

```
package ca1;

public class Records {
    int id;
    String name;
    String surName;
```

```
    String position;

    int age;

    int phoneNumber;


    public Records(int id, String name, String surName, String position, int
age, int phoneNumber) {

        this.id = id;

        this.name = name;

        this.surName = surName;

        this.position = position;

        this.age = age;

        this.phoneNumber = phoneNumber;

    }


    @Override

    public String toString() {

        return "Records{" + "id=" + id + ", name=" + name + ", surName=" +
surName + ", position=" + position + ", age=" + age + ", phoneNumber=" +
phoneNumber + '}';

    }

}
```

```
package ca1;


import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;
```

```java
public class RecordsApp {

    // List to store all records

    private List<Records> records = new ArrayList<>(); // we need to import
java.util.ArrayList and java.util.List


    // Initial value for the next unique ID

    private int nextId = 10001; // initial value


    // Method to add a new record

    public void addRecord(String name, String surName, String position, int age,
int phoneNumber) {

        // Create a new record with the next unique ID and provided details

        Records newRecord = new Records(nextId++, name, surName, position, age,
phoneNumber);

        // Add the new record to the list

        records.add(newRecord);

    }


    // Method to display all records using a classic for loop

    public void displayRecords() {

        // Loop through the list of records using a classic for loop

        for (int i = 0; i < records.size(); i++) {

            // Get the record at the current index

            Records record = records.get(i);

            // Print the record

            System.out.println(record);

        }

    }


    public static void main(String[] args) {


        // Create an instance of RecordsApp
```

```
        RecordsApp app = new RecordsApp(); // instance of the RecordsApp

        // Create a new Scanner object to read user input

        Scanner scanner = new Scanner(System.in); // we need to import
java.util.Scanner


        // Variable to control the loop for adding more records

        boolean continueAdding = true;


        // Loop to allow the user to add multiple records

        while (continueAdding) {

            System.out.println("Enter first name:");

            String firstName = scanner.nextLine(); // Read user input for first
name


            System.out.println("Enter last name:");

            String lastName = scanner.nextLine(); // Read user input for last
name


            System.out.println("Enter position:");

            String position = scanner.nextLine(); // Read user input for
position


            System.out.println("Enter age (needs to be a number):");

            int age = Integer.parseInt(scanner.nextLine()); // Convert user
input to int for age


            System.out.println("Enter phone number (needs to be a number):");

            int phoneNumber = Integer.parseInt(scanner.nextLine()); // Convert
user input to int for phone number


            // Add the new record with the provided details

            app.addRecord(firstName, lastName, position, age, phoneNumber); //
Call the add method
```

```
            // Ask if the user wants to add another record
            System.out.println("Do you want to add another record? (yes/no)");
            String response = scanner.nextLine().trim().toLowerCase();
            // Trim method to remove any leading or trailing spaces
            // toLowerCase method to convert response to lowercase for
comparison

            // Check if the response is not "yes"
            if (!response.equals("yes")) {
                continueAdding = false; // Exit the loop if the response is not
"yes"
            }
        }

        // Once the user finishes the loop, display all recorded information
        System.out.println("We have recorded the following information:");
        app.displayRecords(); // Print all records

        // Close the scanner to release system resources
        scanner.close();
    }
}
```

Task 6

6 - Write a Java Exception that handles special cases and communicates to users to correct the cases. A typical special case is that the "name" field cannot be empty or cannot contain digits only. The exception should generate a message similar to the following: “Person's name cannot be empty. It cannot have only digits! Please correct this!”

I built over the task 5 - being the final code

Some additions include - validateName method

```
   // Method to validate the name
```

```
    private void validateName(String name) throws InvalidNameException {

        if (name == null || name.trim().isEmpty() || name.matches("\\d+")) {
//trim method avoid to have "  " as value recorded. Name cannot be empty or
contain only digits

            throw new InvalidNameException("Person's name cannot be empty. It
cannot have only digits! Please correct this!"); // I neeeded to import
javax.naming, ide recommended it to me

        }

    }
```

Use of try and catch with the scanner / call to the method / integer validation → I know it was asked only to validate name but it was hurtful to see the program crashing when adding an integer on the age and telephone inputs. Also a good practice that ensures I understand the principles of defensive programming.

```
try {

                System.out.println("Enter first name:");

                String firstName = scanner.nextLine();


                // Validate name immediately after reading

                app.validateName(firstName);


                System.out.println("Enter last name:");

                String lastName = scanner.nextLine();


                System.out.println("Enter position:");

                String position = scanner.nextLine();


                System.out.println("Enter age:");

                int age = Integer.parseInt(scanner.nextLine());


                System.out.println("Enter phone number:");

                int phoneNumber = Integer.parseInt(scanner.nextLine());


                app.addRecord(firstName, lastName, position, age, phoneNumber);
```

```java
            } catch (InvalidNameException e) {
                System.out.println(e.getMessage());
            } catch (NumberFormatException e) { // I added this lines to avoid that the program crashed when a user enter a string in the age or phone inputs
                System.out.println("Invalid input for age or phone number. Please enter valid numbers.");
```

```java
package ca1;

public class Records {
    int id;
    String name;
    String surName;
    String position;
    int age;
    int phoneNumber;

    public Records(int id, String name, String surName, String position, int age, int phoneNumber) {
        this.id = id;
        this.name = name;
        this.surName = surName;
        this.position = position;
        this.age = age;
        this.phoneNumber = phoneNumber;
    }

    @Override
    public String toString() {
```

```
        return "Records{" + "id=" + id + ", name=" + name + ", surName=" +
surName + ", position=" + position + ", age=" + age + ", phoneNumber=" +
phoneNumber + '}';

    }

}
```

```
package ca1;


import java.util.ArrayList;

import java.util.List;

import java.util.Scanner;

import javax.naming.InvalidNameException;


public class RecordsApp {

    private List<Records> records = new ArrayList<>();

    private int nextId = 10001;


    // Method to add a new record

    public void addRecord(String name, String surName, String position, int age,
int phoneNumber) throws InvalidNameException {

        validateName(name); //validate method

        Records newRecord = new Records(nextId++, name, surName, position, age,
phoneNumber); //id increases everytieme a record is added

        records.add(newRecord);

    }


    // Method to validate the name

    private void validateName(String name) throws InvalidNameException {

        if (name == null || name.trim().isEmpty() || name.matches("\\d+")) {
//trim method avoid to have "  " as value recorded. Name cannot be empty or
contain only digits
```

```
            throw new InvalidNameException("Person's name cannot be empty. It
cannot have only digits! Please correct this!"); // I neeeded to import
javax.naming, ide recommended it to me
        }
    }

    // Method to display all records using a classic for loop
    public void displayRecords() {
        for (int i = 0; i < records.size(); i++) {
            Records record = records.get(i);
            System.out.println(record);
        }
    }

    public static void main(String[] args) {
        RecordsApp app = new RecordsApp();
        Scanner scanner = new Scanner(System.in);
        boolean continueAdding = true;

        while (continueAdding) {
            try {
                System.out.println("Enter first name:");
                String firstName = scanner.nextLine();

                // Validate name immediately after reading
                app.validateName(firstName);

                System.out.println("Enter last name:");
                String lastName = scanner.nextLine();

                System.out.println("Enter position:");
```

```
                String position = scanner.nextLine();


                System.out.println("Enter age:");

                int age = Integer.parseInt(scanner.nextLine());


                System.out.println("Enter phone number:");

                int phoneNumber = Integer.parseInt(scanner.nextLine());


                app.addRecord(firstName, lastName, position, age, phoneNumber);


            } catch (InvalidNameException e) {

                System.out.println(e.getMessage());

            } catch (NumberFormatException e) { // I added this lines to avoid
that the program crashed when a user enter a string in the age or phone inputs

                System.out.println("Invalid input for age or phone number.
Please enter valid numbers.");

            }


            System.out.println("Do you want to add another record? (yes/no)");

            String response = scanner.nextLine().trim().toLowerCase(); // as we
want the answer is "yes", YES or YeS will be translated as yes

            // because of the lowercase method and trim will be remove blank
spaces


            if (!response.equals("yes")) { //if the user write no or other word
different to yes, the program stops

                continueAdding = false;

            }

        }


        System.out.println("We have recorded the following information:"); //
when the program stops it shows the info added

        app.displayRecords();
```

```
        scanner.close();

    }

}
```