# Report for a project on Computation Thinking with Algorithms

By Mihass Konopelko

Table of Contents

[Report for a project on Computation Thinking with Algorithms 1](#_Toc189736574)

[Introduction 2](#_Toc189736575)

[Sorting Algorithms 3](#_Toc189736576)

[Bubble Sort 3](#_Toc189736577)

[Selection Sort 4](#_Toc189736578)

[Insertion Sort 5](#_Toc189736579)

[Merge Sort 6](#_Toc189736580)

[Counting Sort 7](#_Toc189736581)

[Implementation & Benchmarking 8](#_Toc189736582)

[Implementation 8](#_Toc189736583)

[Benchmarking 11](#_Toc189736584)

[Bibliography 13](#_Toc189736585)

## Introduction

The concept of sorting is an essential part of data and information processing, often being responsible for 35% of the total time taken when processing information (Zhu Zhi-gang, 2020). As such, implementing efficient sorting methods has led to the creation of multiple sorting algorithms and frameworks to measure these algorithms' qualities.

When discussing the algorithms, there is a list of factors which affect the overall efficiency of the sorting algorithms:

* Time Complexity
  + The measurement of the time taken for the sorting algorithm to execute is split into best-case (**Ω)**, average-case (**Θ**), and worst-case (O) notations. Most often the Big-O notation is used when determining the efficiency of the algorithm.
* Space Complexity
  + Measures “the amount of memory an algorithm takes to execute completely” (Shiksha 2024).
* Stability
  + Stability describes the algorithms which preserve the order of identical elements during the sort.
* Comparison-based
  + The sorting algorithms that compare two elements during the sort are referred to as comparison-based sorting algorithms.
* Adaptability
  + “A sorting algorithm is adaptive if it sorts sequences that are close to sorted faster than random sequences, where the distance is determined by some measure of presortedness.” (Peterson and Moffat, 1995). In other words, adaptive algorithms sort partially pre-sorted data faster than completely unsorted data.
* In-place sorting
  + The algorithm that interacts with the input directly is called an in-place sorting algorithm.

These factors will be referred to when discussing the bubble, selection, insertion, merge and counting sort algorithms in the next section.

## Sorting Algorithms

This section will describe five algorithms used in this project, and discuss the parameters like time complexity followed by space complexity, stability and other relevant properties while presenting the logic of each algorithm with step-by-step diagrams of the sorting procedure.

### Bubble Sort

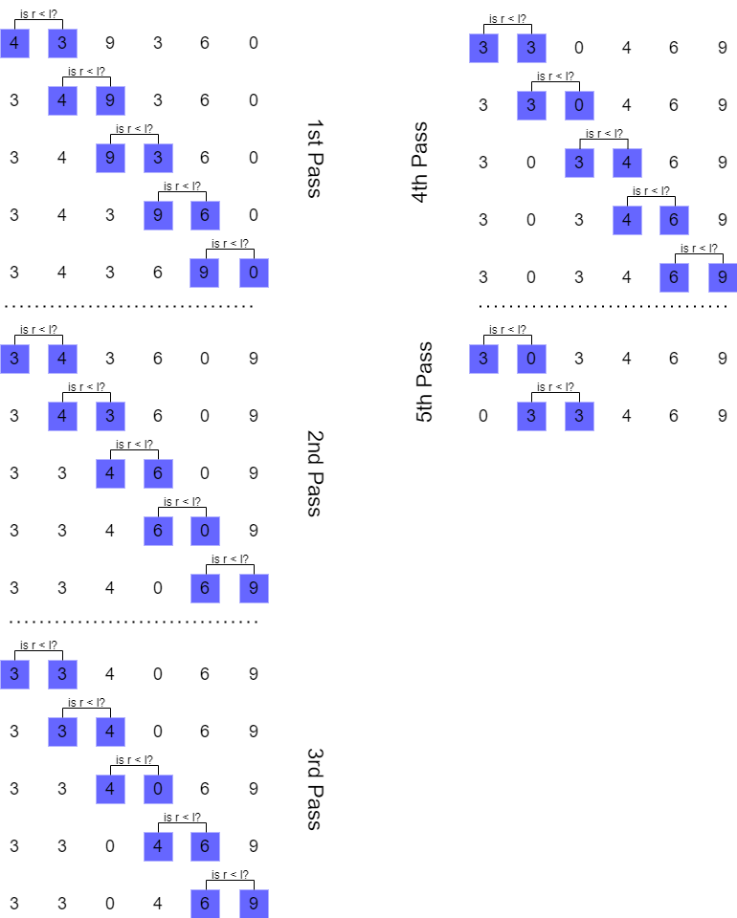
This comparison-based algorithm works by iterating through an array of numbers, swapping two neighbouring elements to sort them. The algorithm repeats the process until there are no swap events during iteration, thus implying that the array is sorted.

Figure 1: A diagram of bubble sort operation using my student id. 5th pass was truncated slightly.

The diagram to the right demonstrates the working of the bubble sort using my ID. To sort the array, the algorithm required five passes, with the last three passes being focused on relocating 0 to the beginning of the array.

Its best-case performance is *O(n)*, with average and worst performance being *O(n2)*.

Since *n* is the size of the input, we can use the array to the right to exemplify it further:

* The best case of *O(n)* is possible if an array is already sorted or has a few elements which are 1 swap away from being sorted. In the case with the demo array, if 0 was closer to the beginning of an array then the sort would have leaned toward a best-case scenario.
* Average/Worst case of *O(n2)* would be 36 comparisons. It took 25 comparison operations to sort the array, and most of the array was sorted on the 8th comparison, with the remaining checks being redundantly used to move 0 to the start of the array.

As such, while bubble sort is straightforward to implement and visualize, it is rather inefficient in both performing the sorting and in being visualized in the diagram in Figure 1.

When it comes to other properties, the bubble sort algorithm does not alter the relative order of identical values and thus is stable. Moreover, it doesn’t need additional memory as it can perform sorting in place.

When it comes to adaptiveness, the bubble-sort algorithm is considered one – after all, if in the example array, the 0 was closer to the beginning of an array, the sorting would have required a few passes less.

### Selection Sort

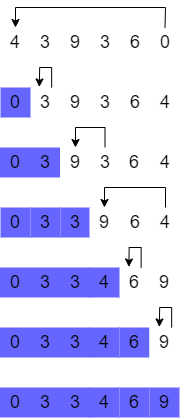
The selection sort is a comparison-based algorithm that operates by zoning an array into “sorted” and “unsorted” sectors, then iterating through the unsorted part of the array in search of the element with the smallest value. When iteration is complete, the element next to the sorted zone is swapped with the element with the smallest value. That element is then marked as sorted. The described procedure is repeated until the array only consists of the sorted sector.

Figure 2: A compact diagram of selection sort operation using my student id.

The diagram to the right demonstrates the selection sort in action. To sort the array, it required six passes, with the total number of operations being (or 6+5+4+3+2+1, where each number equates to the number of operations per pass).

In total, 6 passes required 21 comparison operations to sort the example array. Considering the design of the algorithm, the time complexity in any case will be *O(n2)*:

* In a best-case scenario, where the array was already sorted, it would still require six passes to affirm that the array is sorted.
* In a worst-case scenario, the algorithm will take 6 passes to sort it.

As such, this algorithm will always perform at *O(n2)*.

Since the algorithm modifies the input array directly, it is an in-place algorithm with a space complexity of 1. This algorithm can move elements across the whole array and is likely to disrupt the order of elements with the same value, thus making this algorithm unstable.

Lastly, this algorithm would not have benefited from any presorted data, thus making it not adaptable.

### Insertion Sort

The insertion sort algorithm works by separating the array into sorted and unsorted zones. The first element of the array is immediately assigned to the sorted zone, with the following iterations taking the first element of the unsorted zone and sorting it into the sorted part of the array via comparison.

Figure 3: Diagram of the Insertion Sort Algorithm using my student id.

The diagram to the right demonstrates the sorted part of the array in blue, and the element to be sorted in red. The red element is compared to the elements within the sorted zone until it meets the following criteria:

*The sorted element value (@ index - 1) is less or equal to the red unsorted element (@ index).*

To sort the array to the right, it took five passes, with 12 comparison operations.

Its best-case performance is *O(n)*, with average and worst performance being *O(n2)*:

* In the best-case scenario of *O(n)*, where we assume that the array is already sorted, it will take 6 comparison operations to verify it.
* In average-case and worst-case scenarios of *O(n2)*, where the array is unsorted, it will require constant iterating through most of the sorted section to insert the element into the correct spot.

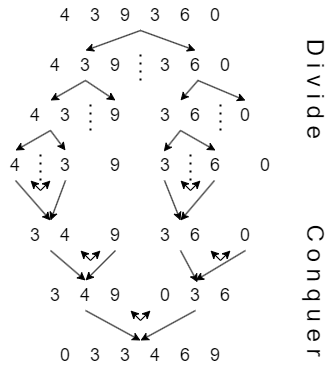
As with the previous algorithms, insertion sort alters the input array to perform sorting. As such, it is an in-place algorithm with a space complexity of 1.

Due to the ordered, one-by-one movement of elements from the unsorted area into the appropriate location within the sorted area, the order of elements with the same values would remain the same, thus making this algorithm stable.

Furthermore, this algorithm would have benefited from any presorted data, thus making it adaptable.

### Merge Sort

Figure 4: A diagram of the merge sort algorithm in action using my id as a sample array.

This comparison-based sorting algorithm operates by continuously separating an input array into smaller subsets until these subsets consist of a single element, then merging these subsets (and sorting as well) until the array is reconstructed back in a sorted state.

In the example array to the right, it is demonstrated how the array was, at first, split into two parts, then split more and more until there were 6 arrays of one element each. This stage is often referred to as “Divide”.

Once the array was dissected into small chunks, it was put back together. Through the design of the algorithm, the separated segments were re-joined back in reverse. While joining in small pieces, the elements were sorted as well. This stage is referred to as “Conquer”.

Its best-case, average-case and worst-case performance is *O(nlog(n))*, and is considered one of the best performances possible time-wise. However, this algorithm is not very effective space-wise, being *O(n)*, because it consumes memory during the creation of subsets and chunks of the initial input array (thus, it is not an in-place sorting algorithm).

Although the algorithm separates the array and then re-joins it, this does not affect the sorting order of the value elements, making it stable. Furthermore, the algorithm is not adaptive as it will always perform the same procedures regardless if data is sorted or not.

### Counting Sort

This is a non-comparative sorting algorithm which works by counting the number of times each element has occurred within the input array, recording it into an ordered counting array, and then using that array as a map to reconstruct a sorted output array.

Before describing this algorithm further, it is important to highlight two characteristics:

* This algorithm initially only works with positive whole numbers.
* To do counting, it is desirable to know the largest value contained in the input array.

Using an example diagram, the following steps can be identified in the operation of this algorithm:

1. Counting – the algorithm traverses the sample array once, incrementing the value in the counting array where the index of a counting array equals the traversed element’s value.
2. The counting array is then translated into the frequency map.
3. Then, a sample array is iterated through in reverse order, and the element’s value is used as an index in the frequency map.
   1. The value stored at that index is decreased by 1, then used as a location index for where to sort the value from the sample array into the output array.

When it comes to time complexity, this algorithm is *O(n+k)* in all cases, where *n* is the number of inputs and *k* is the range of inputs. Since the last stage of the algorithm iterates through the input array in reverse order, the order of elements with the same values is not disturbed, making it a stable algorithm. Considering the design of the algorithm, the space complexity is also *O(n+k)* as there is a need for extra space for the counting array, which is *k* size, and output array of *n* size (what also makes this algorithm not in place).

Figure 5: The demonstration of the counting algorithm using my student id as an example.

Furthermore, it is not an adaptable algorithm – the algorithm is not comparison-based and thus does not benefit from partially sorted data.

## Implementation & Benchmarking

The application used for this report was developed using IntelliJ IDEA 2023.3.3 IDE. There are 2 classes, one for all implemented algorithms (Algorithms.java), and the main class where they are running and being benchmarked (Main.java).

When implementing the first three algorithms, I relied on code demonstrated in the week 9 slides provided by Dominic Carr. The remaining two algorithms, Merge Sort and Counting Sort, required me to look online for potential pseudocode sources which I referenced as comments in the source code.

The first part of application development focused on implementing all requirements for data presentation and benchmarking of the first algorithm (bubble sort).

### Implementation

#### Test Arrays

In the implementation of the benchmarking, there was an important consideration to be made regarding test arrays - there was an option to either:

* generate a set of new data before each algorithm benchmark, or
* clone a set of data from data which was generated once at the start of the application
  + or even have a file with generated array data, which could be loaded into the application and be a sort of “constant”.

While generating new data before each benchmark was not a difficult option to implement, it would introduce random unfairness (which would be minuscule to non-existent, but still possible).

Generating an array set and saving/loading it into the application appeared to be too excessive for the scope of this application since the data generated was quick and straightforward.

As such, the approach for data was to generate a set of arrays on application start-up. This set of arrays would be then cloned each time to be used with algorithms.

The logic for the creation of data can be found in Main.java in methods:

* *private static int[][] generateUnsortedArrays()*
* *private static int[] arrayCreator (int length)*

The initialisation of the data:



#### Benchmarking and Measurement

The process of measurement was straightforward: set a timer, start a timer just before using an algorithm, execute the algorithm, and then stop the timer.

Difficulty was:

* to ensure no unnecessary operations took place during the measurement.
* to get an average time for 10 executions.
* to manage and store measurement data.

The figure below demonstrates the benchmark implementation, as well as measurement management.

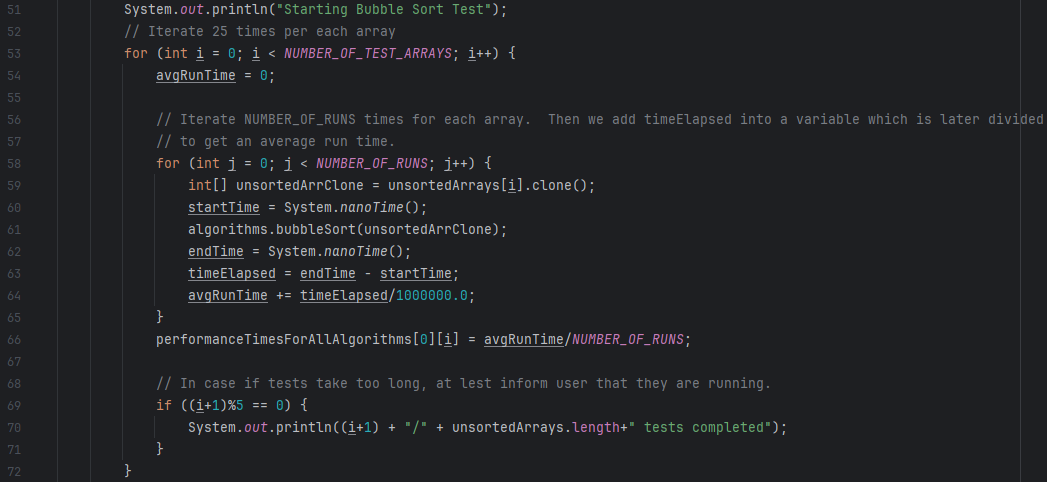


Figure 6: The segment of benchmarking code for the bubble-sort algorithm. A similar code was used for the remaining algorithms.

Before the beginning of the benchmark, the test array is cloned at line 59. That array can be altered without affecting the original *unsortedArrays* because *.clone()* performs a deep copy.

Lines 60 and 62 measure the time it took for an algorithm to execute. Algorithms only contained the sorting logic and excluded print methods or other unnecessary code which could affect the measurement.

On line 64, the calculated timeElapsed is added to the value of avgRunTime. The idea was to store the cumulative time taken to run an algorithm 10 times on the same array, then get an average time by dividing it by 10 in line 66 and storing it in a 2D array where:

* the first dimension corresponded to the algorithm
* the second dimension corresponded to the array index

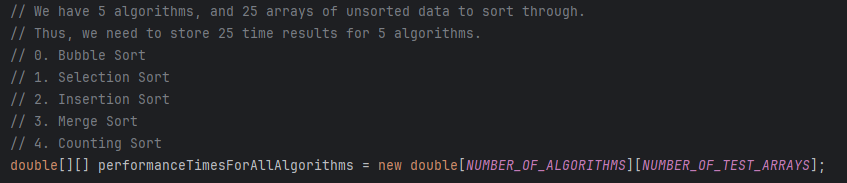


Figure 7: Storing measurements this way allowed it later to be displayed more simply in the console.

#### Algorithms

All of the algorithms and helper methods were implemented in the Algoritm.java, for the separation of logic and ease of reading/evaluation.

A description of what each algorithm does was written in the previous section. The inclusion of pictures of code for each implemented algorithm is redundant, so instead only one algorithm, the counting sort algorithm, will be demonstrated there to show comments and layout expected in other algorithms.

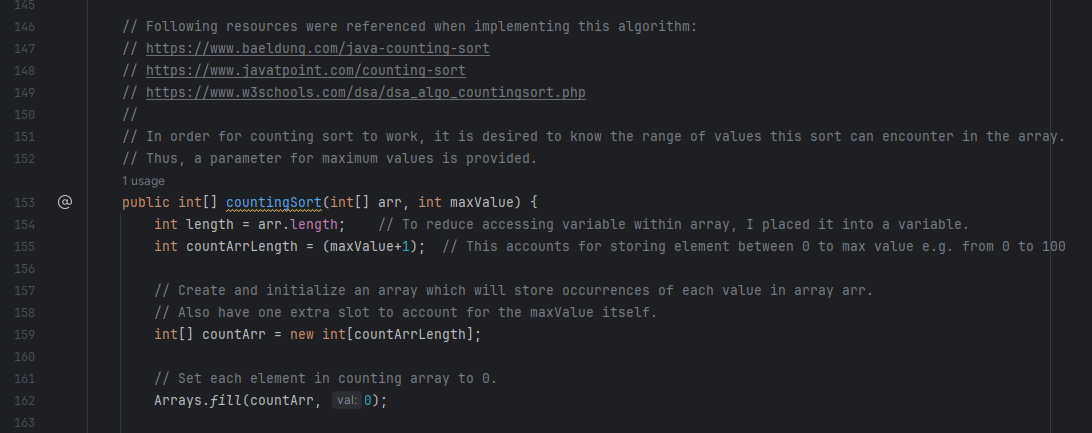


Figure 8: The first half of the counting sort algorithm method.

Lines 146-149 include online resources used to reference the code and pseudocode related to the algorithm. The goal was to not only comprehend the workings of the algorithm but also write it nicely and clearly as if for other students to read.

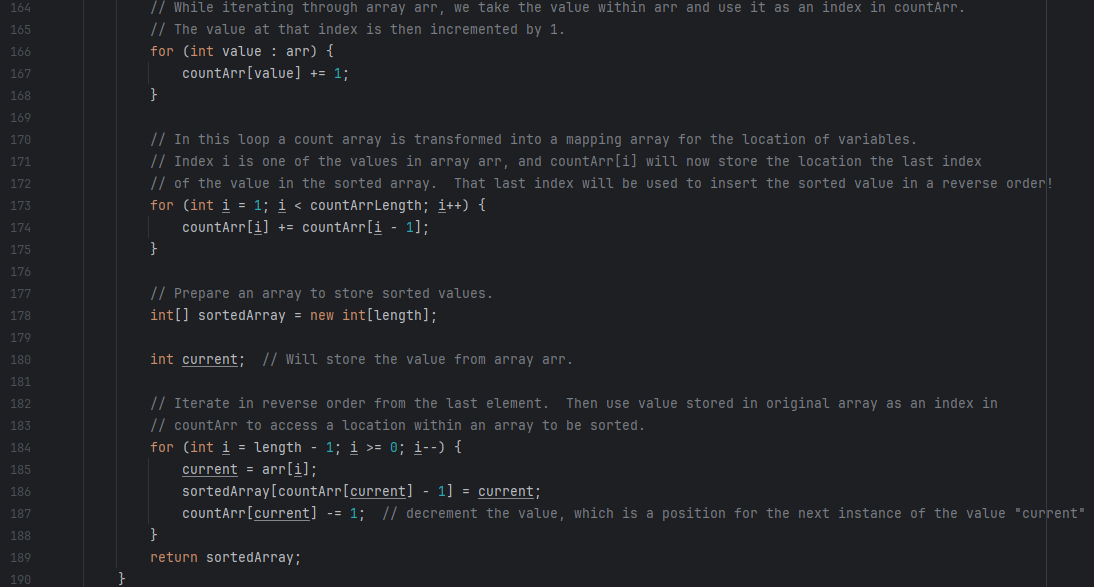


Figure 9: The last half of the counting sort algorithm method.

### Benchmarking

#### Graphs and Results

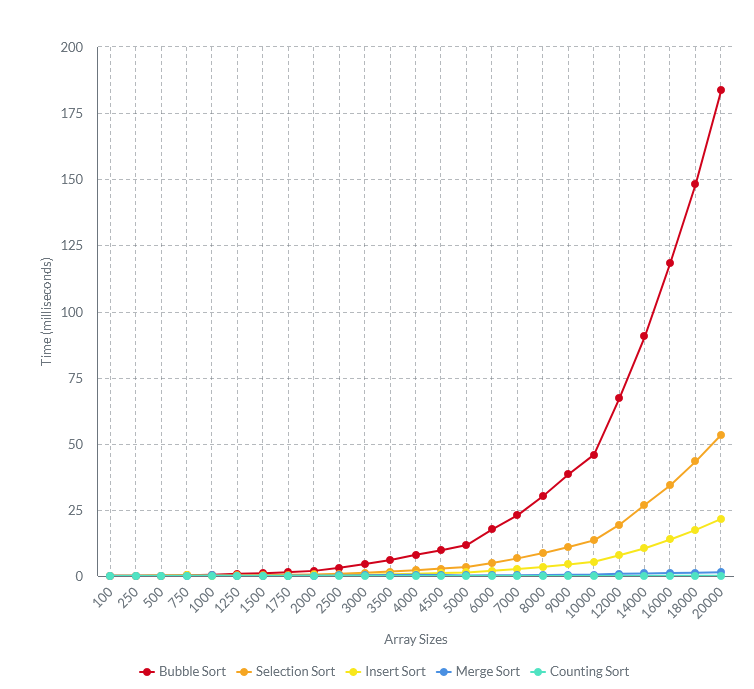


Figure 10: The complete graph of the 5 algorithm performances. Created using pictochart.com

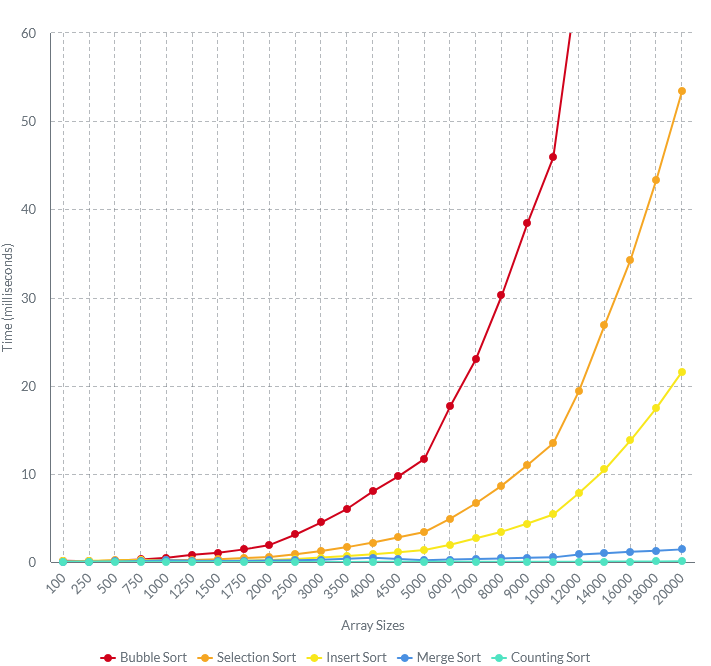


Figure 11: Slightly adjusted line chart to demonstrate the performance of merge and counting sorting algorithms.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Size** | **Bubble Sort** | **Selection Sort** | **Insert Sort** | **Merge Sort** | **Counting Sort** |
| *100* | 0.132 | 0.061 | 0.061 | 0.054 | 0.007 |
| *250* | 0.096 | 0.087 | 0.139 | 0.05 | 0.011 |
| *500* | 0.163 | 0.179 | 0.161 | 0.068 | 0.019 |
| *750* | 0.287 | 0.207 | 0.195 | 0.109 | 0.027 |
| *1000* | 0.47 | 0.197 | 0.162 | 0.204 | 0.035 |
| *1250* | 0.817 | 0.236 | 0.088 | 0.133 | 0.039 |
| *1500* | 1.047 | 0.325 | 0.127 | 0.122 | 0.021 |
| *1750* | 1.432 | 0.442 | 0.17 | 0.141 | 0.009 |
| *2000* | 1.938 | 0.568 | 0.216 | 0.174 | 0.008 |
| *2500* | 3.122 | 0.883 | 0.347 | 0.212 | 0.01 |
| *3000* | 4.495 | 1.25 | 0.502 | 0.258 | 0.014 |
| *3500* | 6.035 | 1.689 | 0.668 | 0.376 | 0.013 |
| *4000* | 8.012 | 2.207 | 0.878 | 0.476 | 0.015 |
| *4500* | 9.74 | 2.802 | 1.101 | 0.336 | 0.015 |
| *5000* | 11.693 | 3.402 | 1.37 | 0.209 | 0.018 |
| *6000* | 17.658 | 4.913 | 1.96 | 0.272 | 0.023 |
| *7000* | 23.03 | 6.664 | 2.69 | 0.327 | 0.024 |
| *8000* | 30.227 | 8.667 | 3.441 | 0.4 | 0.027 |
| *9000* | 38.438 | 10.995 | 4.366 | 0.495 | 0.031 |
| *10000* | 45.884 | 13.467 | 5.389 | 0.525 | 0.034 |
| *12000* | 67.222 | 19.403 | 7.83 | 0.854 | 0.041 |
| *14000* | 90.641 | 26.914 | 10.492 | 0.987 | 0.047 |
| *16000* | 118.42 | 34.287 | 13.802 | 1.162 | 0.053 |
| *18000* | 148.29 | 43.338 | 17.446 | 1.267 | 0.071 |
| *20000* | 183.639 | 53.411 | 21.594 | 1.432 | 0.074 |

Figure 12: Due to the amount of data, the results were transposed. Time is in milliseconds.

#### Discussion

The results met the expectations, as that bubble sort was the slowest sorting algorithm, and that counting sort would have been the fastest.

There were two things which this report helped to highlight:

* How efficient/inefficient the selected 5 algorithms are, visually and in terms of data.
  + Counting sort is the greatest example here – it takes less than a tenth of a millisecond to sort 20,000 element arrays
  + While for a bubble sort, it takes 183 milliseconds on average to do so.
  + In short, counting sort is around 2481 times faster than bubble sort.
* That algorithms with similar described time complexity of O(n2) show the same trend in performance, but each algorithm had its point of becoming slow.
  + Which is falsely accentuated by the change in size difference of array size in the diagram but is still present.

Also, the data demonstrates three separate anomalies, explanations for which were not discovered:

* at array size 250 for bubble sort and merge sort, there is an improvement in sorting speed.
* merge sort demonstrates occasional speed-ups, which can be noted in the diagram at 4500 and 5000 elements,
* counting sort demonstrates erratic behaviour, where it sometimes gets faster at sorting
  + sorted 1500 elements faster than 1250, but at the same time slower than 5000 elements.
  + It is inconsistent, which should not be the case as it cannot be affected by data, meaning that either the implementation is incorrect or buggy, or there is some other characteristic that was missed in this project.

# Bibliography

Zhu Zhi-gang, 2020, Analysis and Research of Sorting Algorithm in Data Structure Based on C, Journal of Physics Conference Series, <https://www.researchgate.net/publication/341875554_Analysis_and_Research_of_Sorting_Algorithm_in_Data_Structure_Based_on_C_Language>

Shiksha, 2024, Time and Space Complexity of Sorting Algorithms, <https://www.shiksha.com/online-courses/articles/time-and-space-complexity-of-sorting-algorithms-blogId-152755#2>

Ola Peterson and Alistair Moffat, 1995, A framework for adaptive sorting, Discrete Applied Mathematics Volume 59 Issue 2, <https://www.sciencedirect.com/science/article/pii/0166218X93E0160Z>