Computational Thinking:

Sorting Algorithms Project

Developer:	Michael Clarke
Student No:	G00411256
Tutor:	Dr Dominic Carr
Module:	Computational Thinking with Algorithms
University:	Atlantic Technological University

Date: 14 May 2023

Table of Content

Introduction

This report explores the topic of sorting algorithms in computer science and presents the development and findings of the benchmarking application I developed for this project.

Section 1 introduces the concepts sorting and sorting algorithms.

Section 2 provides a high level overview of each algorithm used in the benchmarking application and presents bespoke diagrams that explain their execution process.

Section 3 discusses the implementation of the benchmarking tool and explores the results.

Finally, the conclusion summarises the findings and lists recommendations.

1 Concept of Sorting and Sorting Algorithms

This section introduces the concept and importance of sorting algorithms. Additionally, it discusses complexity, performance, and types of sorting.

1.1 Sorting Algorithms

In general, sorting algorithms provide a way to rearrange a given list of elements (Sorting Algorithms 2023). Sorting algorithms are a key theme in computer science (Carr, 2023b; Healy, 2022), and Skiena (2020 p.109) identifies three important reasons to study these algorithms:

- Sorting is the foundation that many other established algorithms are based upon.
- The design of sorting algorithms introduce important computer science concepts such as divide-and-conquer, data structures and randomised algorithms.
- Sorting problems are among most studied concepts in computer science.

Essentially, Skiena (2020) argues that by understanding sorting algorithms, a programmer can gain 'powerful' problem solving techniques that they can apply to other computer science problems. Examples of these problems are searching, determining element uniqueness and element selection.

1.2 Space and Time Complexity

Space complexity play an important role in evaluating sorting algorithms because the concepts help programmer to understand how the algorithm uses memory. Additionally, time complexity provides insight into how the logic of a sorting algorithm, particularly, how it cycles through comparisons using loop structures, impacts the total time it takes to execute an input of increasing size (Carr 2023a; Healy 2022; Skiena 2020; Time Complexity of Sorting Algorithms n.d.).

Different sorting algorithms can behave in very different ways as their input dataset gets larger. By way of example, the logic of the *counting sort* algorithms takes less time to process large datasets compared to the logic of *selection sort*. One factor that makes *selection sort* slower is its logic, which contains a nested *for loop* (Carr 2023b). Essentially, in a worst case scenario *selection sort*'s time complexity is O(n^2) brought about by the nested loops (Healy 2022; Skiena 2020). In comparison, *counting sort's* time complexity is O(n + k) (k being the range of the input elements), because it's logic uses a linear set of *for loops* to process data (Counting Sort Algorithm n.d.). Time complexity considerations reveal how *counting sort* uses less time compared to *selection sort*.

Space complexity considerations add another dimension to the analysis. In most cases, time complexity considerations win out over space complexity issues (Understanding Space Complexity 2021). From the example above, *counting sort's* space complexity is O(max)/O(k) making it essentially linear, whereby *selection sort*'s space complexity is constant, being O(1) (Comparison of Sorting Algorithms (2019); Counting Sort Algorithm n.d.; Selection Sort Algorithm n.d.). Even though *selection sort*'s memory consumption is much owner than *counting sort's,* the latter wins out because of the way it handles time complexity issues. However, in situations where memory could be limited, such as embedded systems, then *selection sort*'s memory consumption could become an issue (Sorting Algorithms Explained, 2019).

1.3 Performance

Where space and time complexity can be thought of as theoretical measure of algorithmic design, performance is the measure of an algorithm's efficiency during practical application: that is to say, its performance when ran on computer hardware (Carr 2023a; Skiena 2020). Taking in space and time complexity, performance also considers additional factors such as the programming language, the operating system, the complier and the hardware.

Although programmer's can assess algorithms to high level of accuracy from a pure theory domain (Skiena 2020 p.32), using benchmarking tools that measure performance, such as the one I designed for this project, add depth to algorithmic studies.

1.4 Types of Sorting Algorithms

Sorting algorithms can be categorised based on the following features (Sorting Algorithms Explained, 2019; Stable Sorting Algorithms, 2022).

1.4.1 Swaps, Comparisons, Recursion

Many sorting algorithms work on the basis that they swap elements, such as is the case for bubble sort and selection sort. The number of swaps becomes a feature of these algorithms (Sorting Algorithms Explained, 2019). Additionally, the number of comparisons, and whether the function uses recursion also has an impact on complexity. In particular, how much time it takes to iterate though the comparisons or through the recursive calls.

1.4.2 In-place or Out-of-Place

In-place sorting algorithms sort their input without creating a new list, thus reducing the amount of memory required during run time (Sorting Algorithms Explained, 2019). For example, insertion sort does not need to create a new array as it sorts elements by moving them around a pivot point. In comparison, merge sort creates new memory to store its sorted elements. (Sorting Algorithms Explained, 2019).

1.4.3 Stable or Unstable

Stable sorting algorithms maintain the positions of two equal elements relative to each other: unstable sorting algorithms ignore relative ordering (Stable Sorting Algorithms, 2022). Unstable sorting algorithms use the entirety of a given element as a sort-key, making equal elements indistinguishable from each other. In comparison, stable sorts use the unique attribute of equal elements to create relative positioning. Bubble sort, Insertion sort, counting sort and merge sort are example of stable sorts. Selection sort is of the unstable kind (Stable Sorting Algorithms, 2022).

1.4.4 Comparison-Based and Non-Comparison-Based

Comparison-based sorting algorithms compare element to sort them. In an alternative approach, non-comparison sort uses unique information about keys and operations to determine the sorted order of elements (Comparison of Sorting Algorithms, 2019). Most classic sorting algorithms compare elements, but here are exceptions such as counting sort, which uses arithmetic operations to sort elements (Counting Sort in Java, 2022). See section 2.4 for more information.

1.4.5 Hybrid

Hybrid algorithms take a *best of both worlds* approach by combining two or more algorithms designed to solve the same problem (Carr 2023c). These algorithms have ways to assess the input data and then choose one of their sub-algorithms to achieve the best sorting results. For example, Intro-sort is a hybrid of quicksort and heapsort. During execution, if the quicksort recursion exceeds log n level, then intro-sort's monitoring calculations will switch to heap sort to optimise the algorithm's performance (Carr 2023c).

2. Sorting Algorithms

This section provides a high level overview of each algorithm used in the benchmarking application and presents bespoke diagrams that explain their execution process. The input data for each diagram is 4, 1, 1, 5, 6. All Big O notation considers the **worst case performance only** for each algorithm listed in this section.

2.1 Bubble Sort

Bubble sort is an in-place, comparison-based, stable sort, element swapping algorithm (Bubble Sort in Java, 2023). This algorithm is good for sorting small sets of data, but it does not scale well due to its time complexity. Essentially, it is more useful as a teaching aid and is not used in practice (Carr, 2023c). Bubble sort has a quadratic time complexity or O(n^2), and a constant memory complexity or O(1). (Bubble Sort in Java, 2023).

Figure 2.1 present the execution process of bubble sort. This algorithm contains a do-while loop with a nested for-loop. The for-loop iterate though each input element and uses an if-statement to check and swap elements that are greater than one another.

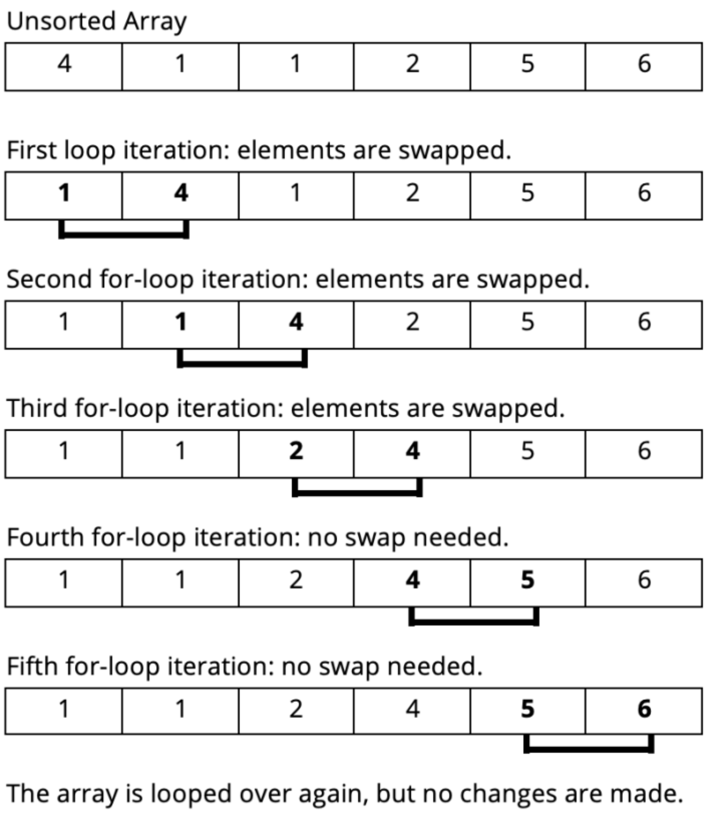

Unsorted Array
4 1 1 2 5 6
First loop iteration: elements are swapped.
1 4 1 2 5 6
Second for-loop iteration: elements are swapped.
1 1 4 2 5 6
Third for-loop iteration: elements are swapped.
1 1 2 4 5 6
Fourth for-loop iteration: no swap needed.
1 1 2 4 5 6
Fifth for-loop iteration: no swap needed.
1 1 2 4 5 6
The array is looped over again, but no changes are made.

Figure 2.1 Bubble Sort. Figure adapted from Bubble Sort in Java (2023)

2.2 Selection Sort

Selection sort is an in-place, comparison-based, unstable sort, element swapping algorithm. Like bubble sort, this algorithm is good for sorting small sets of data, or partially sorted data, but it does not scale well due to its time complexity. (Selection Sort in Java, 2022; Stable Sorting Algorithms, 2022). Unlike bubble sort, selection sort requires **fewer** swaps to sort element (Sorting Algorithms Explained, 2019). Selection sort has a quadratic time complexity; O(n^2); and a constant memory complexity; O(1) (Selection Sort in Java, 2022).

Figure 2.2 present the execution process of selection sort. This algorithm contains a for-loop with a nested for-loop. The nested for-loop iterates though each element of the input data and swaps smaller elements with larger elements.

Unsorted Sub-list	Sorted Sub-list	Least Element
(4, 1, 1, 2, 5, 6)	()	1
(4, 1, 2, 5, 6)	(1)	1
(4, 2, 5, 6)	(1, 1)	2
(4, 5, 6)	(1, 1, 2)	4
(5, 6)	(1, 1, 2, 4)	5
(6)	(1, 1, 2, 4, 5)	6
()	(1, 1, 2, 4, 5, 6)	

Figure 2.2: Selection Sort: Figure adapted from Selection Sort (n.d.)

2.3 Insertion Sort

Insertion sort is an in-place, comparison-based, stable sort algorithm (Insertion Sort in Java, 2023). Like bubble sort and selection sort, insertion sort is useful for sorting small sets of data, or partially sorted data, but it does not scale well due to its time complexity. Insertion sort has a quadratic time complexity; O(n^2); and a constant memory complexity; O(1) (Insertion Sort in Java, 2023).

Figure 2.3 present the execution process of insertion sort. The sorting process is similar to how card players sort their hand of playing cards (Insertion Sort in Java, 2023). We imagine elements moving from right to left. On the right, there are the unsorted elements. Elements are then moved one by one to the left and their values are sorted against the values of the elements on the left as they increase. The process continues until the elements on the right are depleted.

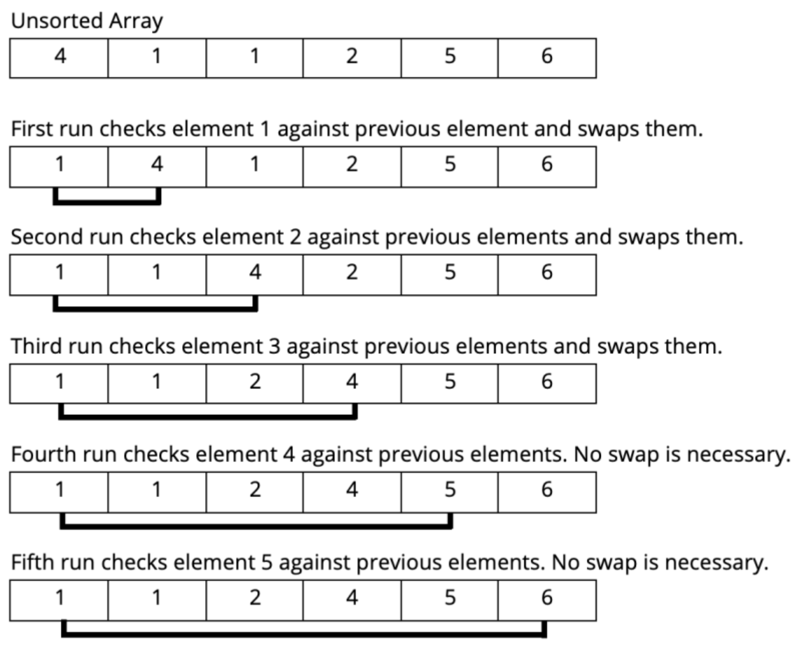

Unsorted Array
4 1 1 2 5 6
First run checks element 1 against previous element and swaps them.
1 4 1 2 5 6
Second run checks element 2 against previous elements and swaps them.
1 1 4 2 5 6
Third run checks element 3 against previous elements and swaps them.
1 1 2 4 5 6
Fourth run checks element 4 against previous elements. No swap is necessary.
1 1 2 4 5 6
Fifth run checks element 5 against previous elements. No swap is necessary.
1 1 2 4 5 6

Figure 1.3: Insertion Sort: Figure adapted from Insertion Sort (n.d.)

2.4 Counting Sort

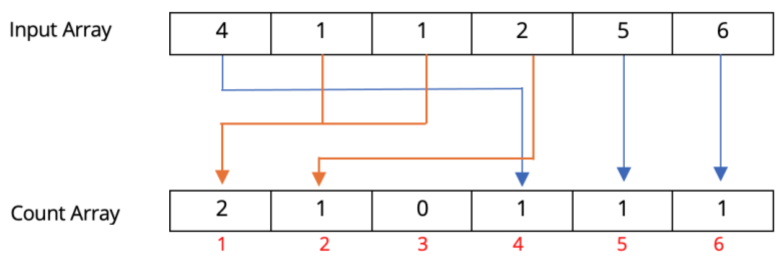
Counting sort is an out-of-place, non-comparison based, stable sort algorithm (Counting Sort in Java, 2022). Unlike most sorting algorithms that compare element in a dataset, the counting sort algorithm counts the number of times each unique element appears in the input, and then it performs arithmetic operations to determine the positions of elements (Non Comparison based Sorting Algorithms n.d.). In essence, this algorithm creates a histogram of the elements in the input array (Carr 2023c). Counting sort is useful when the range of input data is static (Sorting Algorithms Explained 2019).

Counting sort has time complexity of O(n + k) where *n* is the input size and *k* is the maximum value in the input dataset (Counting Sort in Java, 2022; Non Comparison based Sorting Algorithms, n.d.). Counting sort's space complexity is O(n + k) (Non Comparison based Sorting Algorithms, n.d.) or O(max), where max is the largest range of element Counting Sort Algorithm (n.d.).

Figure 2.4 present the execution process of counting sort.

The algorithm retrieves the minimum value and maximum value among the elements in the array:
Minimum = 1; Maximum = 6.
The algorithm creates a 'count ' array of size (6 – 1 + 1 = 5).

The first for-loop iterates though the input array and counts the number of instances of a particular element. The count array stores instances of each element and they are stored in chronological order.



The next for-loop iterates though the input array and replaces it with values sorted in the count array.

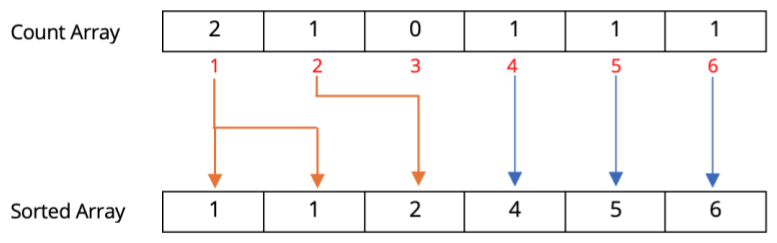


Figure 2.4 Counting Sort: Figure adapted from Geekific (2021)

2.5 Merge Sort

Merge sort is a recursive, out-of-place, comparison based, stable sort algorithm that uses a *divide and conquer* paradigm when sorting data (Merge Sort in Java, 2022). By dividing the data into sub-data, the algorithm has a log linear time complexity, meaning it scales well as the input data increases in size. This algorithm is useful if you want stable sorting, and a consistent performance for average and wort-case sorting scenarios (Carr 2023c). Merge sort's time complexity is O(n log n) and it has a linear space complexity of O(n) (Merge Sort in Java, 2022).

Figure 2.5 present the execution process of merge sort.

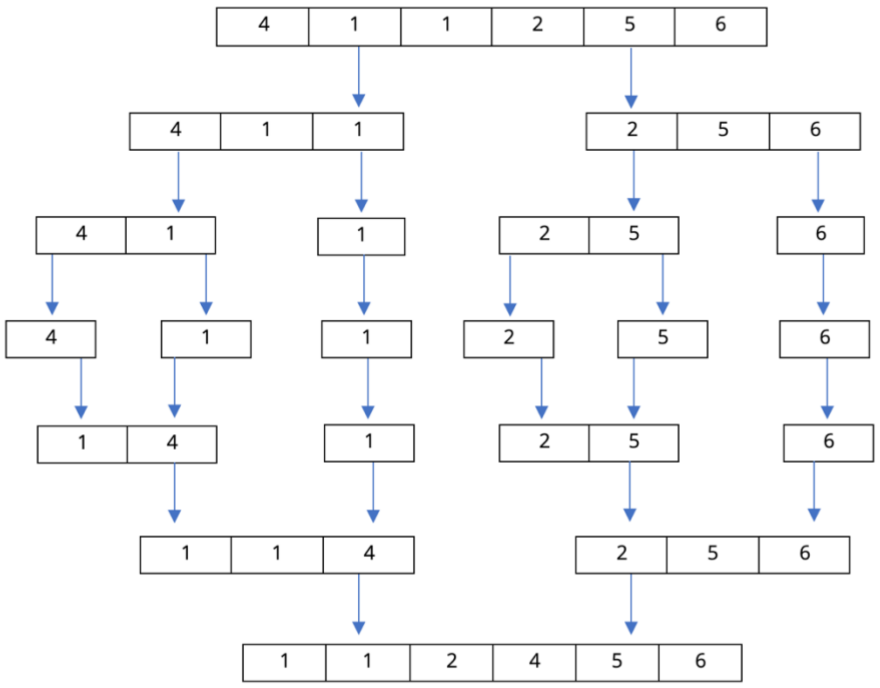


Figure 2.5: Merge Sort: Figure adapted from Merge Sort Algorithm (2023)

3. Implementation and Benchmarking

This section discusses the implementation of the benchmarking tool and explores the results.

3.1 Implementing the Application

I designed the benchmarking tool following best practices in Object Oriented Programming that I learned from Dr John Healy in Object Oriented Software Development. My goals were to create classes that implemented not more than one function, avoid repetition, and to write reusable code. I took influence from Porter *et al.* (2023) for the time measurement implementation of each algorithm. Additionally, I cited Carr (2023b) for the bubble sort, selection sort, insertion sort algorithms, Geekific (2021) for the counting sort algorithm, and Merge Sort Algorithm (2023) for the merge sort algorithm.

I used individual classes for the sorting algorithms, the bench-marker and the data table. I find that this approach keeps things tidy, reusable and easy to refactor. In retrospect, I probably should have created a separate class for each algorithm, but the application seems to function properly all the same. I used a class called application interface to host interface menu and to keep the application running in a do-while loop. Finally, I became aware early on in the development that you need to create a new array to sort for each iteration of a test, so I designed the bench-marker class in such a way that it can dynamically create arrays of random number to specified sizes. It was very hard to prevent repetition of code in the bench-marker class.

3.2 Results

The test in this section was compiled on the Java Virtual Machine in Mac OS 11.6.7. The hardware was a 2015 MacBook Pro, Intel Core i5 2.6Ghz processor and 8 GB of RAM.

Table 3.1 presents the results of the benchmarking exercise. The times are in milliseconds. Additionally, each row entry is an average of ten runs for each array size column.

Table 3.1: Benchmark results.

Size	**100**	**250**	**500**	**750**	**1000**	**1250**	**2500**	**3750**	**5000**	**6250**	**7500**	**8750**	**10,000**
Bubble Sort	0.031	0.126	0.500	1.117	1.474	2.183	8.712	19.597	37.814	61.240	93.575	134.090	184.474
Selection Sort	0.007	0.032	0.110	0.214	0.362	0.546	2.066	4.441	7.812	12.031	17.919	24.487	31.129
Insertion Sort	0.003	0.015	0.053	0.118	0.216	0.346	1.368	2.943	5.223	8.213	11.341	15.677	19.962
Counting Sort	0.003	0.004	0.007	0.013	0.014	0.017	0.037	0.050	0.071	0.082	0.101	0.116	0.133
Merge Sort	0.011	0.027	0.207	0.065	0.088	0.109	0.225	0.362	0.516	0.627	0.770	0.935	1.053

Next, figure 3.1 presents the findings of the benchmarking results plotted on graph representing time and input size.

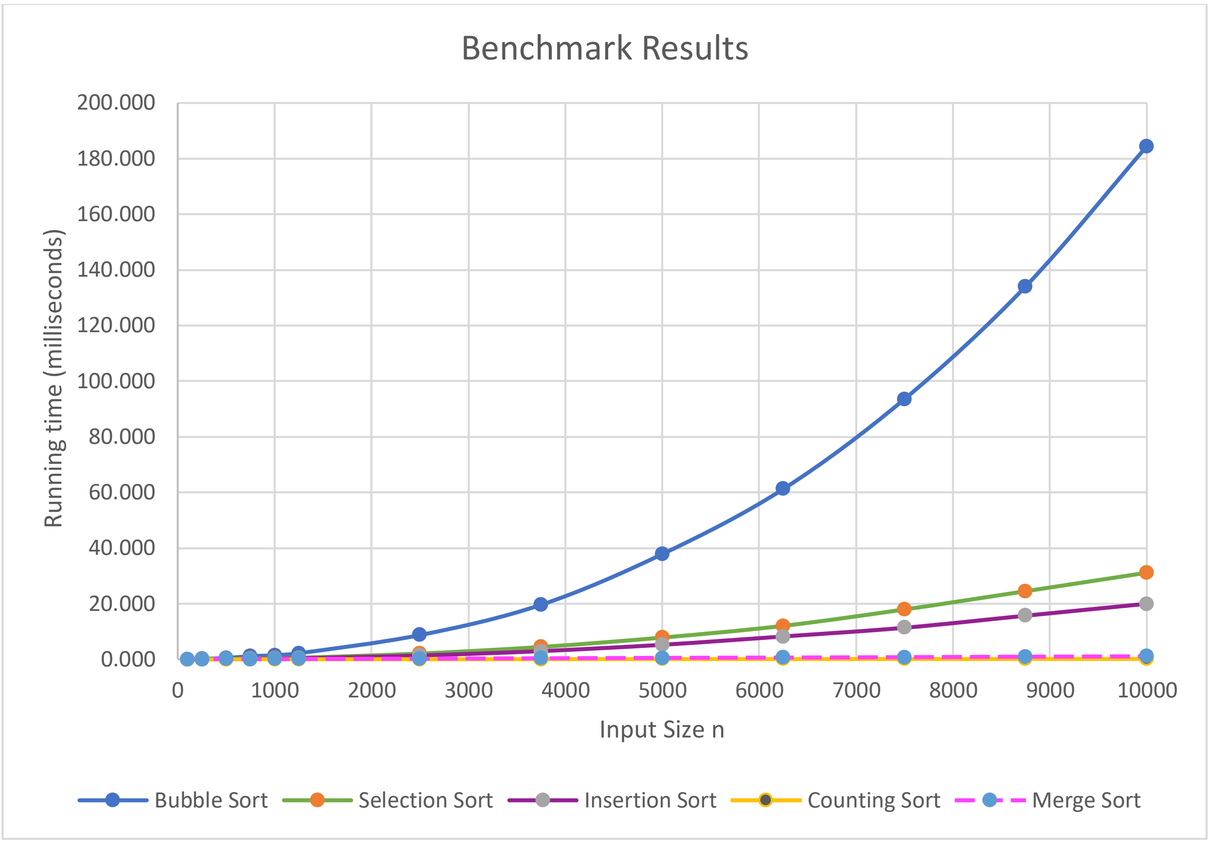


Figure 3.1: Benchmark Results. As the running time of counting sort is very low, it is hard to see it on the graph.

Table 3.1 and figure 3.1 show that bubble sort becomes increasingly slower as the input rate gets higher. This make it the slowest scaling algorithm among the five tested. Selection sort and insertion sort have about the same performance when handling larger data sets, with insertion sort performing marginally better. Counting sort and merge sort have the best performance as both algorithms maintain a low running time even as the data set become larger. In table 3.1, we can see that counting sort performed better than merge sort.

From my research in the previous sections, I assert that bubble sort has the slowest runtime because of its quadratic time complexity, and by the way that it compares and swaps each element of an input array. In comparison, the manner in which selection sort and insertion sort compare elements in an array optimises their runtime. Additionally, insertion sort's logic optimises its performance over selection sort, even though selection sort it supposed to have the fewest amount of comparisons (Sorting Algorithms Explained, 2019). However, the quadratic time complexity of bubble sort, selection sort and insertion sort mean that they are always going to slow down at a fast rate when processing large unsorted data sets.

Counting sort and merge sort out perform the other sorting algorithms due to their linear (counting sort) and log linear (merge sort) time complexity. Additional, their logic supports an optimised approach to processing large amounts of data. Although counting sort out performed merge sort, its worth noting that counting sort is only useful for sorting numeric data (Geekific, 2021). This limitation highlights the effectiveness of merge sort's *divide and conquer* logic because it can handle more data types.

Finally, although the time complexity of counting sort and merge sort is better able to process large data, there is a payoff with space complexity, as both algorithms consume much more memory than the simple sort algorithms. It would be interesting to see how each of the five algorithms perform on hardware where memory is limited.

Conclusion

This report documented the topic of sorting algorithms and presented details about the development and findings of the benchmarking application I developed for this project.

The key findings were:

- Counting sort performed the best in terms of time complexity.
- Merge sort also performed very well in regard to time complexity.
- Selection sort and insertion sort performed with around the same time complexity, but with insertion sort performing slightly better.
- Bubble sort had the worst time complexity, being much slower than the other algorithms when handling large data sets.

To build on this study, I plan to:

- Add more types of sorting algorithms to the benchmarking tool.
- Research how space and time complexity impact other data structures
- Research how sorting algorithms perform on hardware with limited memory capacity.

References

Bubble Sort in Java (2023) *Baeldung*, 15 February, available: https://www.baeldung.com/java-bubble-sort [accessed 15 April 2023].

Carr, D. (2023a) Analysing Algorithms, 46887 Computational Thinking with Algorithms available: https://vlegalwaymayo.atu.ie/pluginfile.php/907178/mod_resource/content/0/CTA___Week_5___Analyzing_Algorithms.pdf [accessed 13 April 2023].

Carr, D. (2023b) Simple Sorts, 46887 Computational Thinking with Algorithms available: https://vlegalwaymayo.atu.ie/pluginfile.php/917614/mod_resource/content/0/CTA___Week_9___Simple_Sorts.pdf [accessed 13 April 2023].

Carr, D. (2023c) Sorting Algorithms, 46887 Computational Thinking with Algorithms available: https://vlegalwaymayo.atu.ie/pluginfile.php/926550/mod_resource/content/0/CTA%20-%20Week%2012%20-%20More%20Sorting%20Algorithms.pdf [accessed 05 May 2023].

Comparison of Sorting Algorithms (2019), After Academy, 25 December, available: https://afteracademy.com/blog/comparison-of-sorting-algorithms/ [accessed 15 April 2023].

Counting Sort Algorithm (n.d.) *Java T Point*, available: https://www.javatpoint.com/counting-sort [accessed 13 April 2023].

Counting Sort in Java (2022) *Baeldung*, 23 June, available: https://www.baeldung.com/java-counting-sort [accessed 16 April 2023].

Geekific (2021) 'Counting Sort Explained and Implemented with Examples in Java', *Sorting Algorithms* [Video] available: https://www.youtube.com/watch?v=YEabFTMDczQ&ab_channel=Geekific [accessed 08 April 2023].

Healy, J. (2022) Big-O Notation, 51848 Advanced Object Oriented Software Development, available: https://vlegalwaymayo.atu.ie/pluginfile.php/670738/mod_resource/content/2/hdipSDDS-BigO.pdf [accessed 13 April 2023].

Insertion Sort (n.d.) *Wikipedia*, available: https://en.wikipedia.org/wiki/Insertion_sort [accessed 30 April 2023].

Insertion Sort in Java (2023) *Baeldung*, 15 February, available: https://www.baeldung.com/java-insertion-sort [accessed 15 April 2023].

Non Comparison based Sorting Algorithms (n.d.). Open Genus, available: https://iq.opengenus.org/non-comparison-based-sorting/ [accessed 15 April 2023].

Merge Sort Algorithm (2023) Geeks for Geeks, available: https://www.geeksforgeeks.org/merge-sort/ [accessed 03 April 2023].

Porter, L, Minnes, M. Alvarado, C. (2023) ‘Core: Using Java Time’ *Data Structures and Performance [*Video] available: https://www.coursera.org/lecture/data-structures-optimizing-performance/core-using-java-time-lTy4W [accessed 03 April 2023].

Selection Sort (n.d.) *Wikipedia*, available: https://en.wikipedia.org/wiki/Selection_sort [accessed 30 April 2023].

Selection Sort Algorithm (n.d.) *Java T Point*, available: https://www.javatpoint.com/selection-sort [accessed 13 April 2023].

Selection Sort in Java (2022), *Baeldung*, 23 May, available: https://www.baeldung.com/java-selection-sort [accessed 15 April 2023].

Skiena, S. (2020) *The Algorithm Design Manual,* 3rd ed. Switzerland: Springer.

Sorting Algorithms (2023), *Geeks for Geeks*, 21 Mar, available: https://www.geeksforgeeks.org/sorting-algorithms/ [accessed 13 April 2023].

Sorting Algorithms Explained (2019), Free Code Camp, 04 December, available: https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/ [accessed 15 April 2023].

Stable Sorting Algorithms (2022) *Baeldung*, 23 June, available: https://www.baeldung.com/cs/stable-sorting-algorithms [accessed 15 April 2023].

Time Complexity of Sorting Algorithms (n.d.) *Java T Point*, available https://www.javatpoint.com/time-complexity-of-sorting-algorithms [accessed 13 April 2023].

Understanding Space Complexity (2021), *Baeldung*, 02 August, available: https://www.baeldung.com/cs/space-complexity [accessed 15 April 2023].