Høyskolen Kristiania

**Date:** 28.03 – 25.04.2025

**Candidate numbers:** 5, 9, 16

**Group:** 20

Final Exam

# *Algorithms and Data Structures*

**Course code:** PG4200

**Course:** Algorithms and Data Structures

**Semester**: Spring 2025

## Table of Contents

# 1. Introduction

For this exam in PG4200 Data Structures and Algorithms, we have based our work on a dataset consisting of a large number of cities from around the world as of March 19, 2024 (SimpleMaps, 2024).

Our group has chosen to use Java as our programming language when working on the problems given in the exam, as this is a language we all felt comfortable with. We have also been using Java in our lectures during the semester, making it feel natural to continue using it during this exam.

Java is also a programming language that is ideal for implementing data structures and algorithms, seeing how it is portable, offers standard libraries with pre-built data structures and algorithms, handles memory management adeptly, and the robust type system gives protection when constructing applications (GeeksforGeeks, 2024).

We imported the data from the dataset we received in our exam (about 48,000 cities with various attributes like longitude and latitude), and the main goal is to sort this data using four different sorting algorithms given by the exam text: Bubble sort, Insertion sort, Merge sort, and Quicksort.

Even if all four algorithms behave differently, they can still be grouped in pairs based on their approach to sorting; two are comparison-based algorithms (Bubble and Insertion sort), and two use recursion with divide and conquer (Merge and Quicksort). We have then looked at the algorithms, how to implement them, and how they fare in sorting the dataset given in different scenarios.

# 2. Set Up

## 2.1 Dataset - Basic World Cities Database

Our working dataset, `worldcities.csv` (hereby referred to as Cities), is downloaded from "https://simplemaps.com/data/world-cities" and consists of a CSV file with 47,868 cities and towns from around the world. The data is correct and up to date as of March 19, 2024 (SimpleMaps, 2024).

The Cities dataset includes 11 different attributes for each city. Attributes included are city name, city name in ASCII (for non-Latin based alphabets), latitude, longitude, what country the city is located in, alpha-2 and alpha-3 country codes, name of the highest level administration region of the city/town (e.g. a US state or Canadian province), population number and a unique ID generated by SimpleMaps.

We have not verified the accuracy of the dataset for the cities listed, as this is not part of the exam requirements. Also, even if some of the data in the dataset is false, it makes no change to our results when it comes to us being able to sort the data based on the latitude.

## 2.2 Data Structure for Working With Dataset

To save the data from Cities, we created a Java `record City` to hold the attributes for each city. This was done as records are an effective and fast way to store immutable data, as we will not be making any changes to the data from Cities, just sorting the order of the different record objects that we have saved.

```
public record City(String name, String asciiName, double latitude,
                   double longitude, String country, String iso2,
                   String iso3, String adminName, String capital,
                   int population, long id) { }
```

*Code: From src/main/java/City.java*

A record differs from a normal Java class by eliminating extra boilerplate code needed to save data. A record only needs the data type and name of a field, and the Java compiler will then create the constructor, public getters, and an equals method for each field for us. Even if several IDEs can generate these things, they won't automatically update our classes should we add a field, while the record will when compiling (Albano, 2024).

We then used the `CSVReader` class from the `OpenCSV` library to read the `worldcities.csv` file (Comma Separated Values) in a structured way. It is fast, efficient, and lets us quickly read the file line by line without needing to work commas inside field data and not having to write code to remove quotation marks around each data point.

In the beginning, we considered using `BufferedReader` (from `java.io.BufferedReader`) together with `String.split(",")`, but this had us working a lot to get around problems when it came to reading the data (i.e., city names that included comma) that CSVReader fixed automatically.

```java
public static ArrayList<City> readFile() {
   // Path to CSV file
   String worldCitiesFile = "src/main/resources/worldcities.csv";
   // Create a new ArrayList to store all elements from CSV file
   ArrayList<City> worldCities = new ArrayList<>();
   try (CSVReader reader = new CSVReader(new
FileReader(worldCitiesFile))) {
   String[] row;
   // Skips the header row
   reader.readNext();
   // Loop through the CSV file one row at a time until end of file
   while ((row = reader.readNext()) != null) {
      // Saves the information from the current row to a new City
element
      City city = new City(
         row[0],
         row[1],
         Double.parseDouble(row[2]),
         Double.parseDouble(row[3]),
         row[4],
         row[5],
         row[6],
         row[7],
         row[8],
         parseIntOrZero(row[9]),
         Long.parseLong(row[10])
      );
      // Adds the new City element to the City-list
      worldCities.add(city);
      }
   } catch (IOException e) {
      System.err.println("Error reading the file " + worldCitiesFile);
   } catch (CsvValidationException e) {
      System.err.println("Error validating the CSV file " +
worldCitiesFile);
   }
   // Return ArrayList with all cities from the CSV file
   return worldCities;
}
```

*Code: From src/main/java/TestAlgorithm.java*

This lets us set up a `readFile()` method that reads the full dataset, saves each line into a `City` record, and then adds them to an `ArrayList<City>` that is returned for the algorithms to use when sorting. All fields are saved as a string except for: latitude and longitude, which are saved as a `double`, population as an `int`, and the ID as a `long`.

To address some of the cities having a blank population field, we also added a `parseIntOrZero()` method to return 0 so we would always have data, should we want to sort by population at a later stage.

```
private static int parseIntOrZero(String data) {
   if (data == null || data.isBlank()) {
      return 0;
   }
   return Integer.parseInt(data);
}
```

*Code: From src/main/java/TestAlgorithm.java*

## 2.3 Warm-up of JVM Before Running Algorithms

As part of our algorithm work, we have tested the time the algorithms take to be able to see the difference in speed between them. While talking about Big-O and the time complexity of the algorithms, we also wanted to give simple examples with time that readers not used to thinking in the abstract might find easier to understand.

To ensure the best and most effective starting point for testing the timing, we set up the testing class to also include a warm-up set of the Java Virtual Machine (JVM). This was done by utilising the Just-In-Time (JIT) compiler, which "...improves the performance of Java™ applications by compiling bytecode to native machine code at run time." (IBM, 2024).

As the IBM (2024) documentation states, "At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time." This is done by the JVM keeping an invocation count on the number of times a method is called, and if it is called a certain amount, JIT compilation to bytecode is run.

Since the JIT compilation threshold helps the JVM start quickly by not compiling all methods at once, we warm up the JVM by executing the algorithm ten times on a fresh copy of the data before we start the timed tests.

```
public static void runJVMWarmUp(ArrayList<City> dataset,
       int warmUpLength, boolean print,String name,
       Consumer<ArrayList<City>> algorithm) {
   System.out.printf("*** Starting warm-up of JVM with %s ***\n", name);
   for (int i = 0; i < warmUpLength; i++){
      if(i == (double) (warmUpLength / 2)) System.out.println("***
Warm-up half way, be patient... ***");
      if(print) System.out.printf("Warm-up lap %d starting\n", i);

      ArrayList<City> copiedData = new ArrayList<>(dataset);
```

```
        algorithm.accept(copiedData);

        if(!copiedData.getFirst().name().equals("Puerto Williams")){
            System.out.printf("Error: Wrong output run %d!\n", i);
        }
    }
    System.out.printf("*** Finished warm-up of JVM with %s ***\n", name);
}
```

*Code: From src/main/java/TestAlgorithm.java*

The warm-up method is executed 10 times in a `for`-loop, with a quick message with `System.out.println` when starting, halfway, and at the end, so it is possible for us to know the program hasn't unexpectedly stopped.

We also added a simple test at the end of each iteration of the loop to check that the dataset had been sorted correctly, so it will output in the console should something not work as expected.

# 3. Bubble Sort

## 3.1 Implementation of Bubble sort for city latitudes

### Algorithm description

The bubble sort algorithm begins with the first element of the list. If the element is greater than the next element, they are swapped. The algorithm then compares the second and third elements, and if the second is greater than the third, they are swapped. This process continues for each pair of adjacent elements. If the element on the left side is less than the element on the right side, no swapping occurs. The sorting continues until the end of the list is reached, and the largest number is positioned on the right side. Then it goes back to the first element again until the dataset is sorted and no swap occurs during an iteration. (*Rosetta Code*, 2025)

The best use of bubble sort is that it allows students to learn sorting algorithms more easily compared to other sorting algorithms. "Although examples used in first year courses must be simple enough to be understandable and complex enough to be useful in a variety of situations,..."(Astrachan, n.d., s. 5)

### Difference between optimised and non-optimised bubble sort algorithm

An optimised bubble sort algorithm checks during every iteration if any swaps are made. A flag is set when a swap occurs, and if no swaps are made during a pass, the algorithm terminates early. This makes it faster in the best case scenario. The worst case, when the list is unordered, is the time complexity $O(n^2)$.

A non-optimised bubble algorithm continues to make passes through the entire list even if no swaps are needed. This means the best-case scenario and the worst-case scenario will always be $O(n^2)$.

### Explanation of the code

The method starts by getting the size of the list and initializing the swapped flag. The outer loop (i) iterates through the list, reducing the range with each pass. The inner loop (j) compares the latitude of adjacent cities and swaps them if necessary.

After the first iteration, the largest element is swapped to the (n-1)th index and after the next iteration the second is placed at the (n-2)th index. This process continues until no swaps occur, indicating that all elements are in the right order (*Enjoy Algorithms*, 2023.). The `iteration` and `swap` counters are implemented for gaining data for analytics about the algorithm.

Non-optimised bubble sort algorithm

```java
public static int[] bubbleSort(ArrayList<City> cities) {
   // Counter for analysis: iterations and number of total elements
swapped
   int iteration = 0;
   int swap = 0;
   // Outer loop - Will run max n-1 times
   for (int i = 0; i < cities.size() -1 ; i++) {
      iteration++;
      // Inner loop - Compares and swaps adjacent elements if needed
      for (int j = 0; j < cities.size() - i - 1; j++) {
         // Switch the order of the two elements if the largest is on
the left
         if (cities.get(j).latitude() > cities.get(j + 1).latitude()) {
            City temp = cities.get(j);
            cities.set(j, cities.get(j + 1));
            cities.set(j + 1, temp);
            swap++;
         }
      }
   }
   // Returns number of iterations and swaps that can be used for
analytics and graphs
   return new int[]{iteration, swap};
}
```

*Code: From src/main/java/task_1_bubblesort_nonoptimised.java*

The non-optimized bubble sort is missing a swapped flag. As a result, the method will go through the list one more time after the last swap to ensure that the list is fully ordered.

Optimised bubble sort algorithm

```java
public static int[] bubbleSort(ArrayList<City> cities) {
   // Counter for analysis: iterations and number of total elements
swapped
   int iteration = 0;
   int swap = 0;
   // Outer loop - Will run max n-1 times
   for (int i = 0; i < cities.size() - 1; i++) {
      iteration++;
      // Flag for detecting a swapping of elements
      boolean swapped = false;
      // Inner loop - Compares and swaps adjacent elements if needed
      for (int j = 0; j < cities.size() - i - 1; j++) {
      // Swap the current element if the latitude is larger than the
next
         if (cities.get(j).latitude() > cities.get(j + 1).latitude()) {
            City temp = cities.get(j);
            cities.set(j, cities.get(j + 1));
            cities.set(j + 1, temp);
            swap++;
            swapped = true;
         }
      }
      // If no elements swapped this pass the list is sorted and
finishes early
      if (!swapped) break;
      }
   // Returns number of iterations and swaps that can be used for
analytics and graphs
   return new int[]{iteration, swap};
}
```

*Code: From src/main/java/task_1_bubblesort_optimised.java*

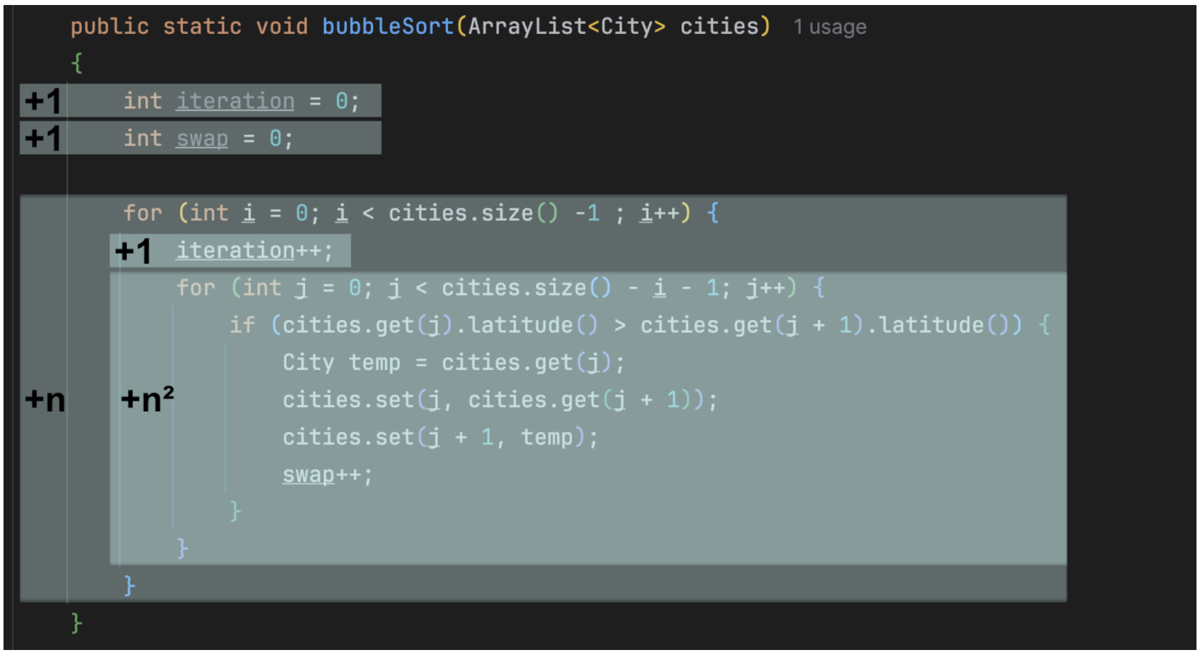
The swapped flag shows that this is an optimised bubble sort algorithm because it will terminate early when no swap is needed, indicating the list is already sorted.

## 3.2 Time and Space Complexity of Bubble Sort

### The Effect of Randomly Sorting the Dataset Before Sorting

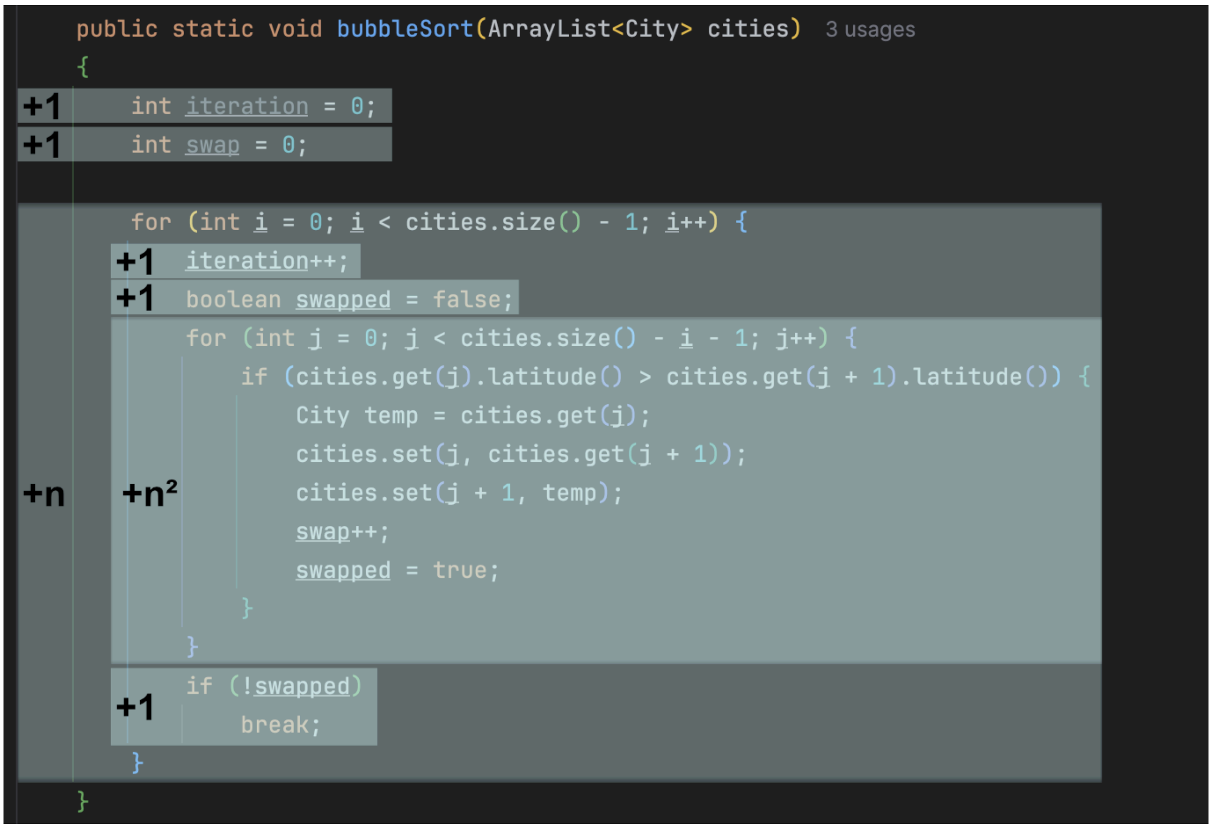
The time complexity of bubble sort is determined by the number of comparisons and swaps needed to sort the array; the algorithm must compare and potentially swap each pair of adjacent elements multiple times. Since we have two nested loops with the size n, the outer loop runs n - 1 times. For each iteration of the outer loop, the inner loop runs 1 - n times in the worst case (W3School, n.d.). Therefore, the time complexity in the worst case is $O(n^2)$.

The space complexity of bubble sort is $O(1)$ because it only requires a fixed amount of space for variables such as size and the flag in the optimized bubble sort code (Simplilearn, n.d.).



```
public static void bubbleSort(ArrayList<City> cities)  1 usage
{
    int iteration = 0;
    int swap = 0;

    for (int i = 0; i < cities.size() -1 ; i++) {
        iteration++;
        for (int j = 0; j < cities.size() - i - 1; j++) {
            if (cities.get(j).latitude() > cities.get(j + 1).latitude()) {
                City temp = cities.get(j);
                cities.set(j, cities.get(j + 1));
                cities.set(j + 1, temp);
                swap++;
            }
        }
    }
}
```

*Image: Calculating time complexity for our non-optimized bubbleSort method.*



```
public static void bubbleSort(ArrayList<City> cities)  3 usages
{
    int iteration = 0;
    int swap = 0;

    for (int i = 0; i < cities.size() - 1; i++) {
        iteration++;
        boolean swapped = false;
        for (int j = 0; j < cities.size() - i - 1; j++) {
            if (cities.get(j).latitude() > cities.get(j + 1).latitude()) {
                City temp = cities.get(j);
                cities.set(j, cities.get(j + 1));
                cities.set(j + 1, temp);
                swap++;
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}
```

*Image: Calculating time complexity for our optimised bubbleSort method.*

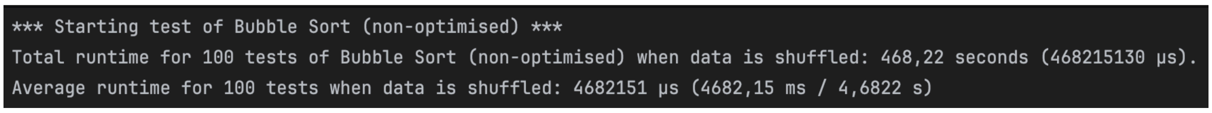
## Does the Time Complexity Change if Randomly Ordering the Dataset Before Sorting?

When testing and timing the Bubble Sort optimized algorithm on our City dataset 100 times, we obtained an average sorting time of 3,18 seconds.

```
*** Starting test of Bubble Sort (optimised) ***
Total runtime for 100 tests of Bubble Sort (optimised) when data is shuffled: 318,48 seconds (318476155 μs).
Average runtime for 100 tests when data is shuffled: 3184761 μs (3184,76 ms / 3,1848 s)
```

*Image: Result from 100 tests running optimised bubble sort on a randomly shuffled dataset*

When testing and timing the non-optimized bubble sort algorithm on our City dataset 100 times, we obtained an average sorting time of 4,68 seconds.



```
*** Starting test of Bubble Sort (non-optimised) ***
Total runtime for 100 tests of Bubble Sort (non-optimised) when data is shuffled: 468,22 seconds (468215130 μs).
Average runtime for 100 tests when data is shuffled: 4682151 μs (4682,15 ms / 4,6822 s)
```

*Image: Result from 100 tests running non-optimised bubble sort on a randomly shuffled dataset*

It is possible that a randomly shuffled list can have more or fewer swaps and therefore more or fewer iterations in an optimised bubble sort algorithm. This depends on how sorted each list is initially; a list that is more sorted will require fewer swaps and iterations. The number of swaps, iterations, and the time (time complexity) it takes to sort a randomly shuffled list changes because we don´t know its initial order. The Time complexity can change depending on how the array is sorted after randomly shuffling.

The space complexity for both bubble sort algorithms is $O(1)$. The sorting algorithms use a constant amount of space, specifically a variable to store the data (*Enjoy Algorithms*, 2023.) .

## 3.3 Further Reflections

The bubble sort is a simple sorting algorithm that can be used to sort a list of elements in ascending or descending order. It is useful for small datasets, for datasets that are nearly ordered, and for educational purposes to show the simplicity of the sorting algorithm. (Astrachan, n.d.)

# 4. Insertion Sort

## 4.1 Implementation of Insertion sort for city latitudes

### Algorithm description

Insertion sort is a simple sorting algorithm that follows the incremental method. It works quite like the way playing cards are sorted in a game of cards (Tutorials Point, n.d.).

The algorithm divides the dataset into two parts: the sorted and unsorted parts of the dataset. Often the lower/left part, a sub-list of the dataset, is maintained, which is always sorted. So when an element is to be inserted into the sorted sub-list, it has to find its appropriate place and then be inserted there. Hence the name, insertion sort (Tutorials Point, n.d.).

The steps of the insertion sort algorithm can be abstracted to the following:

1. If it is the first element, it is already sorted. Continue
2. Pick next element
3. Compare with all elements in the sorted sub-list
4. Shift all the elements in the sorted sub-list that is greater than the value to be sorted
5. Insert the value
6. Repeat until list is sorted

A simple pseudocode of insertion sort can be written as follows (Tutorials Point, n.d.)

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
   key = A[j]
   i = j - 1
   while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i -1
   A[i + 1] = key
```

## Explanation of Our Code

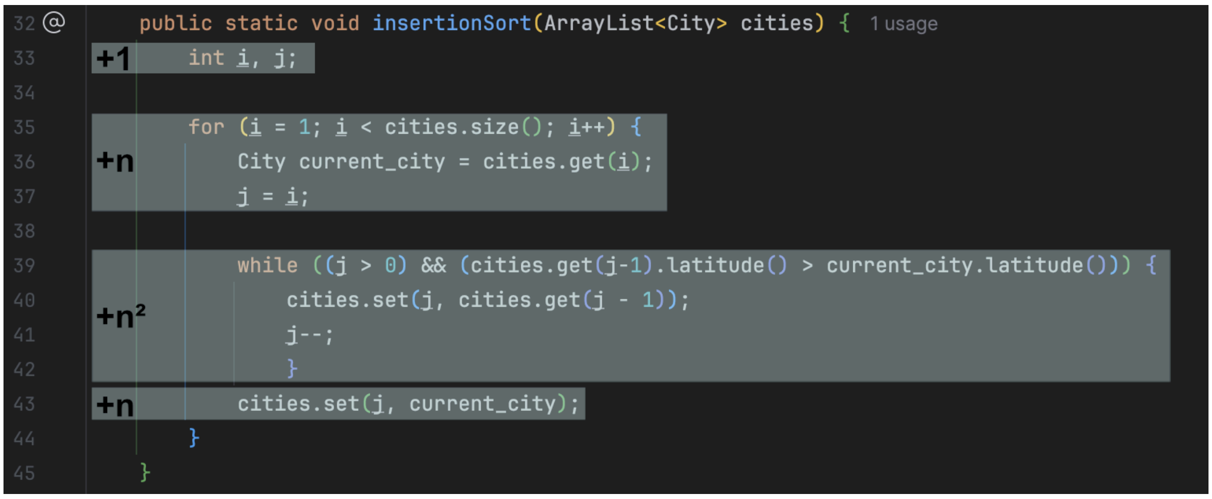
```java
public static void insertionSort(ArrayList<City> cities) {
   int i, j;

   // Starting on index 1 as index 0 is already sorted
   for (i = 1; i < cities.size(); i++) {
      // Keeps a copy of the current city element
      City current_city = cities.get(i);
      j = i;

      // Moves all elements that has a value higher than current city
one index up
      while ((j > 0) && (cities.get(j-1).latitude() >
current_city.latitude())) {
         cities.set(j, cities.get(j - 1));
         j--;
      }

      // Insert the current city element in the correct place in the
sorted sub-list
         cities.set(j, current_city);
   }
}
```

*Code: From src/main/java/task_2_insertionsort.java*

As we are working on an ArrayList with City objects, our code is a bit different than the pseudocode, but can be said to follow the same setup.

- We start on index 1 of the dataset, as we know the first element will already be sorted.
- We set the current index we are checking as the current city to insert, and then check it towards the sorted sublist we have, which is from `i-1`.
- If the current city's latitude is a lower number than the city before it, we move it one place to the right, and we repeat until the city before is no longer smaller, or we are at the start of the list.

## 4.2 Time and Space Complexity of Insertion Sort



```java
public static void insertionSort(ArrayList<City> cities) {
    int i, j;

    for (i = 1; i < cities.size(); i++) {
        City current_city = cities.get(i);
        j = i;

        while ((j > 0) && (cities.get(j-1).latitude() > current_city.latitude())) {
            cities.set(j, cities.get(j - 1));
            j--;
        }
        cities.set(j, current_city);
    }
}
```

*Image: Calculating time complexity for our insertion sort method.*

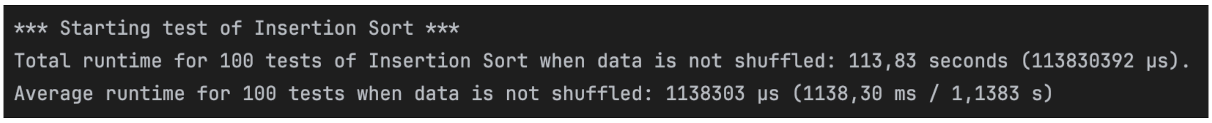
The time complexity of insertion sort depends greatly on the input data given.

If the data given to sort is sorted from the start, the while-loop that looks through the sub-list will never run, and the best possible outcome will be *2n*. This will give a best case time complexity of $\Omega(n)$ time, as it only has to run through the dataset once, and for each object it will just need to keep it in place.

If the data given is shuffled or in reverse order from the start, the time complexity will be $n^2 + 2n + 1$ , which will give a worst case time complexity of $O(n^2)$. This is because the algorithm needs to run through at least part of the sub-list for each object to find the correct place. And if the data is in reverse order, every single object needs to be checked against every single object in the sub-list.

For our City dataset, the time complexity will then be $O(n^2)$ time as it will be needed to run through the sub-list for many of the data objects.

When testing and timing the algorithm on our City dataset 100 times, we obtained an average sorting time of 1,13 seconds.



```
*** Starting test of Insertion Sort ***
Total runtime for 100 tests of Insertion Sort when data is not shuffled: 113,83 seconds (113830392 μs).
Average runtime for 100 tests when data is not shuffled: 1138303 μs (1138,30 ms / 1,1383 s)
```

*Image: Result from 100 tests running insertion sort on the City dataset*

When it comes to space complexity, this stays for the worst case at $O(1)$ space since no extra memory is needed for the algorithm (GeeksforGeeks, 2025).

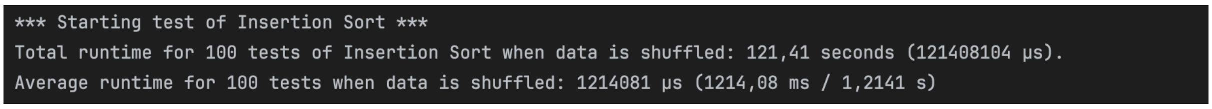
### The effect of randomly sorting the dataset before sorting

To test the difference when running insertion sort on a randomly sorted dataset, we ran the same test again on the City dataset, but we shuffled the dataset using the `.shuffle()` method found in `java.util.Collections`. Oracle (2023) explains in the `.shuffle()` JavaDoc, that it traverses the list backwards, from the last element up to the second, repeatedly swapping a randomly selected element into the "current position".

The time complexity does not change, as it will still be $O(n^2)$ time.

But we found that the actual time taken to run the algorithm does increase a tiny bit in our testing method.

In running the insertion sort on a randomly sorted (shuffled) City dataset 100 times the execution time went up to an average time of 1,23 seconds to sort the list. So even if it is not much, we found a slight increase in the time it takes when the data is shuffled randomly.



```
*** Starting test of Insertion Sort ***
Total runtime for 100 tests of Insertion Sort when data is shuffled: 121,41 seconds (121408104 μs).
Average runtime for 100 tests when data is shuffled: 1214081 μs (1214,08 ms / 1,2141 s)
```

*Image: Result from 100 tests running insertion sort on a randomly shuffled City dataset*

## 4.3 Reflections and Improvements

Insertion sort is a sorting algorithm best suited when the number of elements is small, like less than 50 elements, or if the input list is already mostly sorted, leading to very few inversions (Goodrich et al., 2014, 555), (GeeksforGeeks, 2025).

There are ways to optimize the insertion sort, and by creating a binary insertion sort, it is possible to lower the time it takes to execute the sorting, as the binary search will reduce the number of elements to compare the current element against the sorted elements in the sub-list.

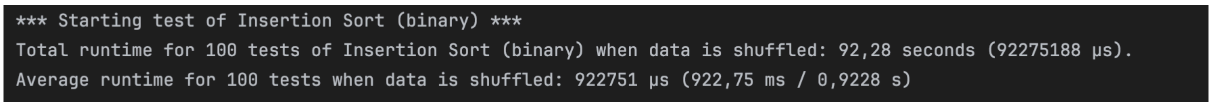
A binary insertion sort works by doing a binary search on the sub-list to find the insertion spot for the current element. The binary search works by creating a midpoint by dividing the sorted sub-list into two and checking it against the current element. If it is smaller, we search the left half, if it is larger, we search the right half until the correct position is found.

```
public static void binaryInsertionSort(ArrayList<City> cities) {
        // Starting on index 1 as index 0 is already sorted
        for (int i = 1; i < cities.size(); i++) {
            // Creates a copy of the current city element
            City current_city = cities.get(i);
            // Create a key based on the value we are sorting that we
use for the binary search
            double key = current_city.latitude();
            int low = 0;
            int high = i;

            // Execute the binary search to find the right index for
inserting the current element
            while (low < high) {
                // Use an unsigned right bit shift to safely get the
middle index between low and high
                // Avoids potential integer overflow from just using
(low + high) / 2
                int mid = (low + high) >>> 1;
                // If current latitude is larger than key, set mid value
to be new high
```

```java
                if (key < cities.get(mid).latitude()) high = mid;
                // If current latitude is smaller than key, set mid + 1 value to be new low
                else low = mid + 1;
            }
            int pos = low;
            // If pos/low is equal to i, we are already in the right place, so we continue to next iteration
            if (pos == i) continue;

            // If pos is lower than i, moves all elements with a value higher than current city one index up
            for (int j = i; j > pos; j--) {
                cities.set(j, cities.get(j - 1));
            }
            // Insert the current city element in the correct place in the sorted sub-list
            cities.set(pos, current_city);
        }
    }
```

*Code: From src/main/java/task_2_insertionsort.java*

By running the binary insertion sort, we found that reducing the number of comparisons to $O(\log n)$, we save noticeable time (from 1,21 down to 0,92 seconds) when running the algorithm through our test 100 times (Simic, 2024).



```
*** Starting test of Insertion Sort (binary) ***
Total runtime for 100 tests of Insertion Sort (binary) when data is shuffled: 92,28 seconds (92275188 µs).
Average runtime for 100 tests when data is shuffled: 922751 µs (922,75 ms / 0,9228 s)
```

*Image: Result from 100 tests running binary insertion sort on a fresh copy of the City dataset*

While the binary version of insertion sort cuts the actual time used to sort the dataset, the time complexity will still be $O(n^2)$ for the algorithm. But this does show that even if the time complexity is the same, a well-engineered algorithm can save time even on slower algorithms.

# 5. Merge Sort

## 5.1 Implementation of Merge Sort for City Latitudes

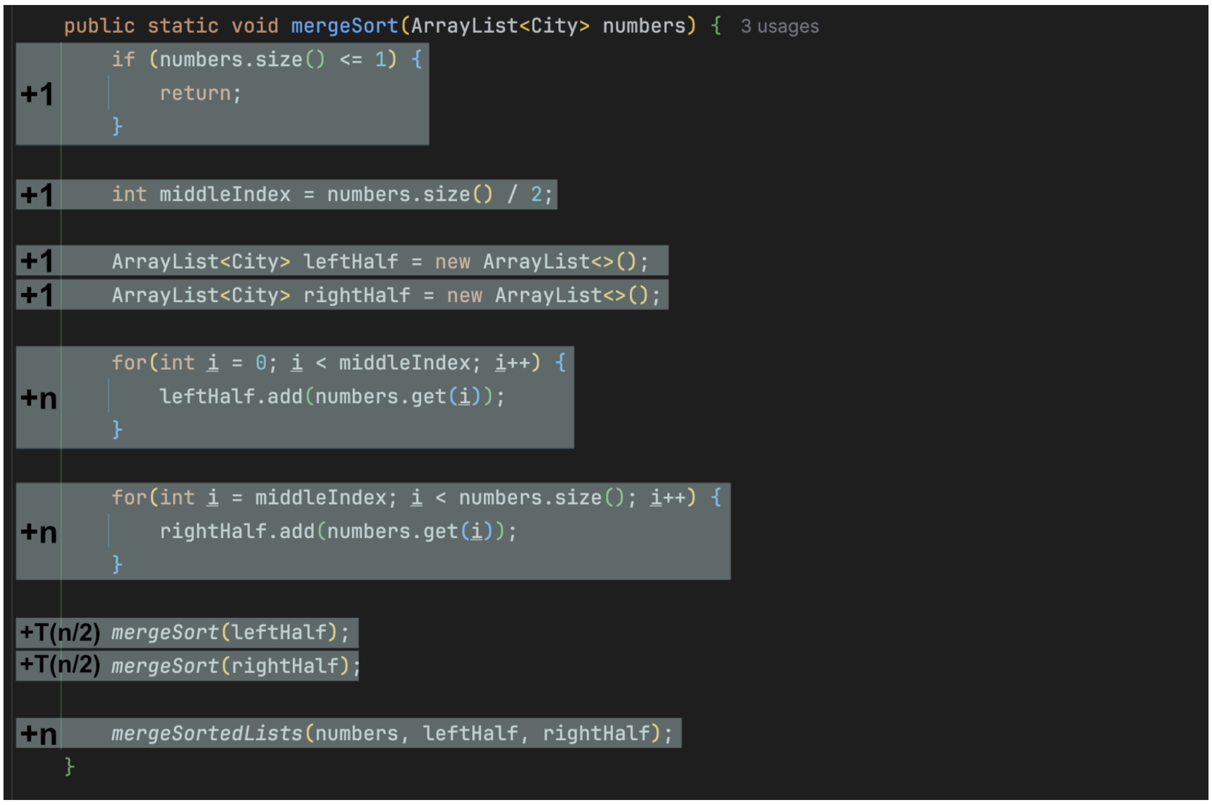
### Algorithm description

The merge sort algorithm recursively divides an unsorted array or dataset into smaller sub-arrays until each sub-array contains only one element. It then sorts and merges these sub-arrays back together in sorted order.

Merge sort is particularly efficient and stable for large datasets because it has a consistent time complexity of $O(n \log n)$ across all cases (worst, average, and best case). This predictable performance makes merge sort a reliable choice for sorting tasks (W3Schools, n.d.).

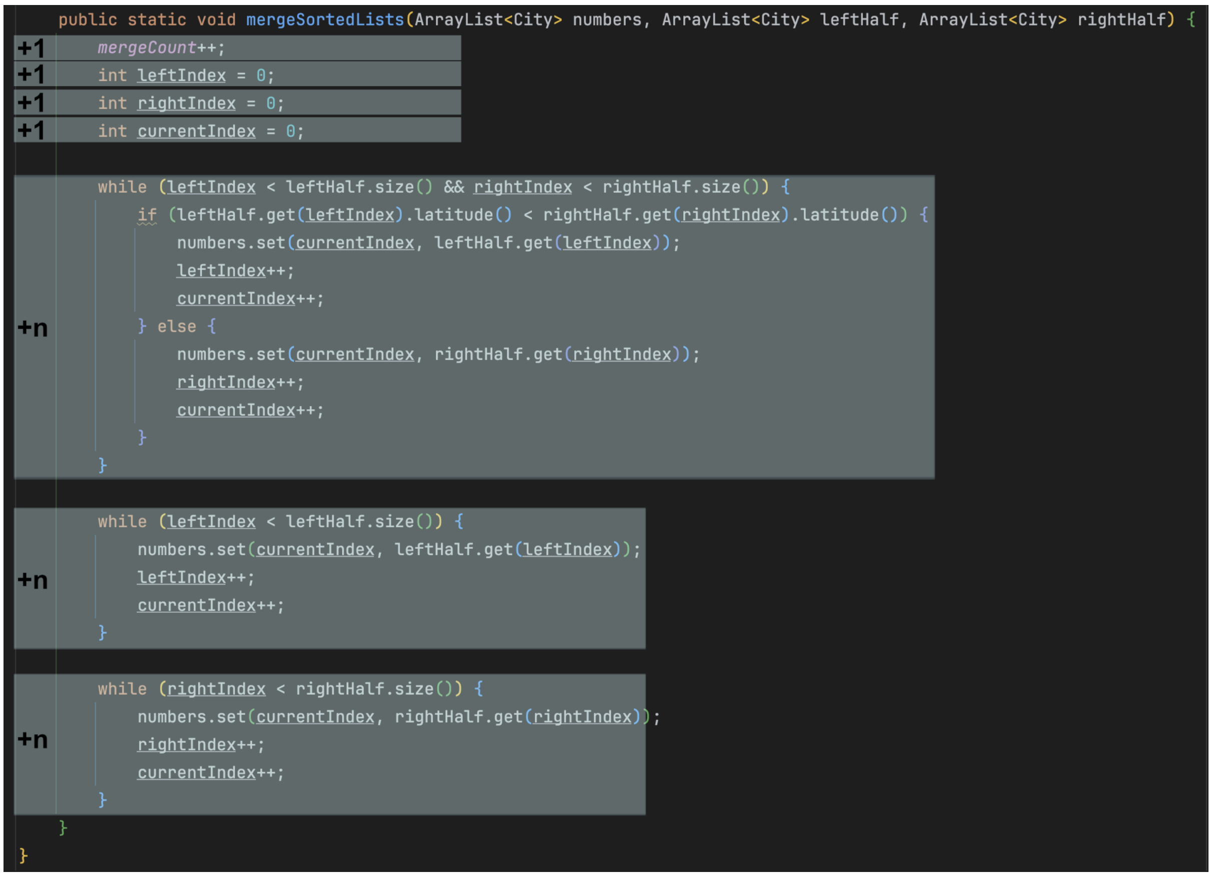
### Time and Space Complexity

The time complexity of merge sort is consistently $O(n \log n)$. This is due to the divide-and-conquer approach, where the array is repeatedly divided into halves until each sub-array contains a single element, which takes $\log n$ time. These sub-arrays are then merged back together in a sorted manner, which takes $O(n)$ time (Goodrich et al., 2014, s.539).



```
    public static void mergeSort(ArrayList<City> numbers) {  3 usages
        if (numbers.size() <= 1) {
+1          return;
        }

+1      int middleIndex = numbers.size() / 2;

+1      ArrayList<City> leftHalf = new ArrayList<>();
+1      ArrayList<City> rightHalf = new ArrayList<>();

        for(int i = 0; i < middleIndex; i++) {
+n          leftHalf.add(numbers.get(i));
        }

        for(int i = middleIndex; i < numbers.size(); i++) {
+n          rightHalf.add(numbers.get(i));
        }

+T(n/2) mergeSort(leftHalf);
+T(n/2) mergeSort(rightHalf);

+n      mergeSortedLists(numbers, leftHalf, rightHalf);
    }
```

*Image: Calculating time complexity for our mergeSort method*



```java
    public static void mergeSortedLists(ArrayList<City> numbers, ArrayList<City> leftHalf, ArrayList<City> rightHalf) {
        mergeCount++;
        int leftIndex = 0;
        int rightIndex = 0;
        int currentIndex = 0;

        while (leftIndex < leftHalf.size() && rightIndex < rightHalf.size()) {
            if (leftHalf.get(leftIndex).latitude() < rightHalf.get(rightIndex).latitude()) {
                numbers.set(currentIndex, leftHalf.get(leftIndex));
                leftIndex++;
                currentIndex++;
            } else {
                numbers.set(currentIndex, rightHalf.get(rightIndex));
                rightIndex++;
                currentIndex++;
            }
        }

        while (leftIndex < leftHalf.size()) {
            numbers.set(currentIndex, leftHalf.get(leftIndex));
            leftIndex++;
            currentIndex++;
        }

        while (rightIndex < rightHalf.size()) {
            numbers.set(currentIndex, rightHalf.get(rightIndex));
            rightIndex++;
            currentIndex++;
        }
    }
}
```

*Image: Calculating time complexity for our mergeSortedLists method.*

For space complexity, merge sort requires Θ(n) extra space for the temporary arrays used during the merge process, and Θ(log n) space for the recursion call stack (Gautam, S., n.d.). The total is then $S(n) = \Theta(n)$.

## Explanation of Our Code

```java
public static void mergeSort(ArrayList<City> numbers) {
   // Stops recursion if the list is empty or just have one element
   if (numbers.size() <= 1) {
      return;
   }
   // Creates two new sub-arrays for right and left side of the middle index
   int middleIndex = numbers.size() / 2;
   ArrayList<City> leftHalf = new ArrayList<>();
   ArrayList<City> rightHalf = new ArrayList<>();
   // Add all elements between index 0 and middle index to leftHalf sub-array
   for(int i = 0; i < middleIndex; i++) {
      leftHalf.add(numbers.get(i));
   }
   // Add all elements between from middle index to the end to the right Half sub-array
   for(int i = middleIndex; i < numbers.size(); i++) {
      rightHalf.add(numbers.get(i));
   }
   // Recursively calls mergeSort() on both halves
   mergeSort(leftHalf);
   mergeSort(rightHalf);
   // Once both halves are sorted, merge the sorted sub-arrays back
together
   mergeSortedLists(numbers, leftHalf, rightHalf);
}
```

*Code: From src/main/java/task_3_mergesort.java*

The `mergeSort()` method is responsible for recursively splitting the array of `City` objects into two halves. This step is important because MergeSort relies on breaking down the problem into smaller subproblems, which makes the sorting process more manageable and efficient.

If the array has one or zero elements, it is already sorted, and we return it as it is. For arrays with more than one element, the method calculates the middle index (`int middleIndex = numbers.size()/2`) to divide the array into two sub-arrays, `leftHalf` and `rightHalf`. It then recursively calls `mergeSort()` on both halves.

Once both halves are sorted, the method calls `mergeSortedLists(leftHalf, rightHalf)` to merge the sorted sub-arrays back together, resulting in a fully sorted array of `City` objects.

```
public static void mergeSortedLists(ArrayList<City> numbers,
ArrayList<City> leftHalf, ArrayList<City> rightHalf) {
   mergeCount++;
   int leftIndex = 0;
   int rightIndex = 0;
   int currentIndex = 0;

   // Compares elements from left and right side and adds the smallest
element to main array
   while (leftIndex < leftHalf.size() && rightIndex < rightHalf.size())
{
      if (leftHalf.get(leftIndex).latitude() <
rightHalf.get(rightIndex).latitude()) {
         numbers.set(currentIndex, leftHalf.get(leftIndex));
         leftIndex++;
         currentIndex++;
      } else {
         numbers.set(currentIndex, rightHalf.get(rightIndex));
         rightIndex++;
         currentIndex++;
      }
   }
   // Adds any remaining elements from left side
   while (leftIndex < leftHalf.size()) {
      numbers.set(currentIndex, leftHalf.get(leftIndex));
```

```
        leftIndex++;
        currentIndex++;
    }
    // Adds any remaining elements from right side
    while (rightIndex < rightHalf.size()) {
        numbers.set(currentIndex, rightHalf.get(rightIndex));
        rightIndex++;
        currentIndex++;
    }
}
```

*Code: From src/main/java/task_3_mergesort.java*

The `mergeSortedLists()` method takes the two sorted sub-arrays, *leftHalf* and *rightHalf*, and merges them into the original array (`numbers`) based on the `currentIndex` by overwriting the original values with the new sorted value. This is where the actual element comparisons happen.

The method compares elements from each sub-array based on their latitude values, inserting the smaller element into the *`numbers`* array. This process continues until all elements from one sub-array have been added. Any remaining elements from the other sub-array, which are already sorted, will then be appended directly to the *`numbers`* array. This merging step ensures that the final result remains sorted.

## 5.2 Counting the Number of Merges

### Is the Number of Merges Affected By the Order of the Dataset?

The number of merges needed to sort the dataset stays the same, regardless of the initial order of the data. By creating a global counter (`public static int mergeCount = 0;`) and incrementing it during each merge operation on the Cities data, we get a total of 47,867 merges. Even after shuffling the dataset randomly using (`Collections.shuffle(numbers);`), the number of merges remains 47,867.

This consistency happens because MergeSort always splits the array into smaller halves/sub-arrays until each piece contains a single element, then merges them back together in the same way. Since the dataset size remains unchanged, the number of merges required is the same. It is however the number of comparisons within each merge step that may vary depending on the initial order of the elements. Therefore will shuffling the list, not affect the total number of merges needed.

## 5.3 Further Reflections

This implementation of MergeSort provides a clear, efficient, and stable way to sort the `City` objects from the Cities dataset. By separating the sorting and merging logic, the code becomes easier to maintain, debug, and optimize.

Additionally, tracking the number of merges offers insight into the algorithm's behaviour, which can be useful for performance analysis or optimization. MergeSort is a great choice for large datasets and works well with different levels of computer memory, as it has reduced memory transfers, making it very efficient. It also preserves the order of equal elements, which is a helpful feature even though it was not needed for the Cities dataset.

However, MergeSort is challenging to implement in-place for arrays, and it requires additional memory allocation, which makes it less attractive if you need to save memory or if the data fits easily in the computer’s main memory.  (Goodrich et al., 2014, 562).

# 6. Quick Sort

## 6.1 Implementation of Quicksort for City Latitudes

### Algorithm Description

The quicksort algorithm sorts an array or dataset using recursion. One of the values in the array is chosen as the ‘pivot’, which is a selected element from the array. The algorithm then partitions/moves the array into two sub-arrays: elements that are lower than the pivot are moved to the left side of the pivot, and elements that are higher than the pivot are moved to the right side of the pivot. These sub-arrays are then sorted recursively, using the same process. Quicksort is known to be a fast sorting algorithm, but its performance can vary based on where the pivot is placed. It typically performs at $O(n \log n)$ in average cases and $O(n^2)$ in the worst-case. Although quicksort often outperforms merge sort in practice,  it is not stable due to element swapping during partitioning. Historically, quickSort has been the default choice for sorting arrays in C language libraries, Unix operating systems, and Java for primitive types. (Goodrich et al., 2014, 544, 555, 562), (*W3Schools*, n.d.).

### Time and Space Complexity

The time complexity of quicksort can be $O(n^2)$ in the worst-case scenario, such as when the last or first element is chosen as the pivot and the array is already sorted. As Goodrich et al. (2014, s.550) state, “... this worst-case behavior occurs for problem instances when sorting should be easy- if the sequence is already sorted”. In most other cases the time complexity is $O(n \ \log n)$ “... even if the split between L and G is not as perfect” (Goodrich et al., 2014, s.550).

For the space complexity, quicksort can be implemented in an in-place or non-in-place manner. In the in-place version of quicksort, the space complexity is $O(\log n)$ due to the use of the call stack for recursive function calls. Goodrich et al. (2014, s.555) elaborate on this by stating, “The main idea is to design a nonrecursive version of in-place quicksort using an explicit stack to iteratively process subproblems…”. This can be helpful in situations where there is limited memory available for the stack. In contrast, a non-in-place quicksort implementation may use additional containers to store elements during the sorting process, resulting in a higher space complexity.

The choice between in-place and non-in-place implementation can affect both performance and resource usage of the algorithm (Goodrich et al., 2014, 553).

### Explanation of Our Code

Here we will only explain the code for when the pivot is the first element, and we will not explain the `comparisonCount`, as this counter will be described later in the chapter.

```java
public static void quickSort(ArrayList<City> array, int low, int high) {
   // Stops recursion if the list is empty or just have one element
   if (low < high) {
   // Partition the array around the pivot, returning index of the pivot
element
      int pivotIndex = partition(array, low, high);
      // The method calls itself recursively twice for the right and
left side of the pivot index
      quickSort(array, low, pivotIndex - 1);
      quickSort(array, pivotIndex + 1, high);
   }
}
```

*Code: From src/main/java/task_4_1_quicksort.java*

The `QuickSort()` method first checks if `low` is less than `high`. If not, it means the sub-array has one or zero elements and is already sorted.

If `low` is less than `high`, the function calls the `partition()` method to partition the array around the pivot, and it returns the index of the pivot element (`pivotIndex`).

The `quickSort()` method then recursively calls itself twice. First to sort the sub-array to the left of the pivot (*`low to pivotIndex – 1`*) and second to sort the sub-array to the right of the pivot (*`pivotIndex + 1 to high`*).

```
public static int partition(ArrayList<City> array, int low, int high) {
   // Chooses the first element in current segment as the pivot element
   double pivot = array.get(low).latitude();
   // leftIndex marks the boundary of elements smaller than the pivot
   int leftIndex = low;
   // Iterates through the subarray to find elements smaller than the
pivot
   for (int rightIndex = low + 1; rightIndex <= high; rightIndex++) {
      comparisonCount++;
      // If element is smaller than the pivot, move it to the left
section
      if (array.get(rightIndex).latitude() < pivot) {
         leftIndex++;
         Collections.swap(array, leftIndex, rightIndex);
      }
   }
   // Place the pivot element in its correctly sorted position
   Collections.swap(array, low, leftIndex);
   // Return the index of the pivot where it currently is placed
   return leftIndex;
}
```

*Code: From src/main/java/task_4_1_quicksort.java*

The `partition()` method selects a pivot, which in this case is the first element of the array (or sub-array), and the element chosen is the latitude value of the city.

The `leftIndex` is initialized to the `low` index, which is the starting index of the sub-array that is being partitioned. The method then iterates through the array from `low + 1` to `high`. It checks if the current element's latitude is less than the pivot's latitude, and if it is, the `leftIndex` is incremented, and we swap the current element with the `leftIndex`.

Once all elements have been through the for-loop, we swap the pivot element with the `leftIndex`, so that it ends up at its right spot in the array.

## 6.2 Differences in Results Based on Pivot Strategy

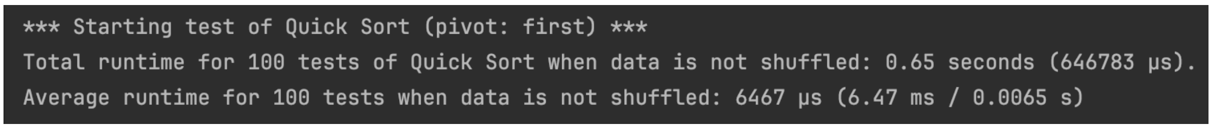
When choosing a pivot for the quicksort algorithm, it is important to know that depending on where the pivot is positioned, it can significantly affect performance. Different strategies for selecting the pivot can lead to varying results in terms of efficiency and balance of the partitions (Goodrich et al., 2014, 555).

### Choosing the First Element as Pivot Element

Choosing the first element as the pivot element is not often ideal. For nearly sorted arrays, already sorted arrays, or arrays with identical elements, this strategy can lead to worst-case behaviour. In such cases, the pivot can cause unbalanced partitions, resulting in a deep recursion tree and inefficient sorting, because the two sub-arrays vary greatly in size (Goodrich et al., 2014, 550), (*W3Schools*, n.d.).

In our research, when using the first element as the pivot on the Cities dataset, the algorithm performed very well. We initially expected the performance to most likely be poor, but since we experienced the opposite, we did some more research. By looking through our dataset, we found the first (35,6897), last (18,342), and middle element (31,309) and could clearly see that the first element is closer to the middle value than the last element.

Running our algorithm a hundred times with the pivot as the first element takes 0.65 seconds. This will prove itself to be highly efficient compared to the following tests, for when we change the pivot. The result is a more efficient partitioning and ends up as our best option for sorting the data.



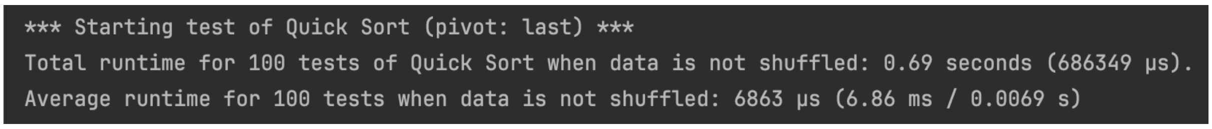
```
*** Starting test of Quick Sort (pivot: first) ***
Total runtime for 100 tests of Quick Sort when data is not shuffled: 0.65 seconds (646783 µs).
Average runtime for 100 tests when data is not shuffled: 6467 µs (6.47 ms / 0.0065 s)
```

*Image: Terminal output from TestAlgorithm.java when running quicksort 100 times with first element being the pivot element*

### Choosing the Last Element as Pivot Element

Choosing the last element as the pivot can also lead to poor performance for nearly sorted, already sorted, or identical elements. Like the first element, the last element can cause unbalanced partitions and deep recursion trees (Goodrich et al., 2014, 555).

In our research, we discovered that using the last element as a pivot leads to somewhat inefficient sorting of the Cities dataset. This is simply because the last value in the set is quite low and somewhat far away from the middle value, resulting in poor partitioning. Running our test a hundred times, with the pivot as the last element, it takes 0.69 seconds. Here we can see the slight increase in time, and would therefore not choose this pivot strategy for our Cities dataset.



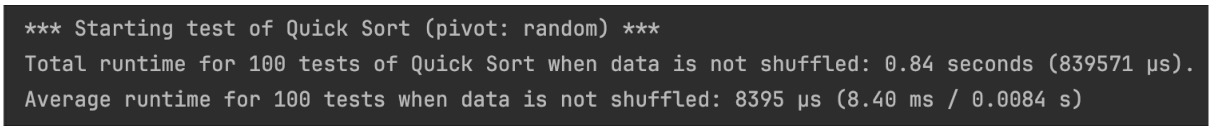
```
*** Starting test of Quick Sort (pivot: last) ***
Total runtime for 100 tests of Quick Sort when data is not shuffled: 0.69 seconds (686349 μs).
Average runtime for 100 tests when data is not shuffled: 6863 μs (6.86 ms / 0.0069 s)
```

*Image: Terminal output from TestAlgorithm.java when running quicksort 100 times with last element being the pivot element*

### Choosing a Random Element as Pivot Element

Choosing a random element as the pivot generally provides good performance. It reduces the chance of worst-case scenarios by avoiding consistently picking the smallest or largest element, leading to more balanced partitions (Goodrich et al., 2014, 555).

In our research, we observed mixed results when using a random element as the pivot. Running the test a hundred times with the pivot as the random element, it took all from 0.84 - 0.89 seconds, testing 10 times in a row. Based on this, the strategy never outperformed the first or last element as a pivot. We believe this is because these elements were already close to the middle value, making it harder for a random element to give a better partition. However, if we had no prior knowledge of the dataset's order, choosing a random element as the pivot would be a safer option.



```
*** Starting test of Quick Sort (pivot: random) ***
Total runtime for 100 tests of Quick Sort when data is not shuffled: 0.84 seconds (839571 μs).
Average runtime for 100 tests when data is not shuffled: 8395 μs (8.40 ms / 0.0084 s)
```

*Image: Terminal output from TestAlgorithm.java when running quicksort 100 times with a random pivot element*

## 6.3 Counting the Number of Comparisons

### Implementing How to Count the Comparisons

To be able to count the number of comparisons made during the sorting process, a global variable (`public static int comparisonCount = 0;`) keeps track of this. This variable is incremented each time a comparison is made between elements in the `partition()` method. As the method iterates through the dataset, it compares each element's latitude to the pivot's latitude, and each comparison is counted. After the dataset is fully sorted, the total number of comparisons is printed to the terminal, giving us the insight we need to determine the efficiency of the algorithm.

### How do Different Pivots Affect the Number of Comparisons

By reading and understanding the Cities dataset, we can tell that the data is randomized and not pre-sorted.

Using the first element as a pivot makes 873,894 comparisons. Compared to other pivot strategies (last element, random element), this one is the most efficient for this dataset because it requires fewer comparisons. Despite the general drawbacks of using the first element as the pivot, it works well here because the data is randomized and does not hold identical latitude values.

Using the last element as a pivot makes 913,159 comparisons. This is higher than when the first element is used as a pivot. This difference comes from the fact that the first latitude value is closer to the middle latitude value than the last latitude value, which makes the first element a more efficient pivot for this dataset.

Using random elements as pivots makes different comparisons like 906,591, 886,233, 979,461, 894,193, and 994,316. While some of these counts are lower than those for the last element, they are generally higher than those for the first element. Although a random pivot is often a good choice, it does not always guarantee the best performance for this specific Cities dataset.

### The Best Pivot Strategy for the World Cities Database

As we were working with time and comparisons on the different pivot strategies, we noticed some performance differences when it came to choosing the random pivot element. When counting the number of comparisons, the random element would sometimes have fewer comparisons and sometimes more comparisons than the last pivot element, but when we measured the execution time, the random element was never faster than the last element. We initially believed that the amount of time the random element took would vary depending on whether it was closer to the middle index, just like the number of comparisons varied, but this was never the case here.

The reasons for this could potentially be that some additional time is spent generating the random pivot element, which includes the overhead of random number generation and swapping elements to place the pivot correctly. Therefore, a random element could potentially have fewer comparisons than the last element but still require slightly more time.

Taking the above results into consideration, we would opt for the pivot to be the first element. We can see that the first element requires far fewer comparisons than the last element, and through many runs with the random element as pivot, it has not yet shown a comparison that is less than the first element. The first element is also far superior in time, being the fastest option. However, if we did not have access to view the original order of the data, we would opt for the random pivot element, as this gives a higher chance of a more consistent and efficient performance, no matter the order of the dataset.

## 6.4 Further Reflections

In conclusion, the quicksort algorithm is efficient and versatile on the Cities dataset. Our research revealed that using the first element as the pivot gave the best performance, even though our expectations were different. This was due to the dataset being randomized and the first element being close to the middle value, which therefore resulted in more efficient partitioning. The least efficient choice ended up being the last element as a pivot because it gave the highest number of comparisons. However, the last element required less time to be sorted than the random element, so it is important to know what your needs are when choosing a pivot strategy. A random pivot element is still a safer choice for datasets with an unknown or varying order, as it minimizes the risk of worst-case behaviour.

These findings highlight the importance of understanding the dataset’s characteristics before choosing a pivot strategy. Our results are specific to the World City dataset, so other datasets may require different pivot approaches to determine the most efficient method.

# 7. Conclusion

## 7.1 Conclusion

While testing the four sorting algorithms we were given (bubble sort, insertion sort, merge sort, and quicksort), we observed that the results were consistent with the theoretical concepts. Specifically, comparison-based sorting algorithms (Bubble and Insertion sort) would result in severely slower run times when applied to larger datasets, such as the Cities dataset we have been working with.

We ran the algorithms a hundred times on a fresh and randomly shuffled Cities dataset to ensure accurate measurements. After the first timed tests, we executed each algorithm 10 times to warm up the JVM. The results of our tests are as follows:

| **Algorithm** | **Total runtime for 100 tests** | **Average runtime** |
|---|---|---|
| Bubble sort (non-optimised) | 468,22 seconds | 4,6822 seconds |
| Bubble sort (optimised) | 318,48 seconds | 3,1848 seconds |
| Insertion sort | 121,04 seconds | 1,2104 seconds |
| Merge sort | 0,91 seconds | 0,0091 seconds |
| Quicksort (first pivot) | 0,65 seconds | 0,0065 seconds |
| Quicksort (last pivot) | 0,69 seconds | 0,0069 seconds |
| Quicksort (random pivot) | 0,84 seconds | 0,0084 seconds |

The results clearly show that the recursion-based sorting algorithms (Merge and Quicksort) are noticeably faster for large datasets such as the Cities dataset. This also aligns with the theory, where these algorithms have better worst-case time complexities compared to bubble sort and insertion sort.

In conclusion, since our Cities dataset is quite large, an algorithm like merge sort or quick sort would be a better choice due to their efficiency. This knowledge is useful for real-world applications where speed and scalability matter.

## 7.2 Limitations

During our research, we have been working with the dataset Basic World Cities, as required for the exam. While this dataset, consisting of almost 50,000 elements, is large compared to the data we have interacted with during our studies, it is still quite small compared to datasets used by larger companies. Although we can confirm that recursion-based sorting algorithms are fast for our dataset, we cannot speak for their efficacy with datasets that are tens or hundreds of times larger.

Another limitation we have not needed to worry about with our dataset, but that could affect much larger datasets, is the space complexity. Our personal laptops have no trouble handling our dataset, but when working with datasets like the paid-for version of World Cities, which consists of all cities and towns in the world (close to 4,3 million elements), we would likely encounter memory issues. This would result from the recursive sorting algorithms having higher space complexity. (SimpleMaps, 2024).

## 7.3 Personal Reflections

After spending four weeks working with these four sorting algorithms and the Basic World Cities dataset, our group has gained a much better understanding of the differences between various sorting algorithms.

While we learned in our lectures about the differences in time complexity, such as bubble sort and insertion sort, both of which use $O(n^2)$ , experiencing the results firsthand had a different impact. For instance, seeing bubble sort take 3-4 times longer to run compared to insertion sort was eye-opening. Additionally, observing the differences in time between comparison-based sorting algorithms and recursive-based algorithms provided valuable insights.

Now that we have gained more insight into different algorithms and how they work, we understand the importance of knowing the type of data we are working with to ensure we choose the best algorithm in terms of memory and time efficiency.

# Bibliography

Albano, J. (2024, January 16). Java record keyword. Baeldung.
https://www.baeldung.com/java-record-keyword

Astrachan, O. (n.d.). *Bubble Sort: An Archaeological Algorithmic Analysis*.
https://dl.acm.org/doi/pdf/10.1145/792548.611918

Azar, E., & Alebicto, M. E. (2016). Swift Data Structure and Algorithms. Packt Publishing.
https://www.google.no/books/edition/Swift_Data_Structure_and_Algorithms/xJvcDgAAQBAJ?hl=no&gbpv=1&dq=lomuto+partitioning+scheme&pg=PA119&printsec=frontcover

*DSA Merge Sort*. (n.d.). W3Schools. Retrieved April 18, 2025, from
https://www.w3schools.com/dsa/dsa_algo_mergesort.php

*DSA Quicksort*. (n.d.). W3Schools. Retrieved April 17, 2025, from
https://www.w3schools.com/dsa/dsa_algo_quicksort.php

GeeksforGeeks. (2024, March 14). *Time and space complexity analysis of merge sort*.
https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/

GeeksforGeeks. (2024, April 19). Why should you choose Java for DSA? Retrieved from
https://www.geeksforgeeks.org/why-should-you-choose-java-for-dsa/

GeeksforGeeks. (2025, March 22). Insertion sort algorithm.
https://www.geeksforgeeks.org/insertion-sort-algorithm/

GeeksforGeeks. (2025, April 23). Bubble sort algorithm.
https://www.geeksforgeeks.org/bubble-sort-algorithm/

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Java* (6th ed.). John Wiley & Sons.

IBM. (2024, February 12). *JIT compiler (Java Just-In-Time compiler)*. IBM Documentation.
https://www.ibm.com/docs/en/sdk-java-technology/8?topic=reference-jit-compiler

Gautam, S. (n.d.). Bubble sort, Selection sort and Insert-sort. EnjoyAlgorithms. Retrieved April 24, 2025, from
https://www.enjoyalgorithms.com/blog/introduction-to-sorting-bubble-sort-selection-sort-and-insertion-sort

Gautam, S. (n.d.). Merge sort algorithm. EnjoyAlgorithms. Retrieved April 17, 2025, from
https://www.enjoyalgorithms.com/blog/merge-sort-algorithm

Oracle. (2023). Collections.shuffle (Java Platform SE 21). Oracle.
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collections.html#shuffle(java.util.List,java.util.random.RandomGenerator)

Pseudo Editor. (n.d.). Merge Sort in Pseudocode. Writing a Merge Sort in Pseudocode. Retrieved March, 2025, from https://pseudoeditor.com/guides/merge-sort

Rosetta Stone. (2025, April 13). Sorting algorithms/Bubble sort. Rosetta Code. https://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort

Simic, M. (2024, March 18). Binary insertion sort. Baeldung. https://www.baeldung.com/cs/binary-insertion-sort

SimpleMaps. (2024, March 19). World cities database. https://simplemaps.com/data/world-cities

Simplilearn. (n.d.). *Bubble Sort Algorithm: Understand and Implement Efficiently*. Simplilearn.Com. Retrieved April 22, 2025, from https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm

Tutorials Point. (n.d.). Insertion sort algorithm. https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm

W3Schools. (n.d.). Bubble sort algorithm. W3Schools. Retrieved April 22, 2025, from https://www.w3schools.com/dsa/dsa_timecomplexity_bblsort.php

W3Schools. (n.d.). Insertion sort algorithm. Retrieved April 18, 2025, from https://www.w3schools.com/dsa/dsa_algo_insertionsort.php