ALGORITHM AND DATA STRUCTURES

Exam 28 March – 25 April 2025

Candidate Number: 167, 208

Group Number: 1 | Course: PG400

TABLE OF CONTENT

Introduction to our Code Solution

Program

We have a class called Program with a method to read the data set:

```
public class Program {
    public double[] readLatitudeFromFile() throws FileNotFoundException {
        File file = new File("Files/worldcities.csv");
        ArrayList<Double> latitudeList = new ArrayList<>(); // We use
ArrayList first because we don't know how much space we need
        try (Scanner scanner = new Scanner(file)) {
            if (scanner.hasNextLine()) { // Skips first line with headings
                scanner.nextLine();
            }
            System.out.println("Starting to read file...");
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine();
                String[] values = line.split(",");
                try {
                    if (values.length > 2) {
                        // We want everything that is not a number, '.' or
'-' in column 3 (index 2)
                        String latitudeString =
values[2].trim().replaceAll("[^\\d.-]", "");(Vogella, 2021)
                        Double latitude =
Double.parseDouble(latitudeString);
                        latitudeList.add(latitude);
                    }
                } catch (NumberFormatException nfe) {
                    System.out.println("NumberFormat Exception: invalid
input string: " + nfe.getMessage());
                }
            }
        } catch (FileNotFoundException e) {
            System.out.println("Error opening file");
            e.printStackTrace();
        }
        // Converts ArrayList to a double[] array based on the final size
of the ArrayList
        double[] latitudeDoubles = new double[latitudeList.size()];
```

```
        for(int i = 0; i < latitudeList.size(); i++){
            latitudeDoubles[i] = latitudeList.get(i);
        }
        return latitudeDoubles;
    }
}
```

We start off by reading index 2 (latitudes) in the CSV file and add the data to an ArrayList. We did not know how much data input there would be, hence the need for a dynamic data structure.

MAIN

In Main, we initialized the Program class and used its method to read latitude data and sort it with Bubble Sort, Insertion Sort, Merge Sort and Quick Sort.

```
public class Main {
    public static void main(String[] args) {
        Program program = new Program();
        try {
            double[] latitudeData = program.readLatitudeFromFile();
            //BubbleSort.bubbleSortOptimized(latitudeData);
            //BubbleSort.bubbleSortUnoptimized(latitudeData);
            //InsertionSort.insertionSort(latitudeData);
            //MergeSort.mergeSort(latitudeData);
            //QuickSort.quickSort(latitudeData,"random");
            System.out.println("Sorted Latitudes:");
            for (double latitude : latitudeData) {
                System.out.println(latitude);
            }
            System.out.println("Number of comparisons: " +
QuickSort.count);

        } catch (FileNotFoundException e) {
            System.out.println("Error when calling method readFile" +
e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Problem 1: Bubble Sort

A.

The Bubble Sort method we implemented, sorts an array of double, so the ArrayList is converted to an array after the data has been read and we know how much memory to allocate. After the conversion (which is done in Program), we sort the array using both an optimized method, where we check if swaps were made, and un-optimized method with no checks.

To be able to see the difference in number of passes, the method also outputs number of passes made in the first loop.

Optimized Method

```
public class BubbleSort {
    public static void bubbleSortOptimized(double array[]){
        int n = array.length;
        // Counts to keep track of number of passes, comparisons and swaps
        int pass = 0;
        int comparison = 0;
        int swap = 0;
        // Optimizing: check if there is need to sort more
        boolean swapped;
        for(int i = 0; i < n - 1; i++){
            pass++;
            swapped = false;
            for(int j = 0; j < n - 1 -i; j++){
                comparison++;
                if(array[j] > array[j + 1]){
                    double temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    swap++;
                    swapped = true;
                }
            }
            if(swapped == false){
                break;
            }
        }
        System.out.println("Number of passes: " + pass);
        System.out.println("Number of comparisons: " + comparison);
```

```
        System.out.println("Number of swaps: " + swap);
    }
```

(Bubble Sort, Geeksforgeeks, 2024)

UNOPTIMIZED METHOD

```
public static void bubbleSortUnoptimized(double array[]){
    int n = array.length;
    int pass = 0;
    int comparison = 0;
    int swap = 0;
    for(int i = 0; i < n - 1; i++){
        pass++;
        for(int j = 0; j < n - 1 - i; j++){
            comparison++;
            if(array[j] > array[j+1]){
                double temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
                swap++;
            }
        }
    }
    System.out.println("Number of passes: " + pass);
    System.out.println("Number of comparisons: " + comparison);
    System.out.println("Number of swaps: " + swap);
}
```

(Bubble Sort, Geeksforgeeks, 2024)

B.

How Bubble Sort Works

The Bubble Sort algorithm is comparing adjacent elements and swaps them into the right order. It starts with index *(i)* 0, compares it with index *(i+1)* 1, and swaps them if the first is greater than the second. It then starts comparing index *(i+1)* 1 with index *(i+2)* 2, and with this method the greatest element in the unsorted section of the array, will be pushed towards the end of the array. The end of the array will become the sorted part. This is because the last element in each pass is already in its correct position, so the algorithm compares up to (n - i - 1) elements, where 'n' is the total number of elements. In a large data set, this will result in many operations.

Time Complexity

The worst and average case time complexity to sort this data set with Bubble Sort is $O(n^2)$ in both cases. This is due to the nested loop, resulting in $O(n) * O(n) = O(n^2)$ (Gupta, 2025, p. 11). It grows quadratically with the input size, as the code will enter both loops regardless. However, if the list is randomly ordered before sorting, the number of passes might decrease in the optimized method. It will still need to run the passes to sort the list, but if there is less to sort, it will check if swaps were made and stop the loop if not.

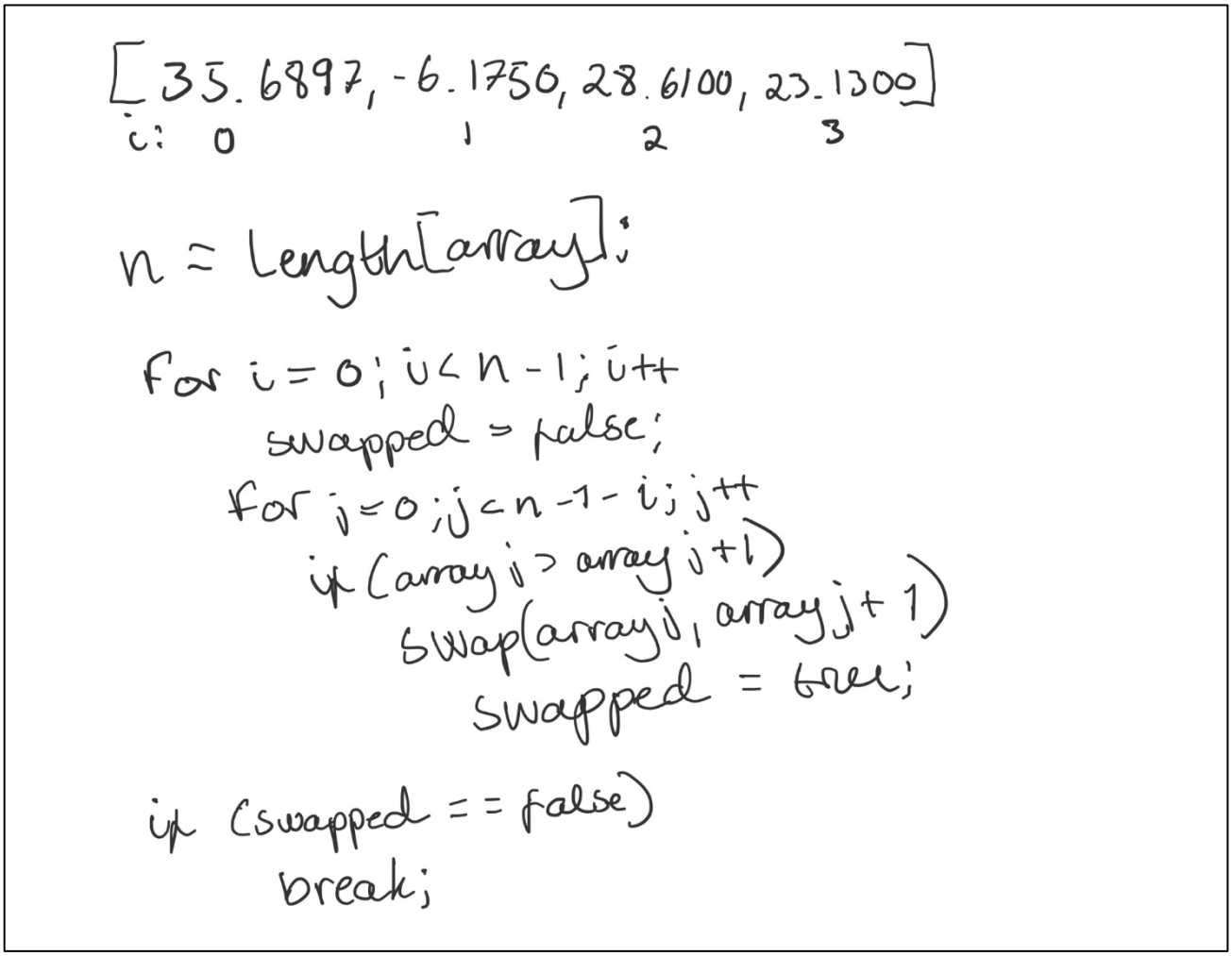
In the unoptimized method, the number of passes in a list of 47 868 elements is 47 867 and 47 841 in the optimized method. Therefore, in a large data set, there is some time to save with the optimized method. All in all, Bubble Sort is not the quickest sorting method for large data sets.

Sorting by Hand

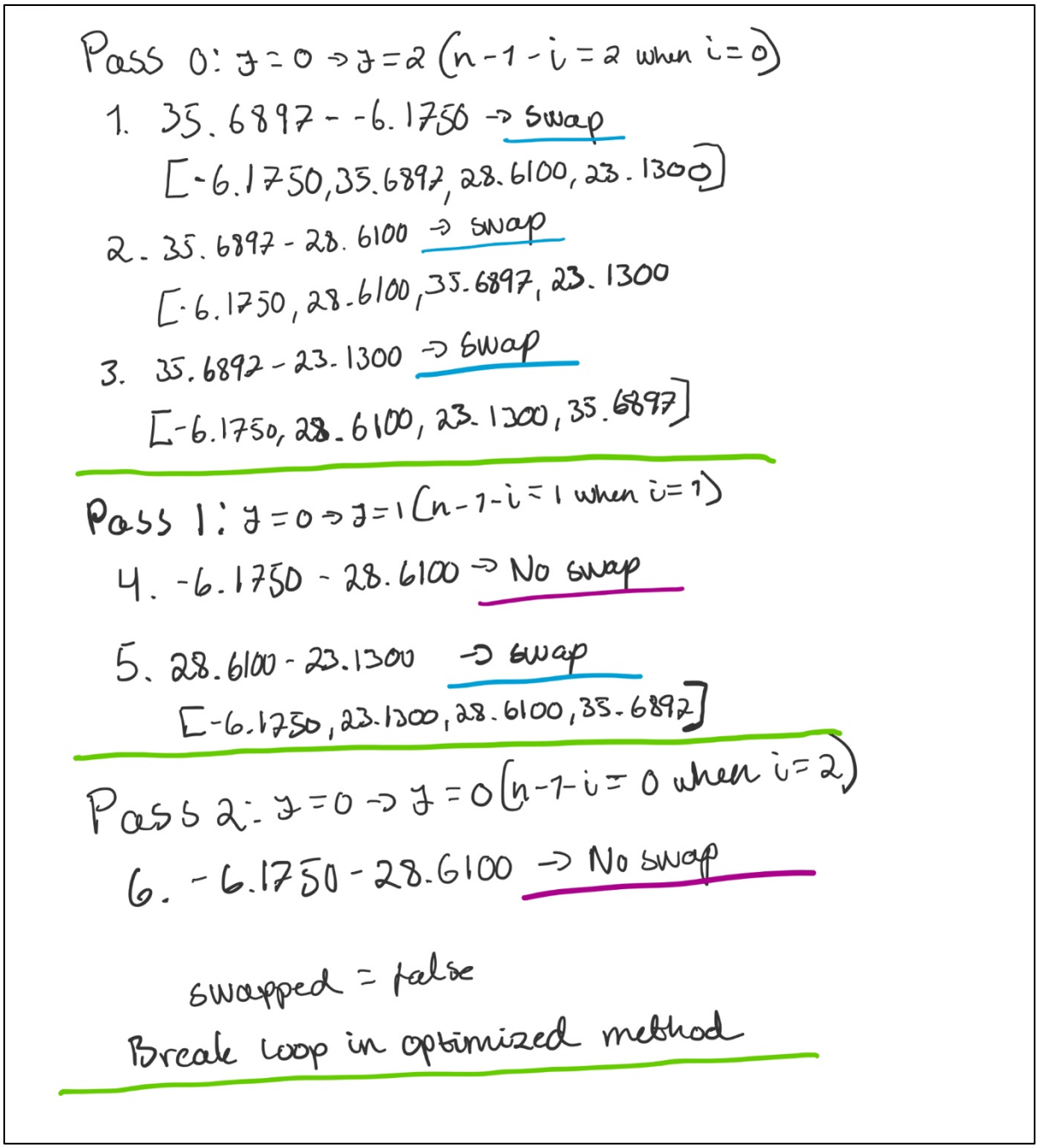
When we were first introduced to the Bubble Sort algorithm in this course, we found that it was quite easy to implement the algorithm in code, but understanding how it works was a little bit harder. Granted, this was when we were new to algorithms. Doing the algorithm on a small data set by hand, made us realize how time consuming this algorithm is compared to, for example, the Quick Sort algorithm. To test the Bubble Sort algorithm “manually”, we took the first four elements in the data set and sorted them out by hand. In this array, there are three passes of *i* and six iterations of *j* before the loop breaks.

SCREENSHOT OF PSEUDO CODE

Note on the last number in the array: wrote down the wrong number from the data set, but that is irrelevant for this demonstration – just one element more to sort.



SCREENSHOT OF THE ALGORITHM



ILLUSTRATIVE CHART

To further conceptualize this algorithm, we made a chart to see the number of passes the algorithm took. Even in a sorted list, the algorithm enters the nested loop, making the input data grow quadratically.

Bubble Sort Optimized		
	Unsorted	**Sorted**
Passes	47 841	1
Comparisons	1 145 648 427	47 867
Swaps	540 325 405	0

Sorting manually is time-consuming regardless of the chosen algorithm. However, with Bubble Sort especially, due to the lack of recursion, one has to repeatedly go through and rewrite the same array, even when it is mostly sorted. Doing it by hand illustrates how many steps the algorithm takes to sort a very small array. In larger data sets, this really highlights the noticeable efficiency of a chosen sorting algorithm.

Even with Bubble Sort on a data set with over 45 000 elements, the execution in Java is quite near to being instantly, since Java is compiled to bytecode, where the JVM, in a *very* short summary, makes the bytecode compact and optimized so that Java programs achieves fast execution (FreeCodeCamp, 2023). The Bubble Sort algorithm is best used on a small data set or to check if a data set is sorted, otherwise, it is mostly used for educational purposes in algorithms (Gupta, 2025, p. 7).

PROBLEM 2: INSERTION SORT

A.

Same as with Bubble Sort, the Insertion Sort method we implemented, sorts an array of double, so the ArrayList is converted to an array after the data has been read and we know how much memory to allocate. The conversion is done in Program.

CODE

```
package src;
public class InsertionSort {
    public void insertionSort(double array[]){
        int n = array.length;
        int pass = 0;
        int comparison = 0;
        int swap = 0;
        for(int i = 1; i < n; i++){
            pass++;
            double k = array[i];
            double j = i - 1;
            while(j >= 0 && array[(int) j] > k){
                comparison++;
                array[(int) (j + 1)] = array[(int) j];
                j = j - 1;
                swap++;
            }
            array[(int) (j + 1)] = k;
        }
        System.out.println("Passes: " + pass);
        System.out.println("Comparisons: " + comparison);
        System.out.println("Swaps: " + swap);
    }
}
```

(Insertion Sort, Geeksforgeeks, 2024)

B.

How Insertion Sort Works

The Insertion Sort algorithm sorts a list by inserting each element of an unsorted list into its correct position (Gupta, 2025, p. 6). The algorithm starts with index 1, as there is no point sorting if there is only one element in the array and the starting point is considered sorted. It then takes out each element by index one by one and stores it in a temporary variable *k*. It then compares and moves all greater elements than *k* to the right, leaving available space for *k* to be inserted in the left section of the array. The sorted section will always be the left side and elements from the unsorted section are inserted into the correct position in the sorted section of the array, until fully sorted – much the same way one would sort out a deck of cards.

Time Complexity

The time complexity for this algorithm is best-case O(n) and worst-case $O(n^2)$. In best-case, the list is already sorted so *n* would be the number of elements in the list (Gupta, 2025, p. 6). In the Insertion Sort algorithm we have implemented, as long as *array[j]* > *k*, it will not enter the while loop, which keeps the algorithm from the quadratic growth of the worst-case scenario.

If the list is randomly sorted, it will also be $O(n^2)$, because it will have to run the necessary number of passes, comparisons and swaps according to the size of the input data anyway. Therefore, regarding worst-case scenario of time complexity, it is of little relevance if it is randomly sorted or in completely reverse order.

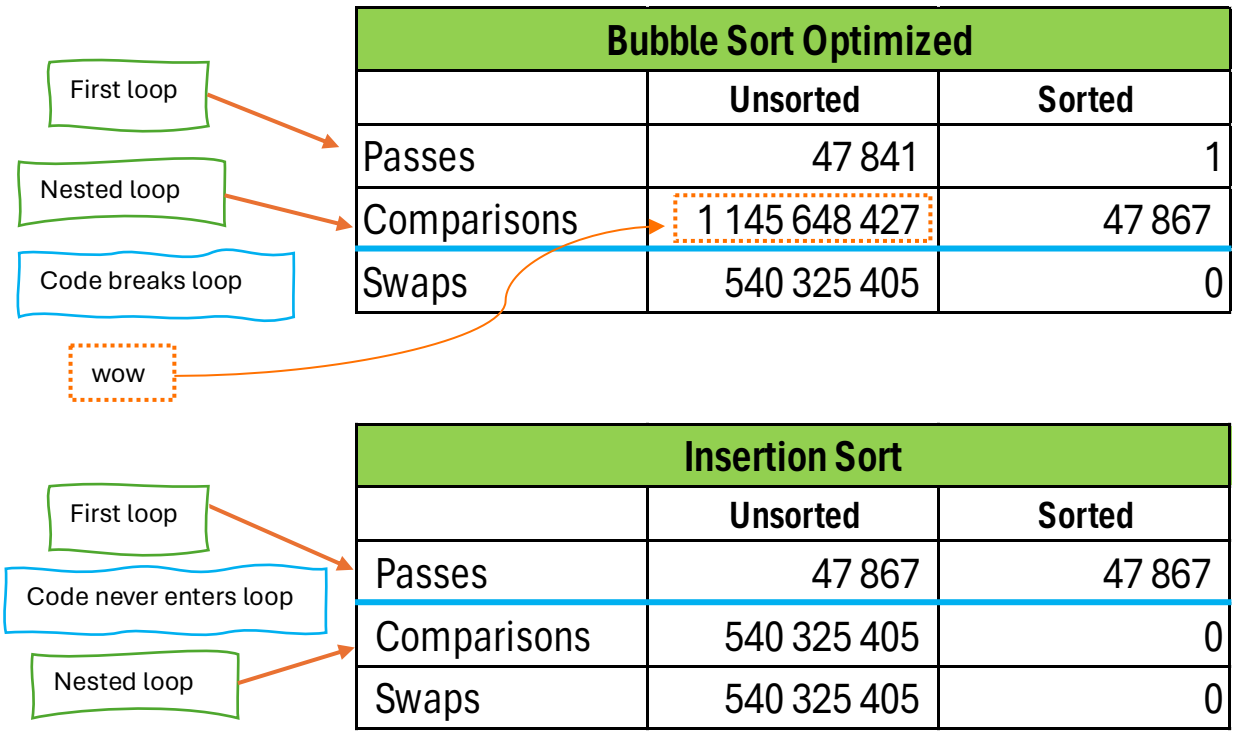
Difference in Unsorted and Sorted Array when Sorting

The Insertion Sort algorithm takes 47 867 (the length of the array - *n*) passes to sort this data set, which is the same as the unoptimized Bubble Sort algorithm. To really see the difference of best- and worst-case scenario, we tested using the sorting methods twice. With Insertion Sort, the second sorting had no comparisons and swaps. It just runs through the array in a line, resulting in only constant time - *n – 1* comparisons. This is what gives the best-case scenario the linear time complexity of O(n). Every new element in the algorithm is already in the correct position, so the condition in the while loop is never met.

In contrast, on a sorted list the optimized Bubble Sort algorithm would take at least one pass and *n* comparisons – giving quadratic time complexity as it always enters the nested loop. This is because the algorithm must take one whole pass to check that no swaps were made and that is why it needs to make the necessary comparisons. Additionally, we see that Bubble Sort makes more than twice the comparisons in the nested loop to sort the unsorted array, while Insertion Sort makes as many comparisons as it makes swaps.

ILLUSTRATIVE CHART

This chart illustrates in numbers how much more efficient Insertion Sort is compared to Bubble Sort:



Bubble Sort Optimized		
	Unsorted	Sorted
Passes	47 841	1
Comparisons	1 145 648 427	47 867
Swaps	540 325 405	0

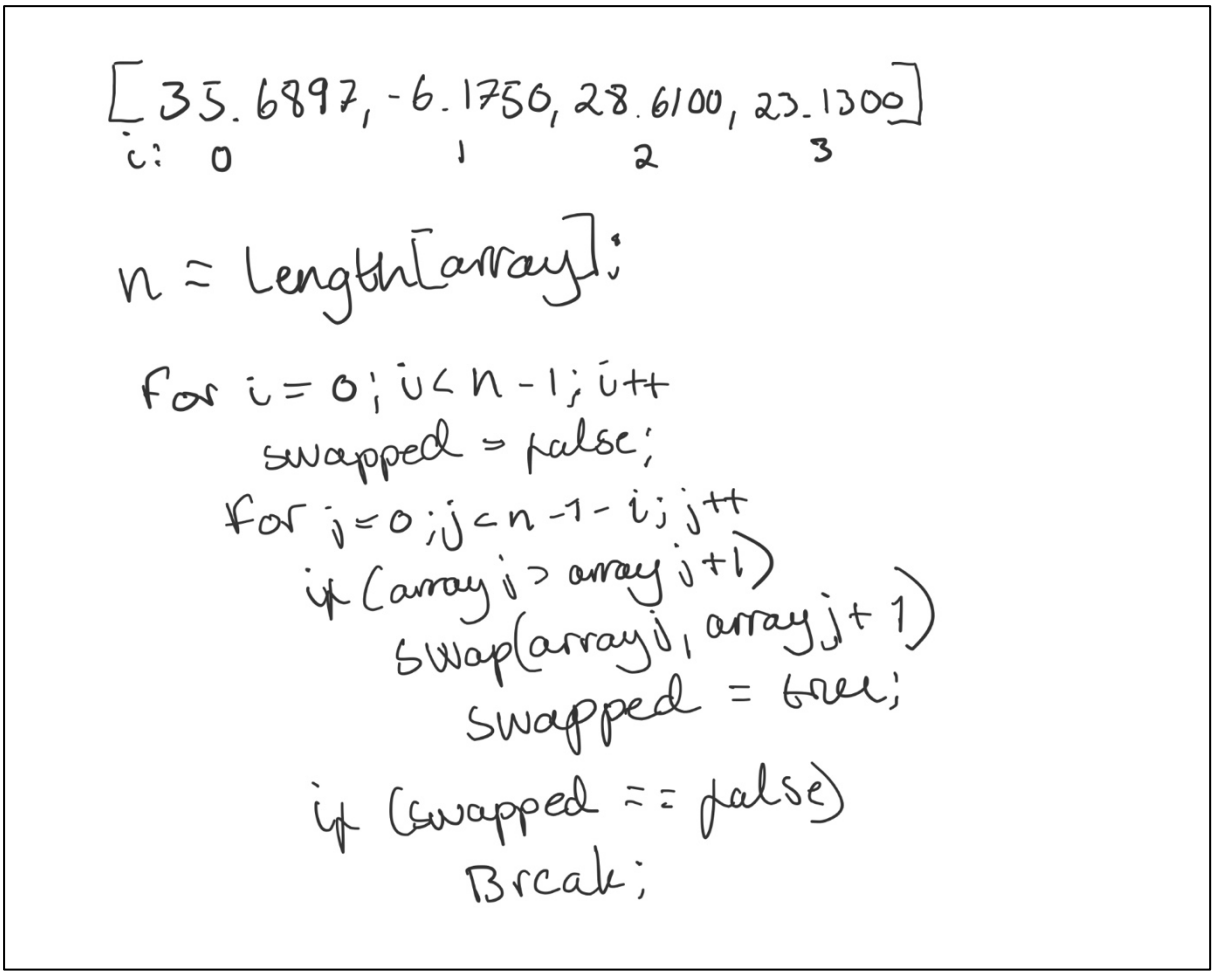
Insertion Sort		
	Unsorted	Sorted
Passes	47 867	47 867
Comparisons	540 325 405	0
Swaps	540 325 405	0

SORTING BY HAND

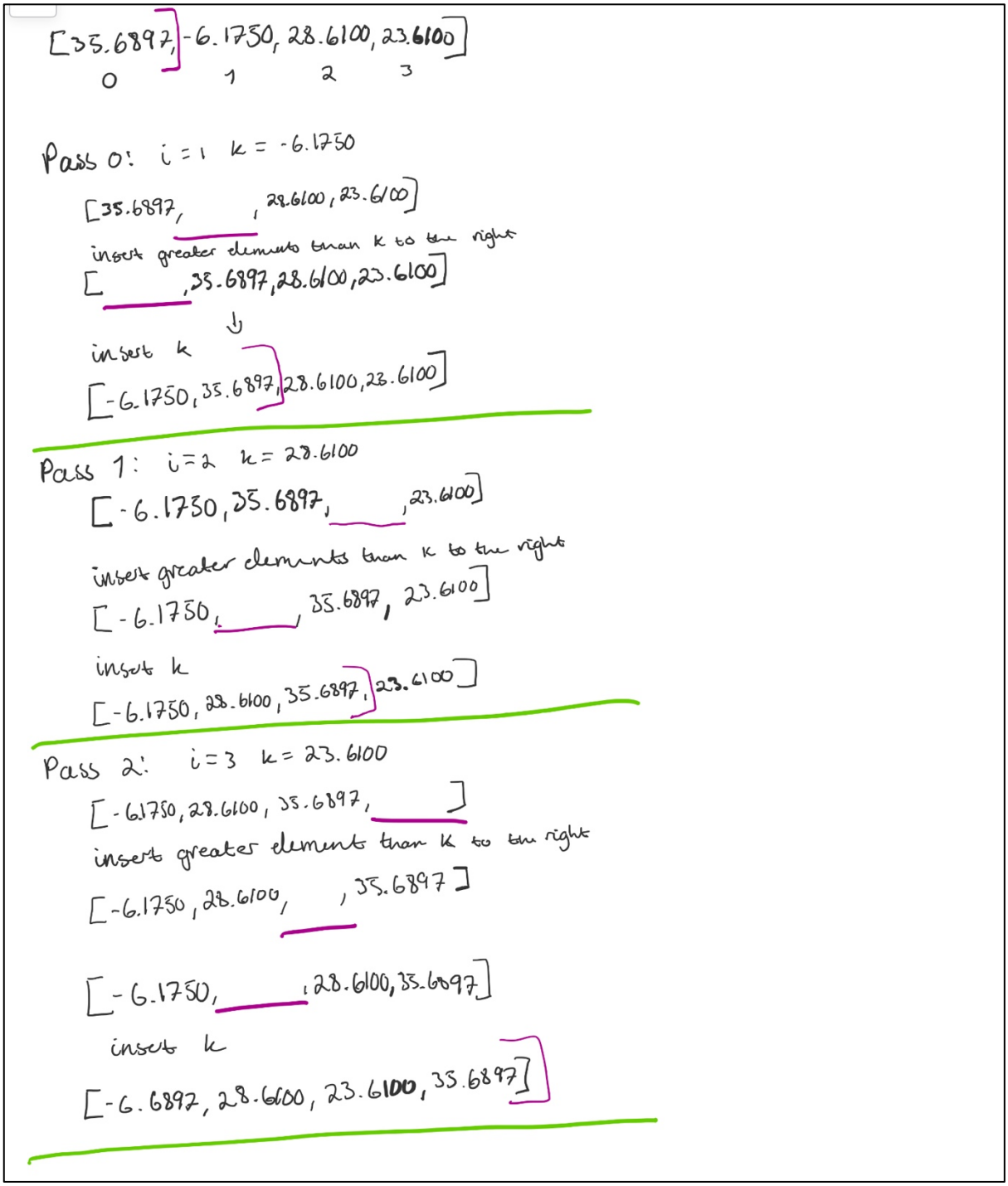
After learning about Bubble Sort, understanding the difference between these two algorithms was a bit tricky. They seemed like they do the same thing. Again, doing it by hand is what really makes one understand how Insertion Sort works differently to Bubble Sort. Bellow follows execution of the algorithm by hand, with the four first numbers from the dataset.

SCREENSHOT OF PSEUDO CODE

Note on the last number in the array: wrote down the wrong number from the data set, and decided to continue with the same array from the previous task.



SCREENSHOT OF ALGORITHM



PROBLEM 3: MERGE SORT

A.

Our Merge Sort implementation is divided into two main functions: sort and merge. The sort function is responsible for recursively dividing the array into smaller subarrays until each subarray contains only a single element. Once the array is fully divided, the merge function takes over.

The merge function compares elements from two sorted subarrays and merges them into a larger, sorted array. This merging process continues step by step, combining increasingly larger sorted subarrays, until the entire array is sorted.

CODE

```
package src;

public class MergeSort {
  public static int count;

  public static void mergeSort(double[] array) {
    // Vi deler opp arrayet i to - venstre og høyre.
    int arrayLength = array.length;

    // Stopper koden når det bare er ett element i arrayet.
    if (arrayLength <= 1) return;
    int midIndex = arrayLength / 2;
    double[] leftArray = new double[midIndex];
    double[] rightArray = new double[arrayLength - midIndex];

    //System.arraycopy(array, 0, leftArray, 0, midIndex);
    //System.arraycopy(array, midIndex, rightArray, 0, arrayLength - midIndex);
    //Vi fyller de to nye arrayene med verdier fra det originale arrayet.
    for (int i = 0; i < midIndex; i++) {
      leftArray[i] = array[i];

    }
    for (int i = midIndex; i < arrayLength; i++) {
      rightArray[i - midIndex] = array[i];

    }
```

```
    // Gjentar prosessen rekursivt for de to delene.
    mergeSort(leftArray);
    mergeSort(rightArray);

    // Slår sammen de to sorterte arrayene tilbake til ett.
    merge(array, leftArray, rightArray);
}

private static void merge(double[] array, double[] leftArray, double[] rightArray) {
    int leftSize = leftArray.length;
    int rightSize = rightArray.length;

    // Indekser for venstre, høyre og hovedarrayet.
    int leftIndex = 0;
    int rightIndex = 0;
    int key = 0;

    // Sammenligner elementer fra begge arrayene og fyller hovedarrayet med de minste verdiene først.
    while (leftIndex < leftSize && rightIndex < rightSize) {
        count++;
        if (leftArray[leftIndex] < rightArray[rightIndex]) {
            array[key] = leftArray[leftIndex];
            leftIndex++;
            key++;
        } else {
            array[key] = rightArray[rightIndex];
            rightIndex++;
            key++;
        }
    }
    // Legger til eventuelle gjenværende elementer fra venstre array.
    while (leftIndex < leftSize) {
        array[key] = leftArray[leftIndex];
        key++;
        leftIndex++;
    }
    // Legger til eventuelle gjenværende elementer fra høyre array.
```

```
    while (rightIndex < rightSize) {
      array[key] = rightArray[rightIndex];
      key++;
      rightIndex++;
    }
  }
}
```

B.

How Merge Sort Works

Merge Sort is an efficient algorithm that works well with large datasets and uses the “divide and conquer” method to solve the problem. In this approach, “divide” refers to splitting the data into smaller parts, while “conquer” involves merging them back together in sorted order. The essence of Merge Sort is that the data (array) is repeatedly divided in half until only one single element remains (and when only a single element remains in a subarray, that subarray is considered sorted) and merged back into a fully sorted output. This entire process happens recursively, meaning that the algorithm calls itself to solve smaller subproblems.

Recursion in Merge Sort is an important part that simplifies the logic of the algorithm by breaking the problem into smaller parts and sorting them step by step. The key components of recursion are the base case and the recursive case, where the base case stops the recursion, and the recursive case calls the algorithm itself (Geeksforgeeks, 2024).

The base case in Merge Sort occurs when the length of the array is one or zero, meaning there is a single element or the array is empty, which is already sorted.

The recursive case starts when the data is split in half, each half is sorted recursively, and then the sorted halves are merged back together.

Time Complexity

Time complexity of merge sort is O(n log n) in all cases, regardless whether this is worst, average or best case. This time complexity is called linearithmic, which combines both logarithmic and linear time complexity (Joshi, 2017).

Merge Sort is an out-of-place algorithm, which means it uses extra space for sorting (and in case of Merge Sort, to save temporary arrays while merging) (Joshi, 2017). It is also a stable algorithm, where the relative order of equal elements remains the same (Khandelwal, 2025).

There is no difference if the data is sorted or not, because both sorting and merging is executed regardless of order of elements. Therefore, whether the data is randomly ordered or already sorted, Merge Sort will still perform all its steps, without skipping any.

BENEFITS OF MERGE SORT

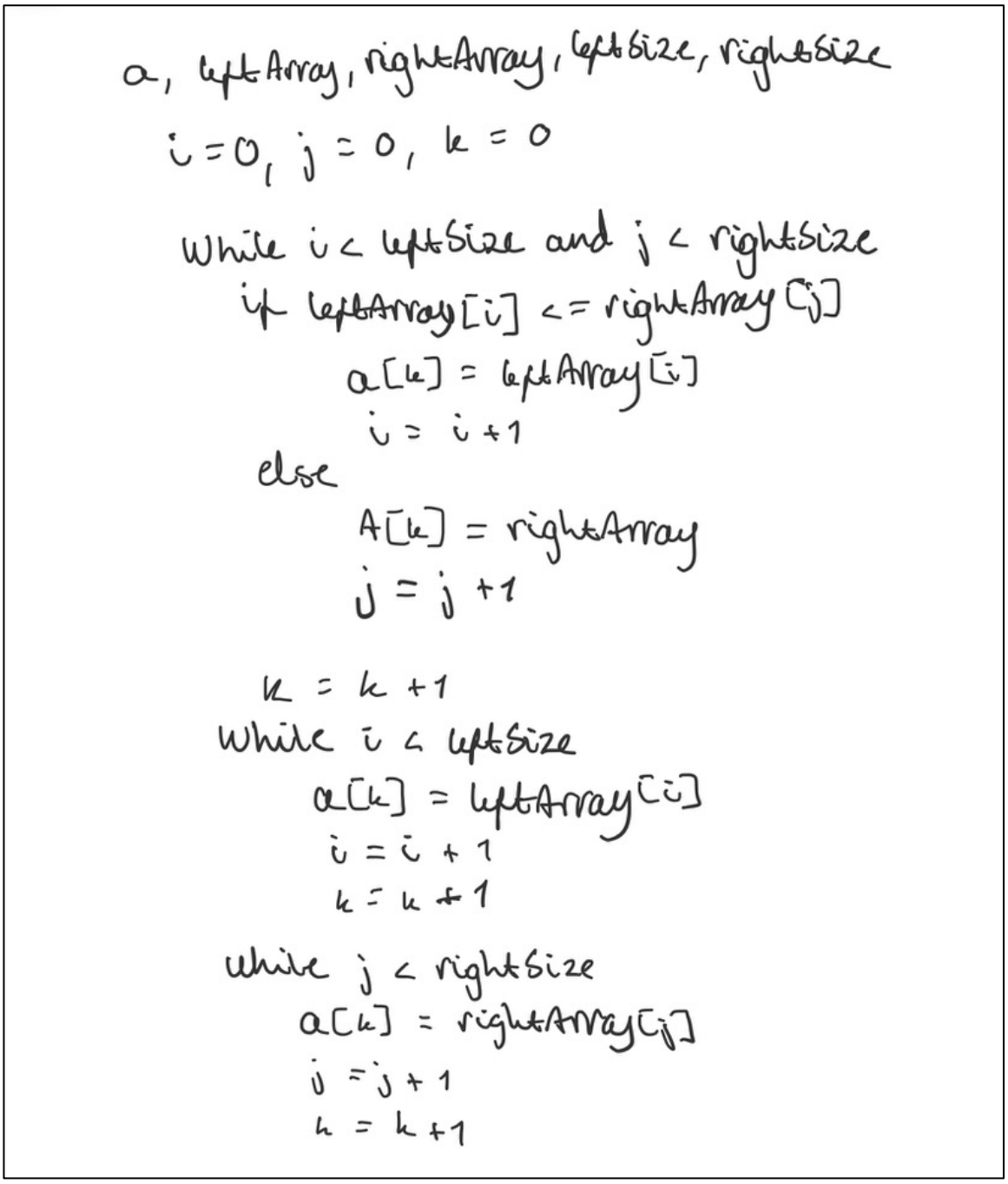
The time complexity of Merge Sort is always O(n log n), meaning it depends solely on the size of the input, not on the order of the elements. It works well when sorting is happening on large datasets. Merge Sort works especially well if the data is stored in a linked list. The reason for this is that most algorithms use random indexing to access values in a data structure, which isn't possible in a linked list. However, Merge Sort does not require random access, and instead splits and merges by manipulating node pointers, rather than copying data. Because of its stable functioning, Merge Sort is a great choice when sorting data that may have some equal elements in it.

In our case we store data in a linear array that consists of doubles, which holds latitude data of cities all over the world. There are X number of entries in this data set, so it is medium size.

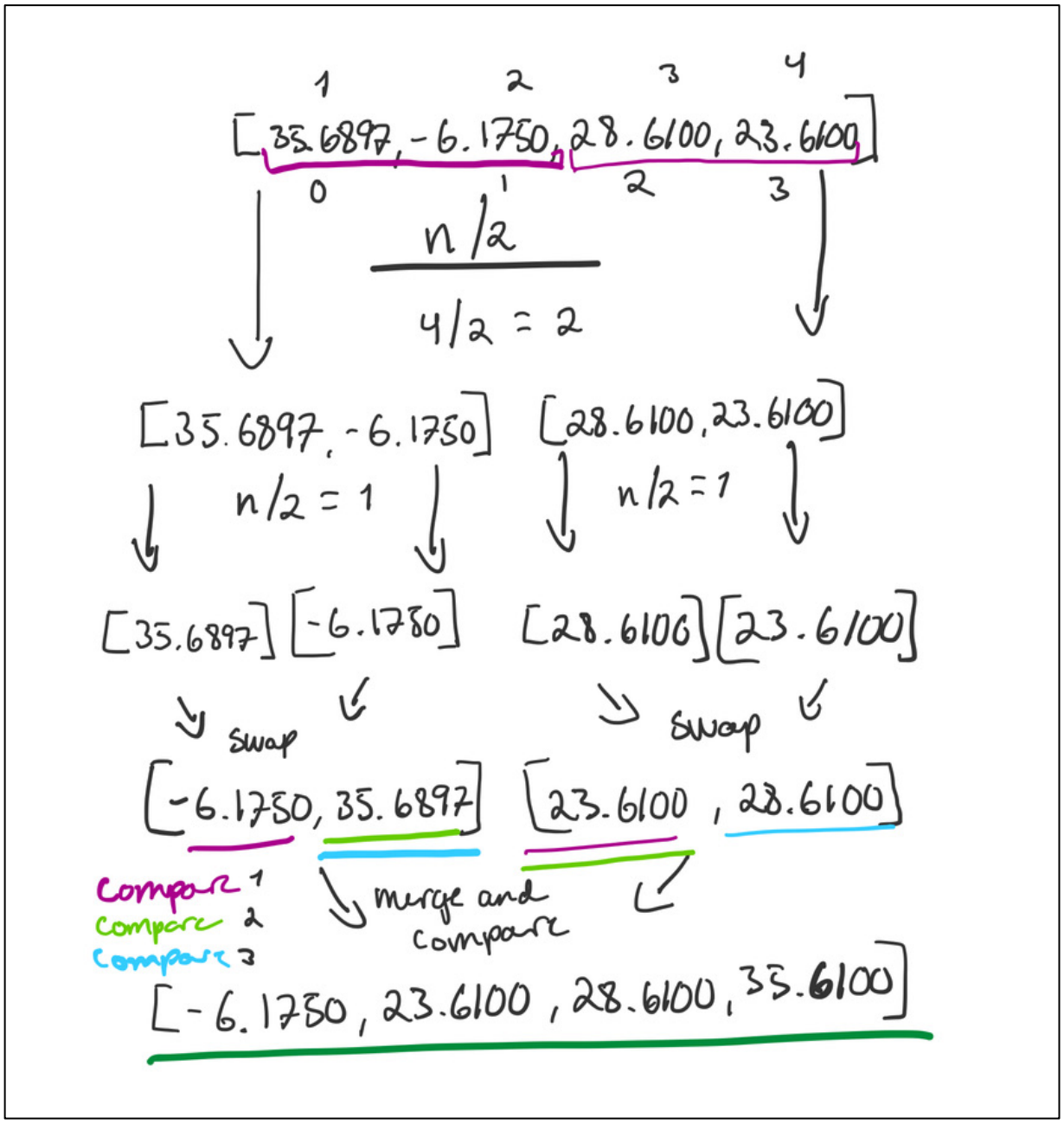
The benefits of Merge Sorts here is that the time complexity cannot be degraded based on the order of elements, but also since latitude can be equal for some cities - it is important for its sorting that if the latitude is the same, order is still relevant based on alphabetic order. Thus, Merge Sort can be an effective way to sort latitude data of this dataset.

SORTING BY HAND

SCREENSHOT OF PSEUDO CODE



SCREENSHOT OF ALGORITHM



PROBLEM 4: QUICK SORT

A.

The code for Quick Sort is divided into several functions, which makes it cleaner and easier to understand. One of the main functions is the partition function, which takes the pivot position as a parameter and sorts the elements around the pivot. It uses a swap function, which we also defined separately. This function takes two elements in the array and swaps them.

Another key function is the "quicksort" function itself, which recursively sorts the array. The recursion stops when there is nothing left to sort (i.e., when the subarrays contain zero or one element).

We also implemented a helper function for Quick Sort. This function calls the main "quicksort" function with a specified start and end index for the array. When this helper function is called from main, the user only needs to provide the array and the pivot position as arguments, making it easier to use.

CODE

```
package src;

public class QuickSort {
  static int count = 0;
  // QuickSort-funksjon som kan brukes i andre klasser.
  // Den setter automatisk start- og sluttindeksen for sorteringen.
  public static void quickSort(double[] array, String pivotPosition) {
    quickSort(array, 0, array.length - 1, pivotPosition);
  }

  // Hovedfunksjonen for QuickSort som deler opp arrayet og sorterer delene rekursivt.
  public static void quickSort(double[] array, int startIndex, int endIndex, String pivotPosition) {
    // Stopper rekursjonen hvis det ikke er noe mer å sortere.
    if (startIndex >= endIndex) return;

    int pivotIndex = partition(array, startIndex, endIndex, pivotPosition);
    quickSort(array, startIndex, pivotIndex - 1, pivotPosition);
    quickSort(array, pivotIndex + 1, endIndex, pivotPosition);
  }

  // Funksjon som velger pivot basert på posisjonen og organiserer elementene.
  // Elementer mindre enn pivot plasseres til, venstre, og større elementer til høyre.
```

```
    public static int partition(double[] array, int startIndex, int endIndex, String pivotPosition) {
        int pivotIndex;
        switch (pivotPosition) {
            case "first" -> pivotIndex = startIndex;
            case "last" -> pivotIndex = endIndex;
            case "random" -> pivotIndex = (int) (Math.random() * (endIndex - startIndex + 1) + startIndex);
            case null, default ->
                    throw new IllegalArgumentException("Please use valid pivot position(last, first or random).");

        }
        // // Flytter pivot til slutten av området som sorteres for å gjøre sammenligninger enklere.
        if (pivotIndex != endIndex) {
            swap(array, pivotIndex, endIndex); // Flytter pivot til slutten
        }
        double pivot = array[endIndex];
        int i = startIndex - 1;
        // Går gjennom elementene og flytter de mindre enn pivot til venstre.
        for (int j = startIndex; j < endIndex; j++) {
            if (array[j] < pivot) {
                i++;
                swap(array, i, j);
                count++;
            }
        }
        swap(array, i + 1, endIndex); // Plasserer pivot på sin endelige plass i den sorterte delen.

        return i + 1;
    }

    // Funksjon som bytter plass på to elementer i arrayet.
    public static void swap(double[] array, int i, int j) {
        double temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
}
```

B.

The solution for the 4th problem was made using Quick Sort – a sorting method that works by choosing a pivot element (an element in the array that is used as a reference to divide the rest of the elements into those that are larger and those that are smaller than the pivot). It is a very fast algorithm, and like Merge Sort, it is recursive and uses the divide and conquer strategy to sort data more efficiently.

The main idea is that we first choose a pivot, which is often moved to the end of the array to make the process simpler. Then each element in the array is compared to the pivot. If it is smaller, it goes to the left and if it is larger, it goes to the right. After partitioning, the pivot is placed in its correct position, with smaller elements to the left and larger elements to the right. After that, each subarray goes through the same process again and again, recursively, until everything is sorted. A great thing about Quick Sort is that it does not really need a separate combined step, because once everything is in the right place relative to each pivot, the array is already sorted. Since sorting in Quick Sort is happening recursively - the base case of it (or with other words case when recursion stops) is when size of subarrays will be 1 or 0, as arrays with one or no elements are already considered sorted. Then the array is sorted.

Time Complexity

The time complexity of Quick Sort can vary a lot, depending on things like the pivot choice and how the data is structured. In the best case, the pivot splits the array into two roughly equal halves every time, resulting in a time complexity of O(n log n) (Patel, 2024). In the average case, when the pivot is chosen randomly or in such a way that it splits the data reasonably well, the time complexity remains O(n log n). But in the worst case, when the pivot ends up being the smallest or largest element each time (like in an already sorted or nearly sorted data set), Quick Sort ends up making way too many comparisons, and the time complexity becomes $O(n^2)$. Quick Sort is an in-place and unstable algorithm. It does not require extra memory for temporary arrays (in-place), and it does not guarantee that the relative order of equivalent elements remains unchanged (unstable).

If the data set had already been sorted and we were using something like first-element or last-element pivot every time, it could cause performance to drop toward $O(n^2)$. If the data is randomized before sorting and the pivot is random for every sorting, then the time complexity is O(n log n). There is big difference between these two types of time complexities, so randomizing both pivot selection and the data is a very efficient way to stabilize the performance of this algorithm.

Benefits of Quick Sort

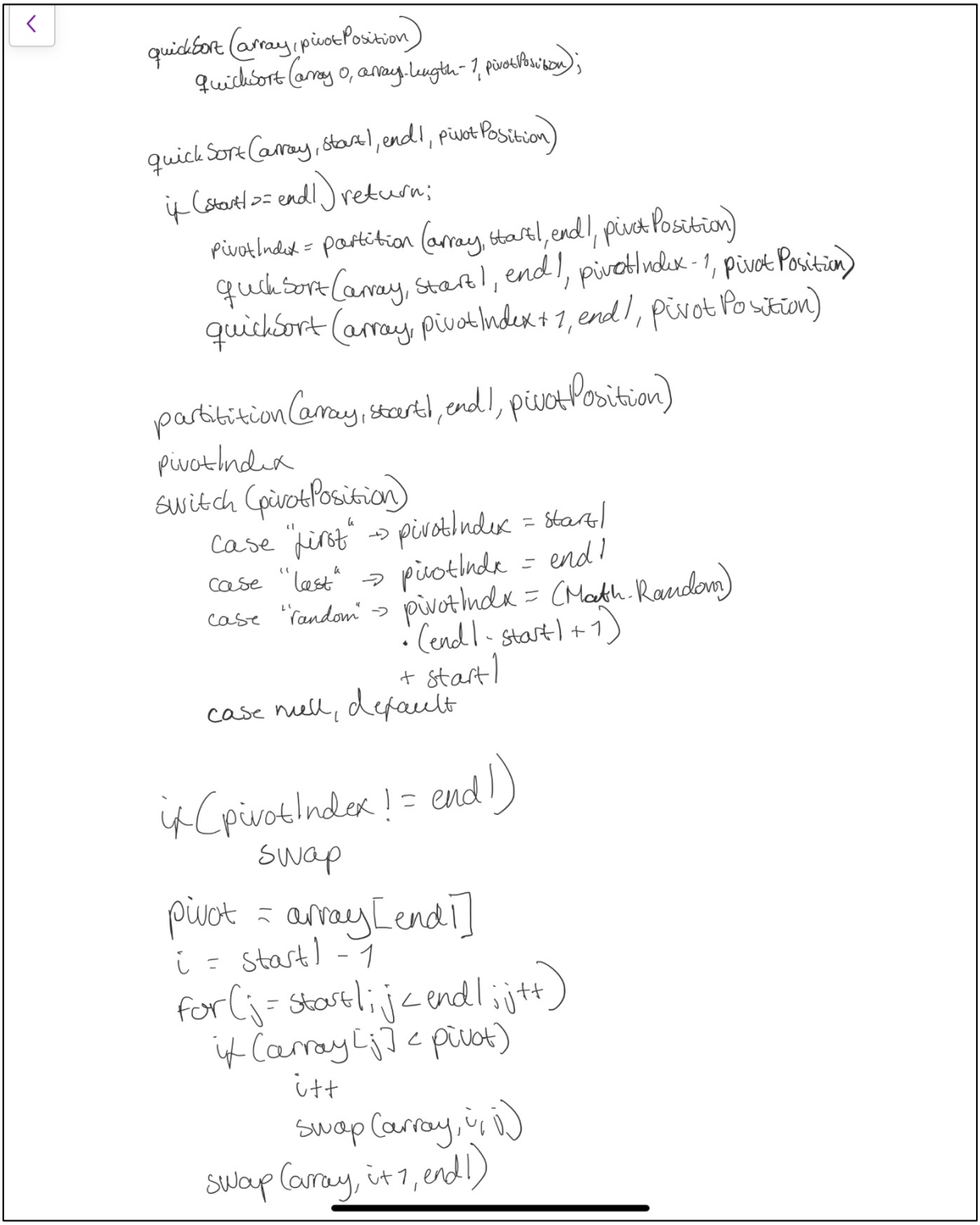
Quick Sort is a good choice when fast and efficient sorting is required. It is an unstable algorithm, so if preserving the original order of equal elements is not important, Quick Sort often outperforms algorithms like Merge Sort in practice. It also tends to make fewer comparisons on average, and since it does not require additional memory like Merge Sort, it is ideal for situations with limited resources.

In our case, we are sorting a medium-sized set of latitude values stored in a simple array of doubles. Since it is an array of primitive types, Quick Sort works very well — accessing and comparing elements is fast, and no extra space is needed.

Also, because we are currently working with a basic data structure (just an array of numbers) and not a more complex one where the order of equal elements might carry additional meaning (like names attached to each value), the fact that Quick Sort is unstable is not a problem in this scenario.
The data is randomly ordered, which is an ideal case for Quick Sort to perform near its average-case time complexity of O(n log n). All in all, Quick Sort is a great fit for this data set.

SORTING BY HAND

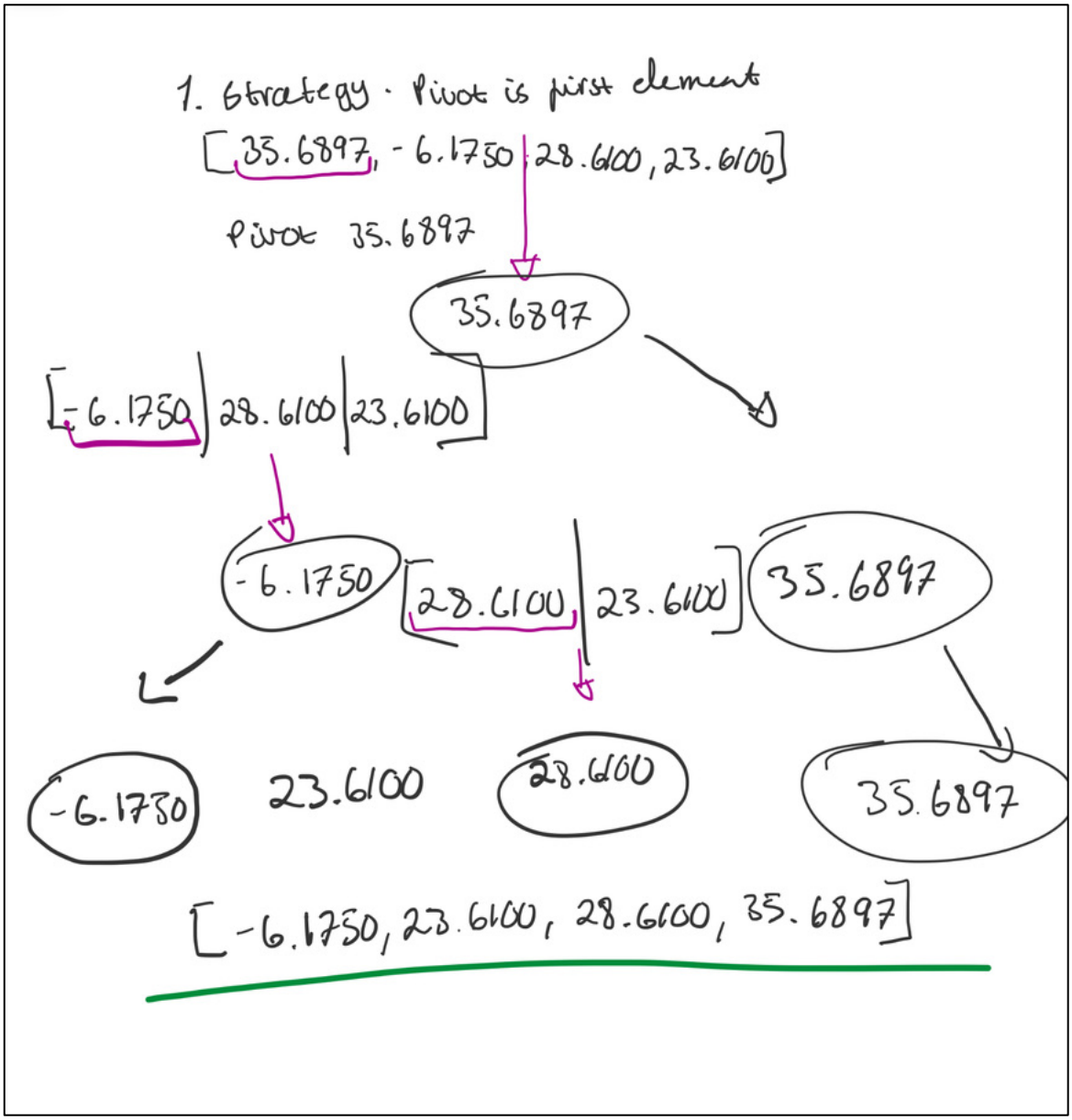
SCREENSHOT OF PSEUDO CODE



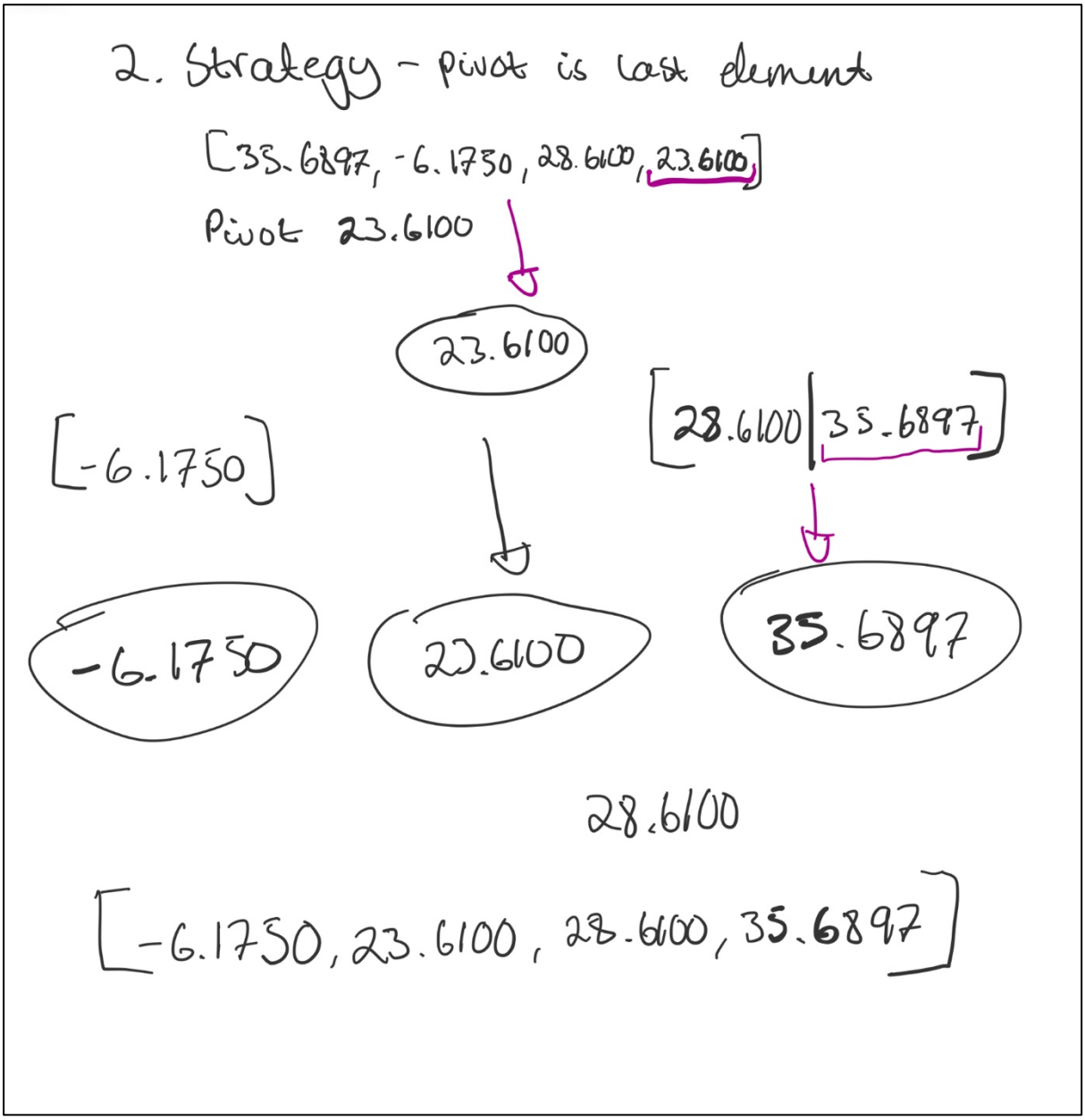
```
quickSort (array, pivotPosition)
    quickSort (array 0, array.length - 1, pivotPosition);

quickSort (array, start1, end1, pivotPosition)
 if (start1 >= end1) return;
     pivotIndex = partition (array, start1, end1, pivotPosition)
     quickSort (array, start1, end1, pivotIndex - 1, pivotPosition)
     quickSort (array, pivotIndex + 1, end1, pivotPosition)

partition (array, start1, end1, pivotPosition)
pivotIndex
switch (pivotPosition)
     case "first" -> pivotIndex = start1
     case "last" -> pivotIndex = end1
     case "random" -> pivotIndex = (Math.Random)
                    * (end1 - start1 + 1)
                    + start1
     case null, default

if (pivotIndex != end1)
       swap

pivot = array[end1]
i = start1 - 1
for (j = start1; j < end1; j++)
   if (array[j] < pivot)
         i++
         swap (array, i, j)
   swap (array, i+1, end1)
```

SCREENSHOT OF ALGORITHM

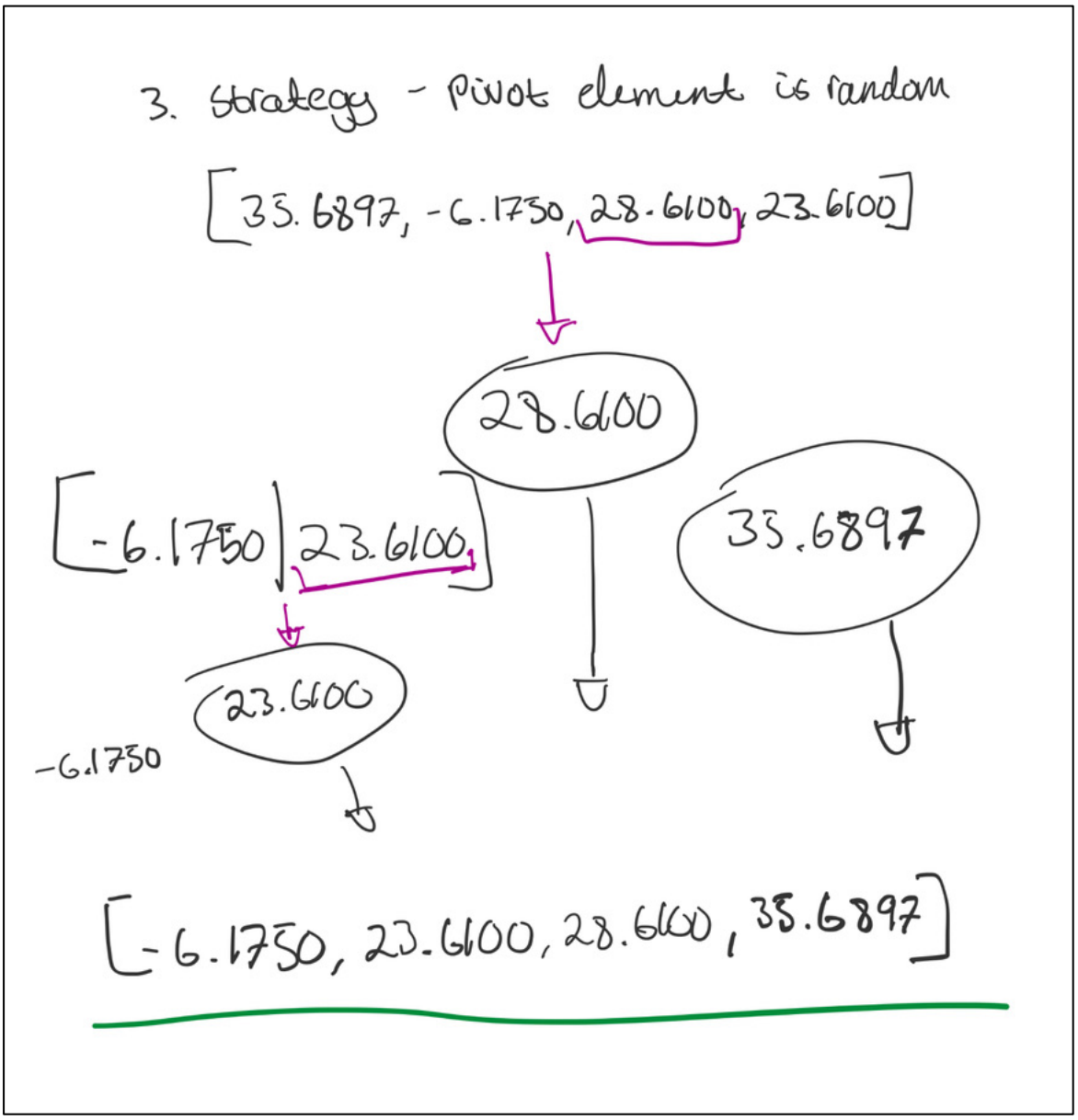
FIRST ELEMENT AS PIVOT ELEMENT



LAST ELEMENT AS PIVOT ELEMENT



RANDOM ELEMENT AS PIVOT ELEMENT



CONCLUSION

As a conclusion, it is easy to see that all the algorithms we researched, work very differently with this data set. Bubble Sort and Insertion Sort are simple and easy to implement in code, but slow. Both have a worst-case time complexity of $O(n^2)$, and they generally require many comparisons and swaps. However, there is an important difference between the two. Insertion Sort is significantly more efficient than Bubble Sort when the data is already sorted or nearly sorted. This is because Insertion Sort can detect already sorted elements and skip unnecessary operations, while Bubble Sort continues to compare all neighboring elements, even if the list is already sorted. Despite this, both algorithms become inefficient as the data set grows, and they are not practical choices for sorting large arrays.

Merge Sort and Quick Sort are much more efficient, both having an average-case time complexity of O(n log n). Merge Sort is a stable algorithm, meaning it maintains the relative order of equal elements. It is especially effective when working with data structures like linked lists, where random access is not available. However, Merge Sort requires additional memory for merging, which can be a disadvantage in cases where memory is limited.

Quick Sort, in contrast, is an in-place algorithm, meaning it does not require extra memory for sorting. It is not stable, so the order of equal elements is not preserved, but in our case, where we are working with a simple array of primitive types (double values representing latitude), this is not a problem. Additionally, Quick Sort is generally faster than Merge Sort in practical scenarios due to better cache usage and less comparisons. As long as the pivot is chosen well, Quick Sort performs very efficiently. In our specific case - sorting a medium-sized, randomly ordered array of latitude values, Quick Sort is a fast, memory-efficient, and practical solution. The structure of the data plays to Quick Sort's strengths, and because stability is not a concern here, its limitations do not affect us. Merge Sort remains a strong alternative when working with linked lists or when stability is important, but for this scenario, Quick Sort is the better fit.

To conclude this report, we drew two charts, the first to illustrate the comparison of time complexity for all the sorting methods we have explored. The second chart illustrates each of the sorting algorithm's characteristics and behavior for different applications.

Average Case Comparasion

n	Bubble Sort $O(n^2)$	Insertion Sort $O(n^2)$	Merg Sort O	Quicksort
8	28	28	24	24
16	120	120	64	64
32	496	496	160	160
2^{10}	~524,288	~524,288	10,240	10,240
2^{20}	$\sim 5 \cdot 10^{11}$	$\sim 5 \cdot 10^{11}$	20,971,520	20,971,520

Algorithm	Stable	Recursive	In-place	Comparison
Bubble Sort	✓	✗	✓	✓
Insertion Sort	✓	✗	✓	✓
Merge Sort	✓	✓	✗	✓
Quick Sort	✗	✓	✓	✓

SOURCE REFERENCE

Baeldung. (2025, February 16). Java CSV File Array. Retrieved March 25, 2025, from https://www.baeldung.com/java-csv-file-array

Baeldung. (2025, February 12). String.replaceAll(). Retrieved March 25, 2025, from https://www.baeldung.com/string-replace-all

BoardInfinity. (n.d.). How to Convert ArrayList to Array in Java. Retrieved March 25, 2025, from https://www.boardinfinity.com/blog/how-to-convert-arraylist-to-array-in-java/#why-convert-arraylist-to-array

Dissanayaka,B.(Sep 20, 2021) In-place and Out-of-place Algorithms. Retrieved April 8, 2025, from https://medium.com/@bhagyabhagya/in-place-and-out-of-place-algorithms-47e6103511d5

FreeCodeCamp. (2023, September 24). Understanding Java Internals: Speed and Performance. Retrieved April 1, 2025, from https://www.freecodecamp.org/news/understanding-java-internals-speed-and-performance/

GeeksforGeeks. (2018, 26 October). Double.parseDouble() Method in Java with Examples. Retrieved March 25, 2025, from https://www.geeksforgeeks.org/double-parsedouble-method-in-java-with-examples/

Geeksforgeeks. (2024, October 22). Java Program for Bubble Sort. Retrieved March 28. 2025, from https://www.geeksforgeeks.org/java-program-for-bubble-sort/

Geeksforgeeks. (2024, October 22). Java Program for Insertion Sort. Retrieved March 28. 2025, from https://www.geeksforgeeks.org/java-program-for-insertion-sort/

GeeksforGeeks. (2024, November 16). Introduction to Recursion. GeeksforGeeks. Retrieved April 8, 2025, from https://www.geeksforgeeks.org/introduction-to-recursion-2/

Gupta, Rashmi, Kristiania University College. (2025). PG4200: Learning Outcome 3 – Understanding Algorithm Efficiency (p. 6 and 7). Retrieved from https://kristiania.instructure.com/courses/13392/files/1574884?module_item_id=541288

Gupta, Rashmi, Kristiania University College. (2025). PG4200: Learning Outcome 5 – Understanding Algorithm Efficiency (p. 11). Retrieved from https://kristiania.instructure.com/courses/13392/files/1558831?module_item_id=537246

Patel, H.(n.d.) Quicksort Algorithm: An Overview. Builtin. Retrieved April 8, 2025, from https://builtin.com/articles/quicksort

Joshi, V. (2017, June 27). Making sense of merge sort (Part 1). Medium. Retrieved April 4, 2025, from https://medium.com/basecs/making-sense-of-merge-sort-part-2-be8706453209

Joshi, V. (2017, June 27). Making sense of merge sort (Part 2). Medium. Retrieved April 4, 2025, from https://medium.com/basecs/making-sense-of-merge-sort-part-2-be8706453209

Khan Academy. (n.d.). Analysis of merge sort. Retrieved April 4, 2025, from https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

StackExchange. (2021, July 19). Complexity of n * (n-1): Should it be O(1) or O(log n)? Stack Exchange. Retrieved April 2, 2025, from https://cs.stackexchange.com/questions/142441/complexity-of-n-n-1-should-be-o1-or-olog-n

StackOverflow. (2014, July 28). Difference between e.getMessage() and e.getLocalizedMessage. Retrieved March 25, 2025, from https://stackoverflow.com/questions/24988491/difference-between-e-getmessage-and-e-getlocalizedmessage

StackOverflow. (2011, April 24). Convert String to Double in Java. Retrieved March 25, 2025, from https://stackoverflow.com/questions/5769669/convert-string-to-double-in-java

StackOverflow. (Oct 19, 2010). For Loop Increment by Double. Retrieved March 25, 2025, from https://stackoverflow.com/questions/3971434/for-loop-increment-by-double

StackOverflow. (2011, October 26). Add Values to Double ArrayList. Retrieved March 25, 2025, from https://stackoverflow.com/questions/7904694/add-values-to-double-arraylist

StackOverflow. (2017, November 16). Convert ArrayList to Double Array in Java. Retrieved March 25, 2025, from https://stackoverflow.com/questions/47324154/convert-arraylist-to-double-array-in-java

Vogella. (2021, July 28). Java Regular Expressions. Retrieved March 25, 2025, from https://www.vogella.com/tutorials/JavaRegularExpressions/article.html

W3Schools.(n.d.). Java Files - Read. Retrieved March 25, 2025, from https://www.w3schools.com/java/java_files_read.asp