candidate number: 12, 40, 172

PG4200 Exam 2025

Introduction and Project Structure

When given the task of sorting all city latitudes in an ordered list, we first solved the task with assign each given latitude with the given city, but decided to go away from this project solution, and only show the algorithms sorting algorithm possibilities and only show latitudes in an ordered list, as specified in the task.

This project is organized into multiple Java files. Each sorting algorithm is isolated in its own class, while shared functionality is grouped under a Utilities package. This makes it easier to read and understand our project, but also makes it more efficient as we don't have to duplicate the same codes in each of the tasks.

CSVReader.java: Reads the unique latitudes values from worldcities.csv
FileUtil.java: Handles the sorted latitude data to CSV files.
ListUtil.java: Provides a helper method for copying and shuffling the lists.
TimerUtil.java: Measures the execution time of code blocks in nano seconds.
SortResult.java: Represents the result of sorting, storing time and comparison counts.

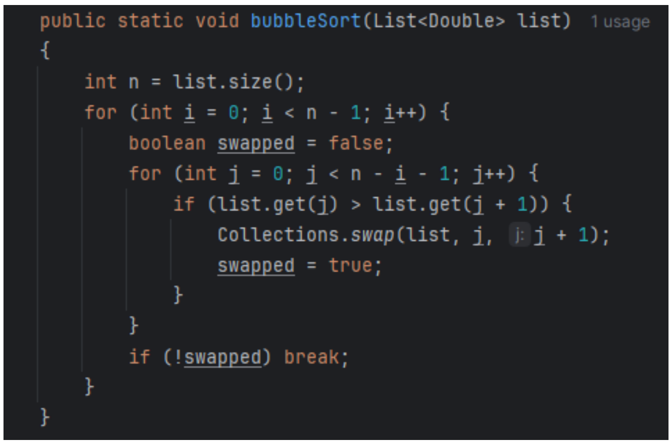
Task 1: Bubble Sort

The bubble sort algorithm goes through each of the list in our given “worldcities.csv” and compares adjacent elements and swaps them if they are in the wrong order. This process is then repeated until there is no longer need for any swaps, which gives the indication that the list is then finished sorting.

The outer loop goes through the entire list, while the inner loop compares adjacent pairs and swaps them only if it is needed. An optimization is then included to break early if no swaps occur in a pass.This is easier explained with pseudocode, We have included an example below:

Optimized Sort

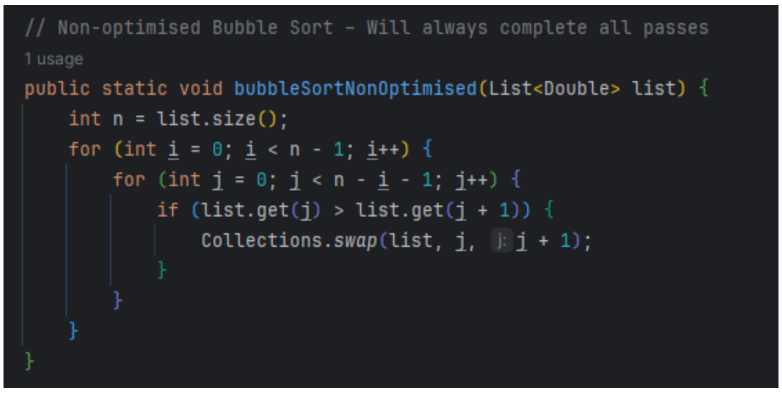
```
SET swapped TO TRUE                      //Start by assuming a swap is needed
WHILE swapped IS TRUE:                   //Continue if swap occurred in the last pass

    SET swapped TO FALSE                 //Reset swapped for the current pass
    FOR i FROM 0 TO length of list - 2:  //Iterate through the list up to the second
                                           last element

        IF list[i] > list[i+1]:          //Compare the adjacent elements
        SWAP list[i] AND list[i+1]       //Swap if left element is larger than right
        SET swapped TO TRUE              //Set swapped to true because a swap
                                           occoured
```



```java
public static void bubbleSort(List<Double> list)
{
    int n = list.size();
    for (int i = 0; i < n - 1; i++) {
        boolean swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (list.get(j) > list.get(j + 1)) {
                Collections.swap(list, j, j + 1);
                swapped = true;
            }
        }
        if (!swapped) break;
    }
}
```

Non-Optimized sort

```
FOR i FROM 0 TO length of list - 2:          // Outer loop: runs n-1 times
   FOR j FROM 0 TO length of list - i - 2:   // Inner loop: compares adjacent
                                                elements
      IF list[j] > list[j+1]:                // Compare left and right element
         SWAP list[j] AND list[j+1]          // Swap if left is greater than right
```



```java
// Non-optimised Bubble Sort - Will always complete all passes
1 usage
public static void bubbleSortNonOptimised(List<Double> list) {
    int n = list.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (list.get(j) > list.get(j + 1)) {
                Collections.swap(list, j, j: j + 1);
            }
        }
    }
}
```

Before giving the result of the given Algorithm, it is useful to know that it is based on a notation called *Big O notation* “Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Big O is a member of a family of notations invented by German mathematicians Paul Bachmann, Edmund Landau, and others, collectively called Bachmann-Landau notation or asymptotic notation.” (Wikipedia contributors, 2025) to shorten this reference up a bit, the Big O notation is then describing the complexity of the written code, by using algebraic terms.

b)

The given time complexity of the Bubble Sort Algorithm varies based on the initial order of the given dataset (Codecademy, 24. Apr, 2025). In the best case, when the list is already sorted, the algorithm performs n-1 comparisons, which have no swaps. Due to the use of a swapped flag, the algorithm then detects that no swaps have occurred and terminates/exits early, giving a best-case time complexity of ***O(n).***

On the other hand, for the worst case, the list is randomly ordered or is reverse sorted, the algorithm performs ***n(n-1)/2*** comparisons and many swaps across multiple passes. This results in a worst-case time complexity of ***O(n^2).***

The sorting time changes when the list is randomly ordered because Bubble Sort needs to make significantly more comparisons and swaps. When the list is already sorted, the early termination mechanism reduces the number of operations. However, when the list is shuffled, the algorithm must fully process the dataset, which leads to an increased time complexity.

In this project, the dataset was randomly shuffled before sorting to simulate the worst case, which increased the sorting time, which is what we expected.

Task 2: Insertion Sort

Although an Insertion sort is not the best optimized Algorithm for sorting, it has its perks: “An insertion sort is less complex and efficient than a merge sort, but more efficient than a bubble sort”(BBC Bitesize, 2025, paras. 1-2).

The insertion sort algorithm starts by creating a sorted list for one element at a time. It proceeds to take each element from the unsorted part of the list and then insert it into the correct position within the sorted part from the left side.

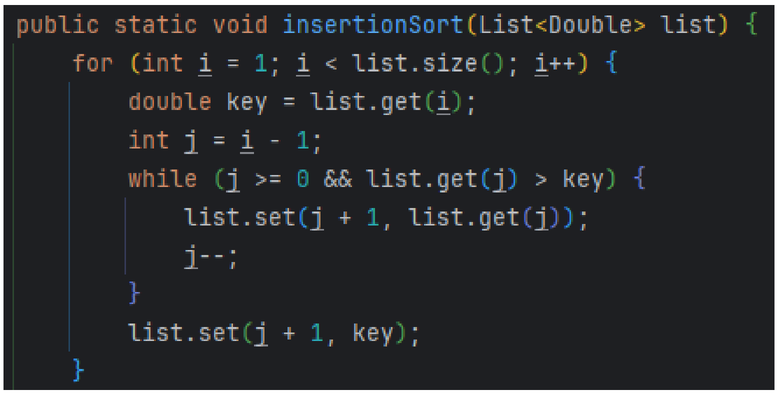
The outer loop goes through each element in the list (starting from the second element), while the inner loop compares the current element with the elements in the sorted part of the list and then shifts the largest elements one position to the right to make space for the current element.

“If implemented well, the running time of insertion-sort is O(n + m), where m is the number of inversions (that is, the number of pairs of elements out of order). Thus, insertion-sort is an excellent algorithm for sorting small sequences, because insertion-sort is simple to program, and small sequences necessarily have few inversions. (Goodrich & Tamassia, 2014, p. 713/714).

This can be explained with pseudocode like this:

```
FOR i FROM 1 TO length of list - 1:          // Start from the second element
SET key TO list[i]                           // Select the element to insert

   SET j TO i - 1                            // Start comparing with previous elements
```

```
    WHILE j >= 0 AND list[j] > key:          // Shift elements larger than key to the right
        SET list[j + 1] TO list[j]
        DECREMENT j BY 1

    SET list[j + 1] TO key                   // Insert the key at the correct position
```



```java
public static void insertionSort(List<Double> list) {
    for (int i = 1; i < list.size(); i++) {
        double key = list.get(i);
        int j = i - 1;
        while (j >= 0 && list.get(j) > key) {
            list.set(j + 1, list.get(j));
            j--;
        }
        list.set(j + 1, key);
    }
```

b)

As done in Task 1 (Bubble Sort), the time complexity of Insertion Sort depends on the dataset's initial order. In the best case (already sorted), the algorithm performs n-1 comparisons with no shifts, resulting in O(n) time complexity. In the worst case (random or reverse order), each element is compared and shifted across the sorted parts, requiring n(n-1)/2 operations, leading to $O(n^2)$ time complexity.

The time it takes to sort depends on how the list is ordered. When the list is already sorted, Insertion Sort finishes quickly because it has very little work to do. But when the list is shuffled, the algorithm needs to make many more comparisons and shifts, which takes more time. In this project, we shuffled the list on purpose to create the worst-case scenario and measured how long the sorting took.

Task 3: Merge Sort

The merge sort algorithm works by dividing the list into halves, which will sort each half recursively, and then merging the sorted halves back together. This process will continue until the entire list is sorted. Merge Sort uses a divide-and-conquer strategy that ensures efficient sorting even when sorting larger datasets (GeeksforGeeks 29 Jan, 2025).

More in-depth described: “Merge-sort is based on an algorithmic design pattern called divide-and-conquer. The divide-and-conquer pattern consists of the following three steps:

1. Divide: If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.

2. Recur: Recursively solve the subproblems associated with the subsets.

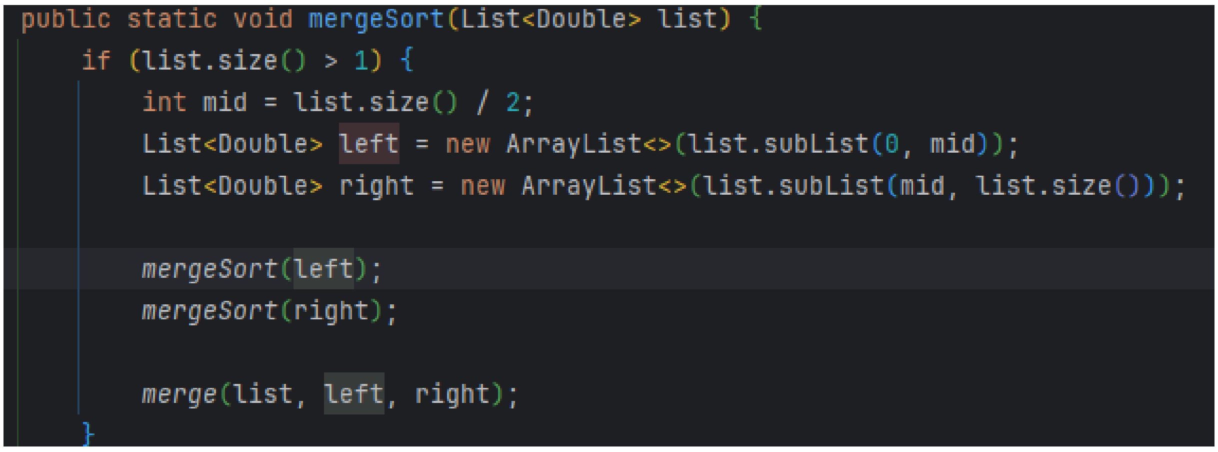
3. Conquer: Take the solutions to the subproblems and "merge" them into a solution to the original problem. (Goodrich & Tamassia, 2014, p. 678).”

The algorithm splits the list repeatedly until each sublist contains only one element, which is considered sorted. It then merges these sublists by comparing the smallest elements of each sublist and building a new sorted list.

This can be explained with pseudocode like this:

Splitting part:

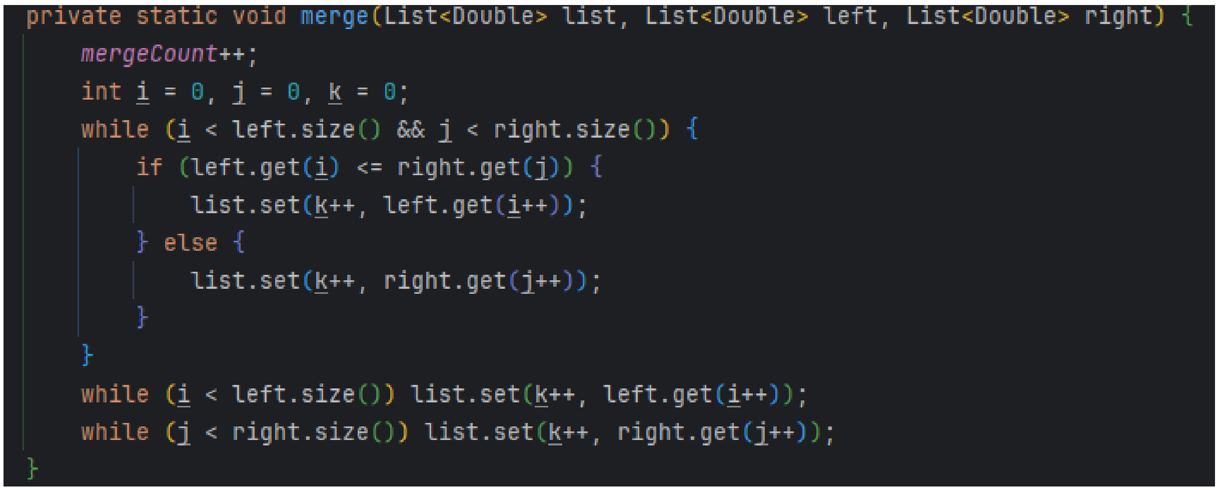
```
IF length of list > 1:                         // Check if the list has more than one
                                                  element
DIVIDE list INTO left half AND right half      // Split the list into two halves
CALL mergeSort ON left half                    // Recursively sort the left half
CALL mergeSort ON right half                   // Recursively sort the right half
MERGE the sorted halves back to list  // Merge the two sorted halves together
```



```java
public static void mergeSort(List<Double> list) {
    if (list.size() > 1) {
        int mid = list.size() / 2;
        List<Double> left = new ArrayList<>(list.subList(0, mid));
        List<Double> right = new ArrayList<>(list.subList(mid, list.size()));

        mergeSort(left);
        mergeSort(right);

        merge(list, left, right);
    }
```

Merging part:

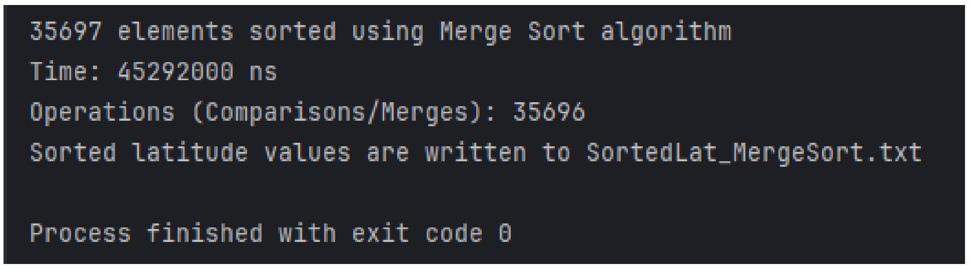
```
START with indexes i, j, k at 0                    // i for left list, j for right list, k for main list

WHILE there are elements left in both left and right lists:
        IF left[i] is smaller than or equal to right[j]:
        PUT left[i] into main list at position k
        INCREMENT i

   ELSE:
        PUT right[j] into main list at position k
        INCREMENT j
        INCREMENT k

COPY any remaining elements from left list into main list
COPY any remaining elements from right list into main list
```



```java
private static void merge(List<Double> list, List<Double> left, List<Double> right) {
    mergeCount++;
    int i = 0, j = 0, k = 0;
    while (i < left.size() && j < right.size()) {
        if (left.get(i) <= right.get(j)) {
            list.set(k++, left.get(i++));
        } else {
            list.set(k++, right.get(j++));
        }
    }
    while (i < left.size()) list.set(k++, left.get(i++));
    while (j < right.size()) list.set(k++, right.get(j++));
}
```

b)

The number of merges in Merge Sort does not change, even if the list is randomly ordered. This is because Merge Sort always splits the list into halves in the same way, no matter what values the list contains. In our experiment, Merge Sort consistently performed 35,696 operations to sort 35,697 elements..

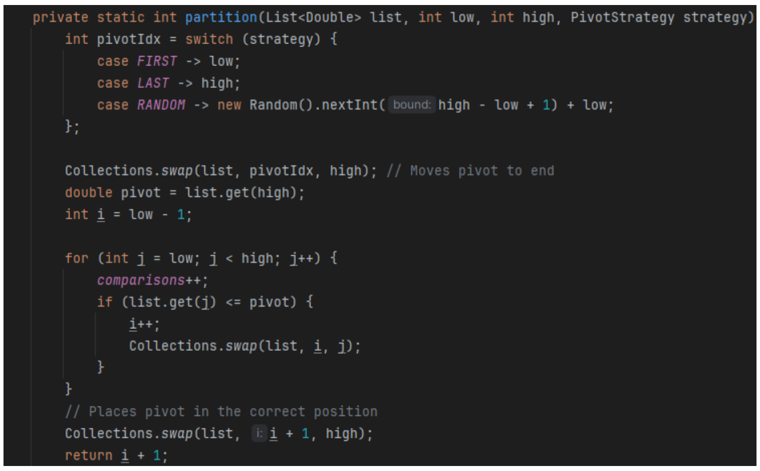


```
35697 elements sorted using Merge Sort algorithm
Time: 45292000 ns
Operations (Comparisons/Merges): 35696
Sorted latitude values are written to SortedLat_MergeSort.txt

Process finished with exit code 0
```

Task 4: QuickSort

The QuickSort algorithm works by selecting a pivot element and partitioning the list into two sides. Elements smaller than the pivot are moved to the left, while all elements larger than the pivot are moved to the right. This process is repeated recursively on both sides until the entire list is sorted. “Quick-sort is an excellent choice as a general-purpose, in-memory sorting algorithm.” (Goodrich & Tamassia, 2014, p. 678).

We implemented the QuickSort algorithm using three pivot selection strategies: FIRST, LAST, and RANDOM. This can be explained with pseudocode like this:



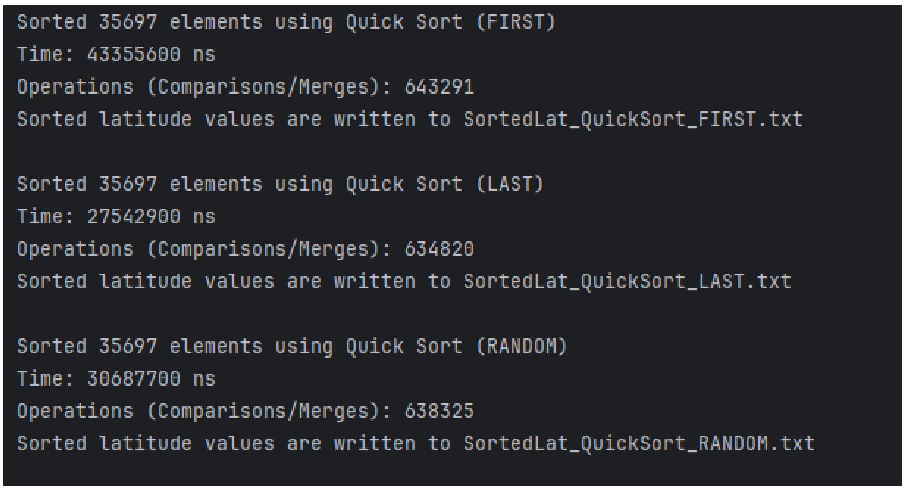
```java
private static int partition(List<Double> list, int low, int high, PivotStrategy strategy)
    int pivotIdx = switch (strategy) {
        case FIRST -> low;
        case LAST -> high;
        case RANDOM -> new Random().nextInt( bound: high - low + 1) + low;
    };

    Collections.swap(list, pivotIdx, high); // Moves pivot to end
    double pivot = list.get(high);
    int i = low - 1;

    for (int j = low; j < high; j++) {
        comparisons++;
        if (list.get(j) <= pivot) {
            i++;
            Collections.swap(list, i, j);
        }
    }
    // Places pivot in the correct position
    Collections.swap(list,  i: i + 1, high);
    return i + 1;
```

```
int pivotIdx = switch (strategy) {
case FIRST -> low;                          // choose first element
case LAST -> high;                          // choose last element
case RANDOM -> new Random().nextInt(high - low + 1) + low;
// choose random element

Collections.swap(list, pivotIdx, high);  // Moves pivot to end
double pivot = list.get(high);            // Gets value
int i = low - 1;

for (int j = low; j < high; j++) {
comparisons++;                          // count comparisons
if (list.get(j) <= pivot) {
```

```
i++;
Collections.swap(list, i, j);             // swaps if the element is smaller than pivot


Collections.swap(list, i + 1, high);      // Places the pivot in correct position
```

The table below shows how many comparisons were made and how long it took for each pivot strategy.

Strategy	Comparisons	Time (ns)
FIRST	643 291	43 355 600
LAST	634 820	27 542 900
RANDOM	638 325	30 687 700



```
Sorted 35697 elements using Quick Sort (FIRST)
Time: 43355600 ns
Operations (Comparisons/Merges): 643291
Sorted latitude values are written to SortedLat_QuickSort_FIRST.txt

Sorted 35697 elements using Quick Sort (LAST)
Time: 27542900 ns
Operations (Comparisons/Merges): 634820
Sorted latitude values are written to SortedLat_QuickSort_LAST.txt

Sorted 35697 elements using Quick Sort (RANDOM)
Time: 30687700 ns
Operations (Comparisons/Merges): 638325
Sorted latitude values are written to SortedLat_QuickSort_RANDOM.txt
```

The result shows that the number of comparisons and time taken change depending on the pivot selection strategy.

- The FIRST pivot had the highest number of comparisons and the longest execution time.
- The LAST pivot performed the best, with the fewest comparisons and the shortest sorting time.
- The RANDOM pivot had the second highest number of comparisons and used the second most time.

This shows that the choice of pivot has an impact on the efficiency of QuickSort. For this dataset, the LAST pivot strategy performed the best, with the fewest comparisons and the shortest time.

References:

Codecademy. (18. Apr, 2025). *Time complexity of Bubble Sort explained with examples.* Retrieved 01. april. 2025, from https://www.codecademy.com/article/time-complexity-of-bubble-sort

GeeksforGeeks. (07 Feb, 2024). *Time and space complexity of Insertion Sort*. Retrieved 07 April, 2025, from https://www.geeksforgeeks.org/time-and-space-complexity-of-insertion-sort-algorithm/

BBC Bitesize. (2025). *Insertion sort*. BBC Bitesize. Retrieved 07 April 2025. https://www.bbc.co.uk/bitesize/guides/zjdkw6f/revision/6

GeeksforGeeks**.** (29. Jan, 2025). *Merge Sort – Data Structure and Algorithms Tutorials,* Retrieved April 25, 2025, from https://www.geeksforgeeks.org/merge-sort/

GeeksforGeeks. (14 Mar, 2024). *Time and Space Complexity Analysis of Merge Sort*. Retrieved 20 April, 2025, from https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-merge-sort/

Goodrich, M. T. (2014). *Data structures and algorithms in Java* (6th ed.). Singapore: Wiley.

Wikipedia contributors. (2025, April 20). *Big O notation*. In *Wikipedia, The Free Encyclopedia*. Retrieved April 21, 2025, from https://en.wikipedia.org/wiki/Big_O_notation

Goodrich, M.T., & Tamassia, R. (2014). *Data structures and algorithms in Java** (4th ed.). John Wiley & Sons. Retrieved April 21, 2025.