Computational Thinking – Sorting Algorithms Bench Marking, Project Report.

Richard Daly
G00246442

Table of Contents

Introduction

This report is going to examine 5 different sorting algorithms: Bubble sort, selection sort, insertion sort, merge sort, and counting sort by benchmarking how long each algorithm takes to run against a given input. First this introductory section will look at what sorting is, and relevant concepts associated with sorting algorithms. The Sorting Algorithms section will examine each algorithm individually and lastly how the benchmarking was implemented and a discussion of the results in Implementation and Benchmarking.

Sorting

In computer terms, sorting refers to reorganising many items or a data set into a specific order. It could be alphabetical, lowest-to-highest value, distances etc. In terms of Object Orientated Programming it could refer to sorting objects in a pre-determined order. A sorting algorithm will take a data set as an input, complete some different operations against that data and output the data set in sorted order. For the purposes of this report and the associated application the algorithms are going to be sorting numbers. An array of numbers {2, 4, 6, 4, 4, 2} would be sorted to a natural ordering of {2, 2, 4, 4, 4, 6}. These numbers will be continuously used through the report to explain the workings of different algorithms with the use of bespoke diagrams.

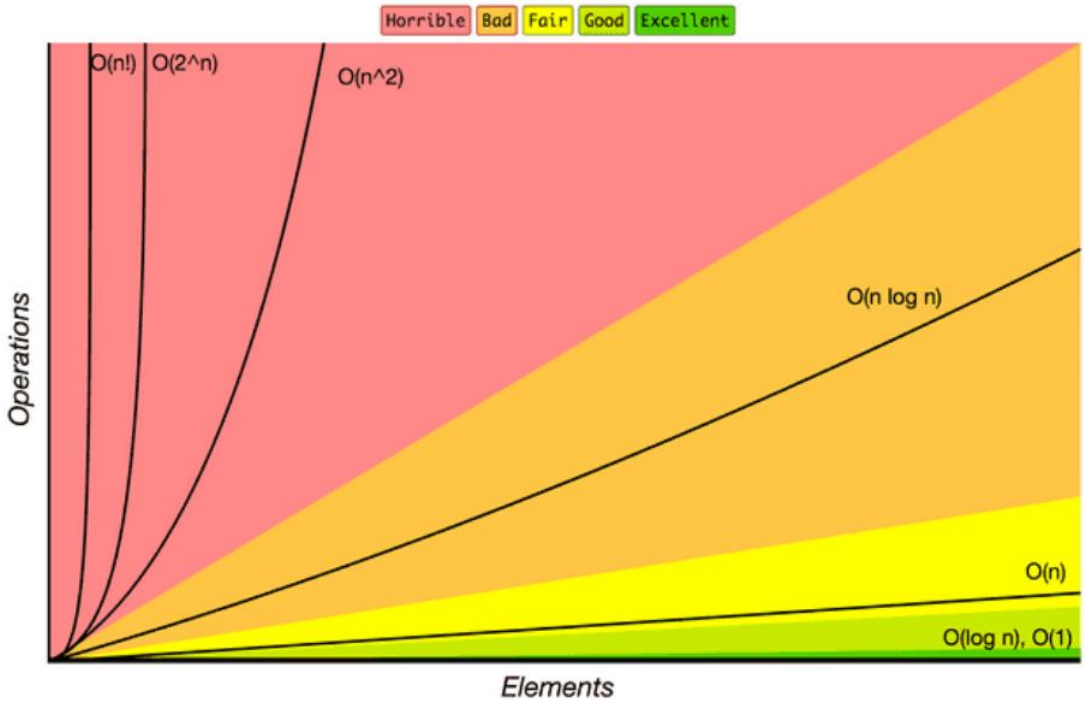
Time and Space Complexity

Time complexity is not to be mistaken with the amount of time it takes for an algorithm to run, the application will be measuring this, but it is not its complexity. Complexity refers to the execution of the statements that make up an algorithm and how they are influenced by the size of the input data. A good algorithm executes quickly ideally, so time complexity measures how quickly it is expected to run and how that runtime will grow with the size of the input. Asymptotic notations are used to express time complexity. Space complexity refers to the amount of space in computer memory that an algorithm will require to store data while it computes. They are both important so that an efficient algorithm can be identified, does the algorithm still run fast as its inputs grows?

Big O Notation a type of asymptotic notation that is used to measure the worst case in the running time of an algorithm and most used notation. For example, O (1 * n + 4) is notation where ‘n’ is defined by the input and the statements within an algorithm, because it is worst case in this notation it would simply be expressed as O(n) due the fact that ‘n’ grows linearly with the size of the input and the cause of the worst case. The following table shows different orders of magnitude of big O notation growing against different input sizes.

Input Size	Constant	Logarithmic	Linear	Log Linear	Quadratic	Cubic
n	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4096
1024	1	10	1024	10,240	1,048,576	1,073,741,824
1,048,576	1	20	1,048,576	20,971,520	10^{12}	10^{16}

The table could be expressed in the following graph showing the asymptotic growths on a curve. While they are all similar on small data sets the growth rates change significantly based on the algorithm and the operations within that it gives its notation and growth rate. This graph will be used later in the discussion of the results of the benchmarking and is a good reference when discussing the workings of each algorithm and their time and space complexity.



Big-Omega (Ω) notation looks at the best case of the algorithm and Big-Theta (Θ) notation applies when both the best and worse cases are the same. Best, worst, and average case is given for each algorithm in this report.

Performance

The performance of an algorithm is the actual runtime of an algorithm on the a given computer. It can be influenced by many factors such as the system architecture, CPU design, Operating system, and background processes to name a few. The application associated with this report will be measuring this and comparing it against the complexity of the algorithm and those results will be shown later. The effects of the growth rate of a given algorithm should be reflected in its performance. The growth rates are in orders of magnitude where 1, 10, 100, 1000 are orders of magnitude of each other, therefore the actual performance of an algorithm may fall anywhere within a scale of values and that is why best, worst, and average case complexities are given. The actual performance depends on the computer, the code, it growth rate and the input.

In-place Sorting

This refers to the means in which an algorithm is sorted:

"In-place means that the algorithm does not use extra space for manipulating the input but may require a small though non-constant extra space for its operation. Usually, this space is O(log n), though sometimes anything in O(n) (Smaller than linear) is allowed." (GeekforGeeks, 2023).

A sorting algorithm with an array of numbers {2, 4, 6, 4, 4, 2} that sorts the numbers within that array is an in-place sorting algorithm, if an algorithm created a new array and transferred some of those numbers to that new array before reorganising them within the original array is not an in-place sorting algorithm as it requires additional space to compute which may also scale with the input depending on the algorithms complexity.

Bubble, selection, and insertion are examples of in-place sorting while merge and counting are not. On examination of these algorithms, this will become apparent.

Stable Sorting

This refers to how an algorithm treats elements within a data set that have equal values and their relative ordering. An algorithm is stable if it sorts the elements, but equal elements still retain their original order.

While this is not as apparent in numbers if an algorithm was sorting a data set of key value items for example and a student name and a course. First if the names were sorted in alphabetical order, this may be fine but if the algorithm was to then sort them by course and the algorithm was not stable, the course would be sorted but the ordering of the names could be lost. A stable algorithm on the other hand would maintain the ordering of the names and still sort by course.

The only algorithm in this report that is not stable is the selection sort where a demonstration will be given.

Comparator Functions

The algorithms in this report will be mainly looking at numbers where simple Boolean based comparisons will be used and return true or false depending on if a number smaller than (<), greater than (>), equal to (==), less than or equal (<=), and greater than or equal (>=). As numbers already have a natural ordering it will allow for a less complex application.

However, if the subject that was being sorted was an object without a natural order, within Java there is an interface comparator when implemented correctly will allow a comparison. While it is out of the scope of this report, it is important to note that sorting is not limited to subjects that already have a natural order.

Comparison & non-Comparison Sorting Algorithms

Bubble, selection, insertion, and merge sort are all examples of comparison-based sorting algorithms that use the comparator functions described above. Counting on the other hand is a non-comparison-based algorithm, that sorts with out the comparator functions. This will be shown in the counting sort section. Another class of sorting algorithms exists, the hybrid sorting algorithm which is a combination of algorithms. While none exist in this report, the ones examined here are contained with hybrid sorting algorithms for example Tim sort uses both insertion and merge sort.

Sorting Algorithms

In this section each algorithm that is used as part of the project will be covered. When speaking about the sorting of the data for this project the data sets will be numbers, with the goal of sorting them in natural order left to right. It will be mainly a priori analysis looking at each algorithm from a theoretical perspective. Later in the section Implementation and Benchmarking, a posteriori analysis will be shown and the differences between the two.

> ***"In theory, theory and practice are the same. In practice, they are not."***
>
> Albert Einstein

Bubble Sort

This is popular introduction to the concept of sorting algorithms due to its simplicity, it generally considered easy to implement and understand. It is a stable sorting algorithm and comparison based. While it works well on small data sets such as the example in the bespoke diagram on the next page, it is not well suited to large data sets since it must make multiple passes. This will be shown later in this report when looking at the results of the benchmarking done as part of this project. Due to this it does not perform well in real world use and is more used now as an educational tool.

The following is a quick breakdown of the space and time complexity of bubble sort:

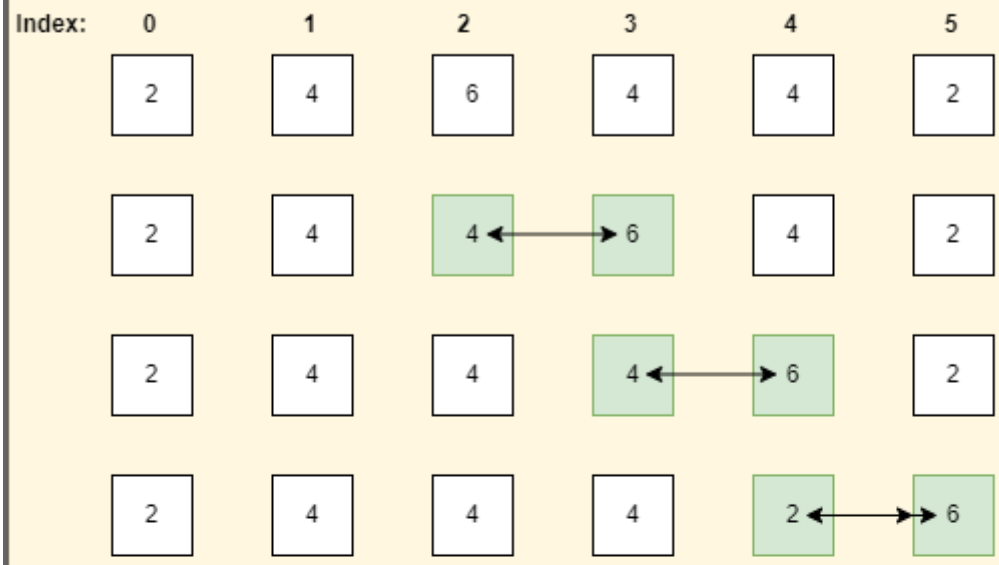
Best Case	Worst Case	Average Case	Space Complexity	Stable Sorting
n	n^2	n^2	1	Yes

As can be seen the worst and average case for bubble sort are the same, meaning that in general as the input of n increases in number the running time will start to rise quite dramatically. While it as a best case of O(n), this is rarely seen unless the data is already sorted or near sorted. This algorithm is sometimes used in computer graphics on nearly sorted data were possibly a swap of only two elements is needed which can be done in linear time (GeeksforGeeks, 2023).

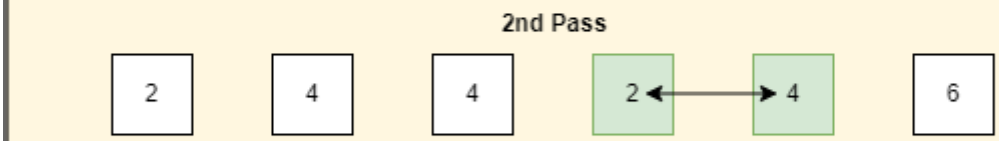
While bubble sort is asymptotically similar to the selection and insertion algorithms which will also be examined in this project, they differ greatly. Owen Astrachan a researcher in Duke University has done experimentations showing bubble sort to be roughly one fifth as fast as insertion sort and seventy percent as fast as selection sort (Wikipedia, 2023 and Astrachan, 2023).

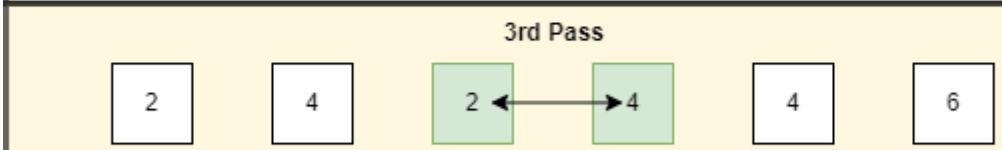
Bubble sort repeatedly passes through data bubbling the elements out to either end. The following bespoke diagram will show the algorithm on a small set of data. Every time it passes through the data it compares one element against the next even if already sorted. The most inversions (switching data between two indexes) occur during the first pass in this example. The remaining passes have been shortened for the purpose of the diagram, but each index is still inspected. It will only complete when it is able to make a pass without any inversions needed.

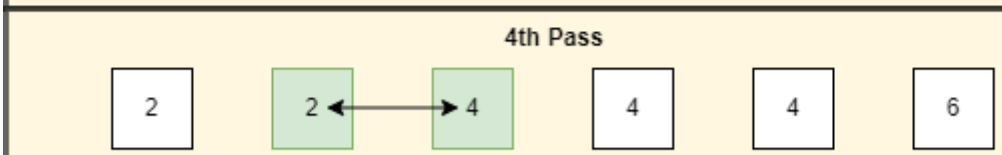
This is the first pass of the array in the bubble sorting algorithm. 3 Inversions will occur. Firstly index 0 will be compared with index 1 and then index 1 with index 2. It will continue in this pattern comparing each index against the next checking for natural ordering.



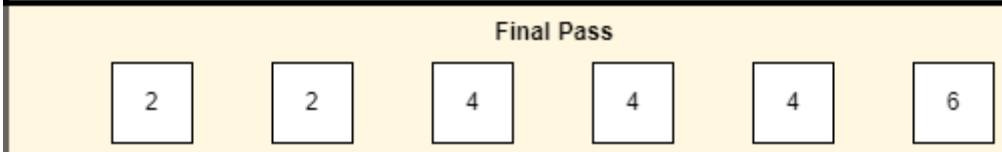
Bubble sort will continue to make passes through the array until there are no more inversions that can occur.







Finally once there is no more inversions that can occur bubble sort will pass through the array a final time and complete the loop on the array it implements.



Selection Sort

Like bubble sort, this sorting algorithm is also comparison based, but is not stable like bubble sort. It is also easy to implement and understand. It is more efficient than bubble sort but less efficient than insertion sort. This is mainly shown with large data sets, all three are similar on small data sets.

The following is a quick breakdown of the space and time complexity of selection sort:

Best Case	Worst Case	Average Case	Space Complexity	Stable Sorting
n^2	n^2	n^2	1	No

Time complexity is the same in each case. Theta notation can be applied here as the best and worst case of the algorithm are the same. In comparison with bubble the best case of this algorithm is $O(n^2)$ instead of O(n). This is because with bubble sort if the data is already sorted, only one pass would be required. Selection sort must find the minimum element in every iteration and to do so has to traverse the entire array.

Selection splits an array into two virtual sub arrays. The sorted elements in an array on the left and the unsorted on the right. Starting with the first index, the presumed smallest element it will compare all elements to the right to find the smallest element. If no element is found no change occurs, if a new smallest element is found the two elements will switch positions. The new smallest element will now be in a sorted array on the left and will not be checked again. This process continues with the next index being presumed the smallest and then compared with the rest of the array until the last element is reached constantly moving the smallest element into the sorted sub array.

The bespoke diagram on the next page will show this in action. In this diagram the fact that selection sort is not stable can be seen. The element with the value 4 at index 1 in the 2nd pass is moved after the other equal elements with same value. If their already relative order was of importance this algorithm would not maintain it. Going back to the example in the introductory explanation of stable sorting if the three elements with the value 4 were instead key values pairs of student name and course number they would be reordered as follows:

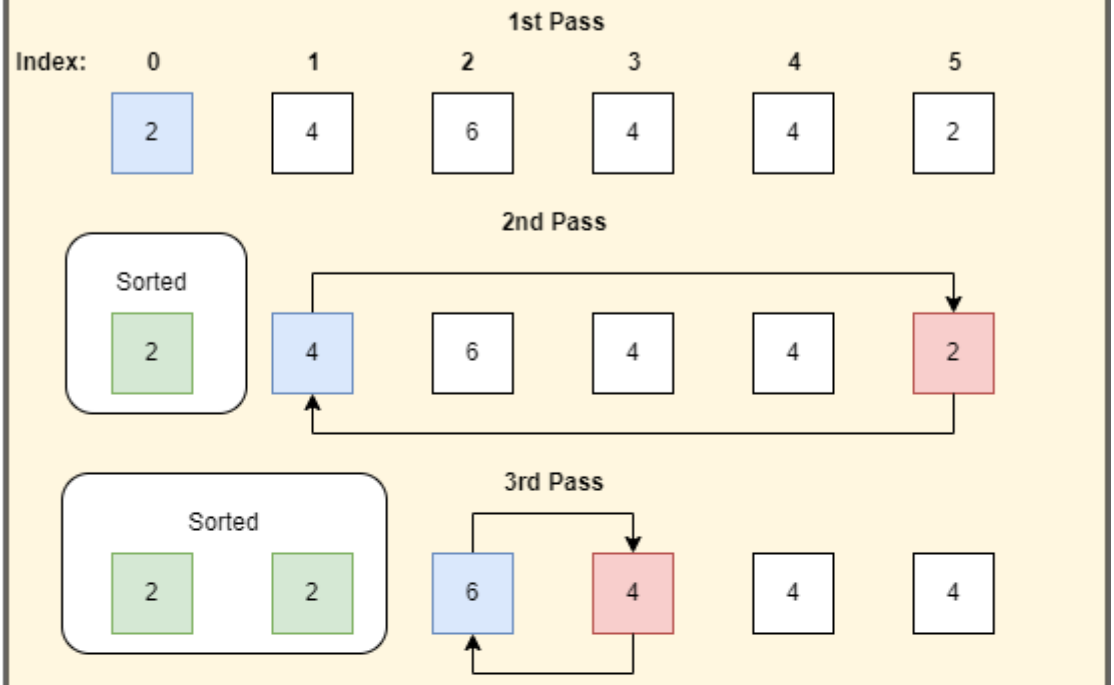
Before		
Alex, 4	Bob, 4	Claire, 4
After		
Bob, 4	Claire, 4	Alex, 4

While the have been sorted if the alphabetical ordering of the names was of important, that has now been lost.

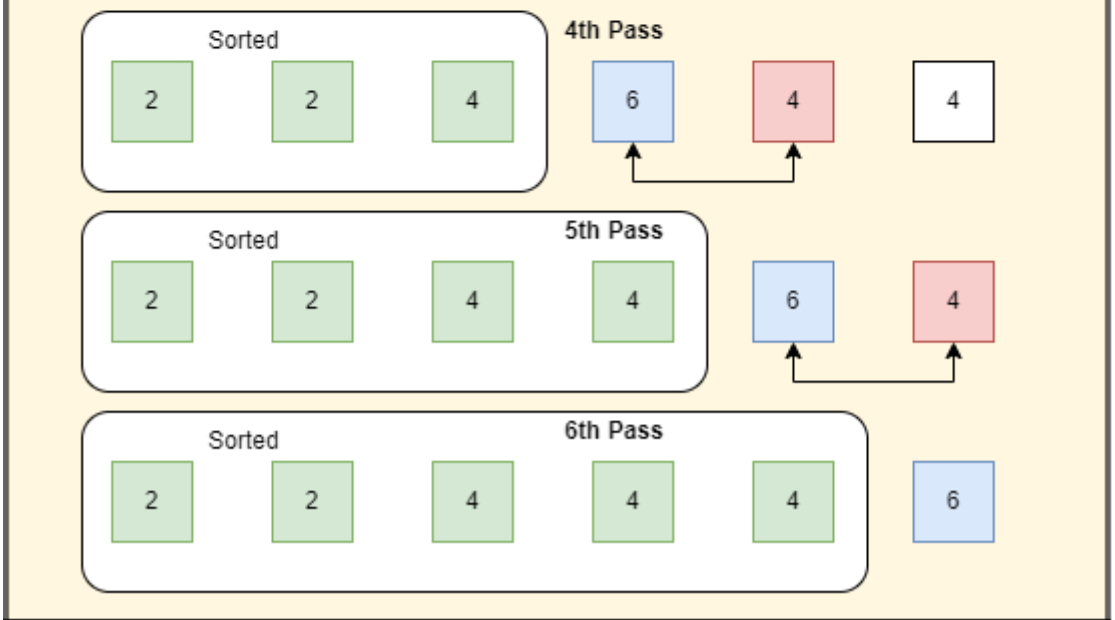
1st Pass: index 0 is the minimum to be checked against entire array. Smallest value so no swap.

2nd Pass: index 1 is now the minimum. Index 5 found to be smaller, inversion of elements.

3rd Pass: index 2 is now the minimum. Index 3 found to be smaller and then compared against rest of array. Nothing Smaller so index 2 and 3 elements are swapped.



This pattern continues in the same fashion passing the element 6 to the end of the array.



As index 5 is the last index and the largest element, the data is now sorted.

Insertion Sort

This algorithm like both bubble and selection is comparison based. It is also stable like bubble. With a best-case time complexity of O(n). It is also like bubble in that this will occur on already sorted or very near sorted data. As discussed previously while it is asymptomatically the same as bubble and selection, insertion sort will in most cases outperform the other two.

The following is a quick breakdown of the space and time complexity of insertion sort:

Best Case	Worst Case	Average Case	Space Complexity	Stable Sorting
n	n^2	n^2	1	Yes

Insertion is different to the previous two in that as it iterates through the array it compares the selected index against all previous elements. For this reason, it starts from the second index in the array so that it already has something to compare against. Like selection there is virtual sorted and unsorted arrays. While selection sort did not look at the sorted subarray, insertion sort will as it will be inserting elements into that array.

Starting with the second index insertion sort will treat that element as a key and compare it against the sorted sub array on the left. If it finds an element that is smaller, it will place the key after that element. If it does not find any element smaller, it will be placed in the first index. To insert an element into the sorted array any element that is greater will moved up one place in a knock-on effect until the space where the key originally was is filled.

This will be demonstrated in the bespoke diagram on the next page, especially in the last step when nearly the entire array is shuffled forward to allow the element to be placed in its correct position.

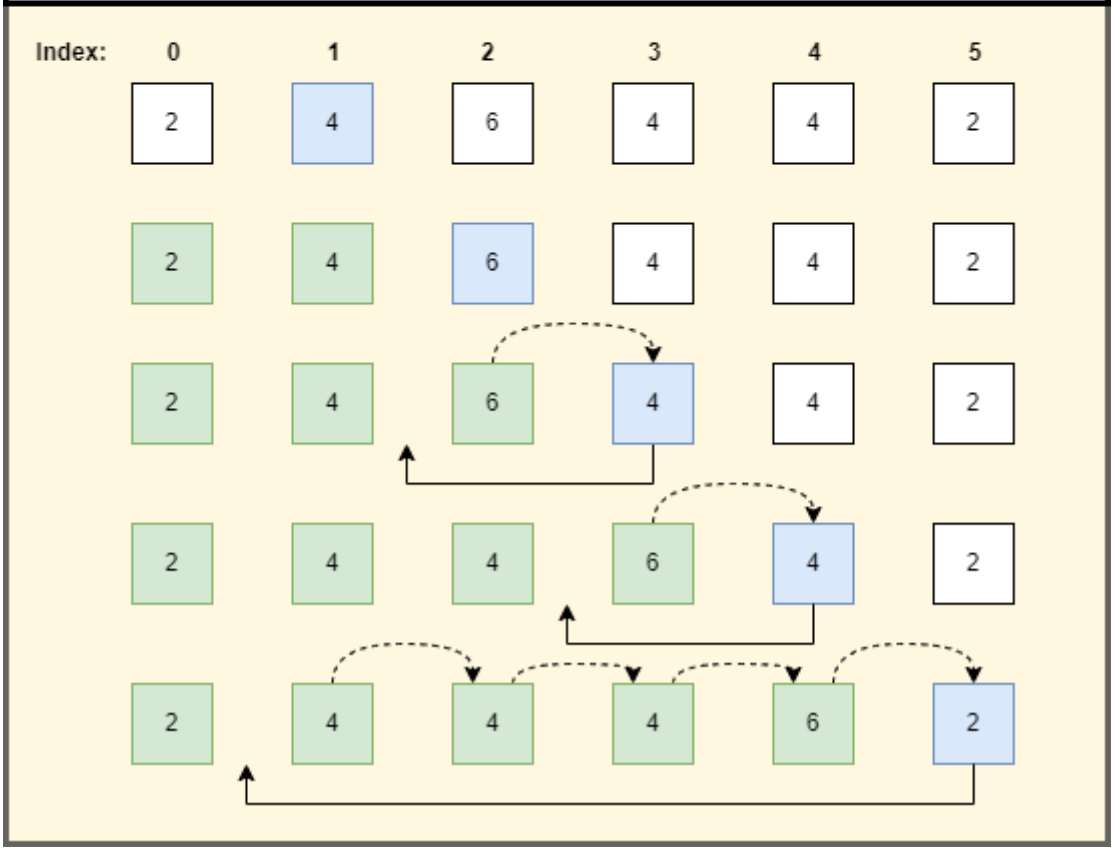
Insertion is more efficient than selection in that when comparing an element against the rest of the array, insertion stops once it finds an element smaller whereas selection must continue through all the elements to check that there is nothing smaller. Insertion due to being stable will retain the relative ordering of elements. Using the same example as in selection:

Before		
Alex, 4	Bob, 4	Claire, 4
After		
Alex, 4	Bob, 4	Claire, 4

Insertion Sort presumes that the first element is sorted. Taking the second element index 1 and treating it like a key it compares it against each element before it until it finds its correct position. As can been seen in this example first two comparison runs are already sorted.

With index 3 as the key, the first insertion occurs. Working backwards 4 is less than 6 but not greater than four and its position is found. 6 is moved forward to allow for the insertion.

The last has the biggest change as the element 2 at index 5 is less than all elements preceding it but the first index. All elements from index 1 are moved forward to allow for the insertion.



Merge Sort

This algorithm while it is comparison based like the preceding algorithms it is one of the most efficient algorithms. It uses a divide and conquer strategy, through recursion it continuously cuts down the original into sub arrays of only one element and then merges them together into a sorted array (Khandelwal, 2023). Due to this reason merge sort requires additional space in the complexity of O(n), while it takes up more space it is acceptable due to being significantly more efficient.

Merge sort has a time complexity of n log n in all cases making it a very popular algorithm due its predictable runtime. In comparison to the previous algorithms, merge can handle large data sets in more efficient time as this report is hoping to show during the benchmarking process. Merge sort is parallelizable which has not been explored in this report until now, meaning that it be adapted to work with multiple processors or threads (GeekforGeeks, 2023).

This is the first algorithm in the report that is not a in place sorting algorithm. Bubble, selection, insertion are all sorted in place. This may be of concern if the application using the algorithm was restricted on space available. Many variations of this algorithm are now used, as well as being used in more complicated hybrid algorithms that combine one or more algorithms.

The following is a quick breakdown of the space and time complexity of merge sort:

Best Case	Worst Case	Average Case	Space Complexity	Stable Sorting
n log n	n log n	n log n	O(n)	Yes

The following diagram on the next page is going to show the process of merge sort broken down into sections:

Section A:

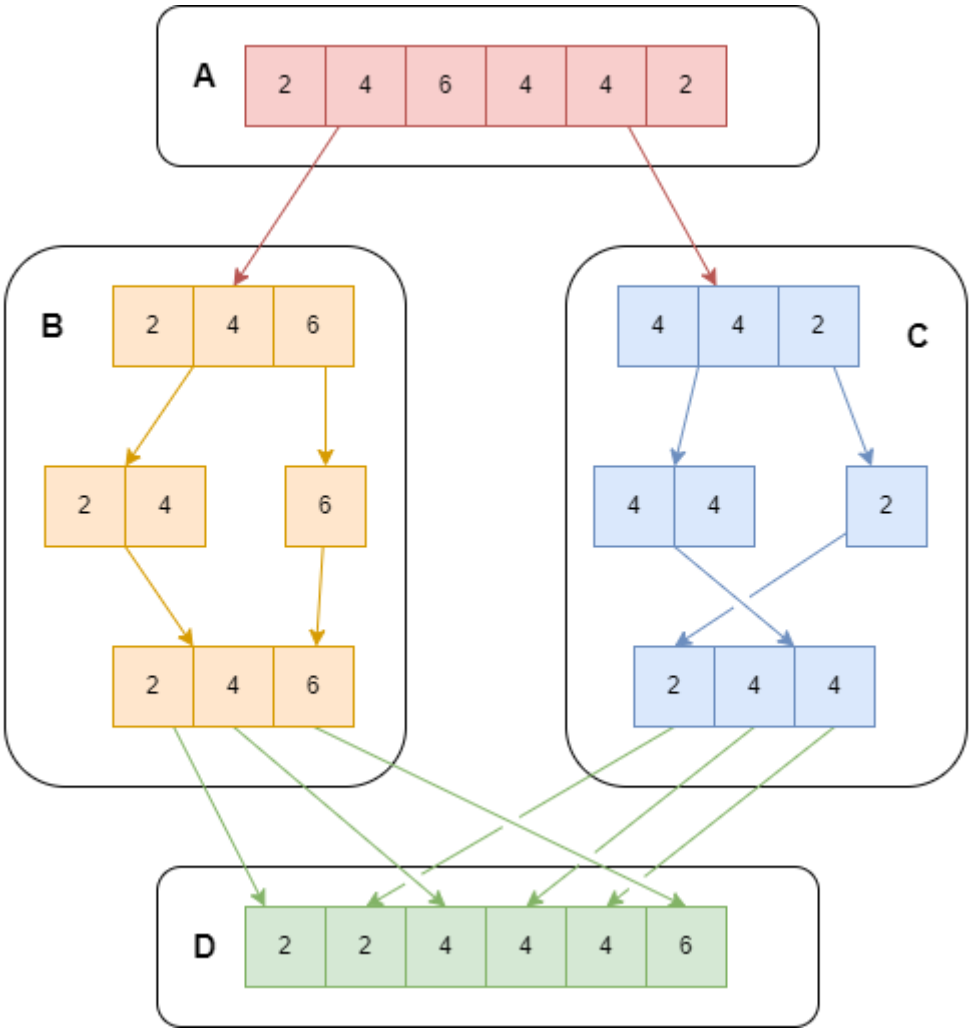
The original unsorted array is split into two arrays from the middle point. All elements are copied into these two new arrays.

Section B + C:

The array is then recursively split into further smaller arrays until they contain less than two elements. Then each array is iterated through comparing the elements of one against the other passing the elements in natural order into a new merged array. No change can be seen in **B,** but a reordering of elements occurs in the merging of **C**.

Section D:

When the recursion is finished, the algorithm comes back to the original call with two sorted merged arrays. These two arrays are then compared against each other and placed into one sorted array that is a merge of **B & C**. Finally, this array is copied back into the original array.


A
2
4
6
4
4
2
B
2
4
6
2
4
6
2
4
6
C
4
4
2
4
4
2
2
4
4
D
2
2
4
4
4
6

Counting Sort

This algorithm is an example of a not an in-place sorting algorithm and the reports only non-comparison-based algorithm, distinguishing itself from all the previous algorithms. In terms of time complexity, it is the most efficient algorithm in this report with best, worst, and average cases time complexity all being n + k, the same applies to the amount of space it requires. Where n is the input and k is the range of those inputs. The algorithm usually must find the range of inputs itself but if there is prior knowledge, it can be optimised further.

While counting and merge are both more complicated than the rest of the algorithms and therefore that bit harder to implement, they are often explored from an educational perspective after bubble, selection, and insertion. They both show far more efficient performance.

The following is a quick breakdown of the space and time complexity of counting sort:

Best Case	Worst Case	Average Case	Space Complexity	Stable Sorting
n + k	n + k	n + k	n + k	Yes

The following bespoke diagram on the next page is going to show the process of counting sort broken down into sections:

Section A:

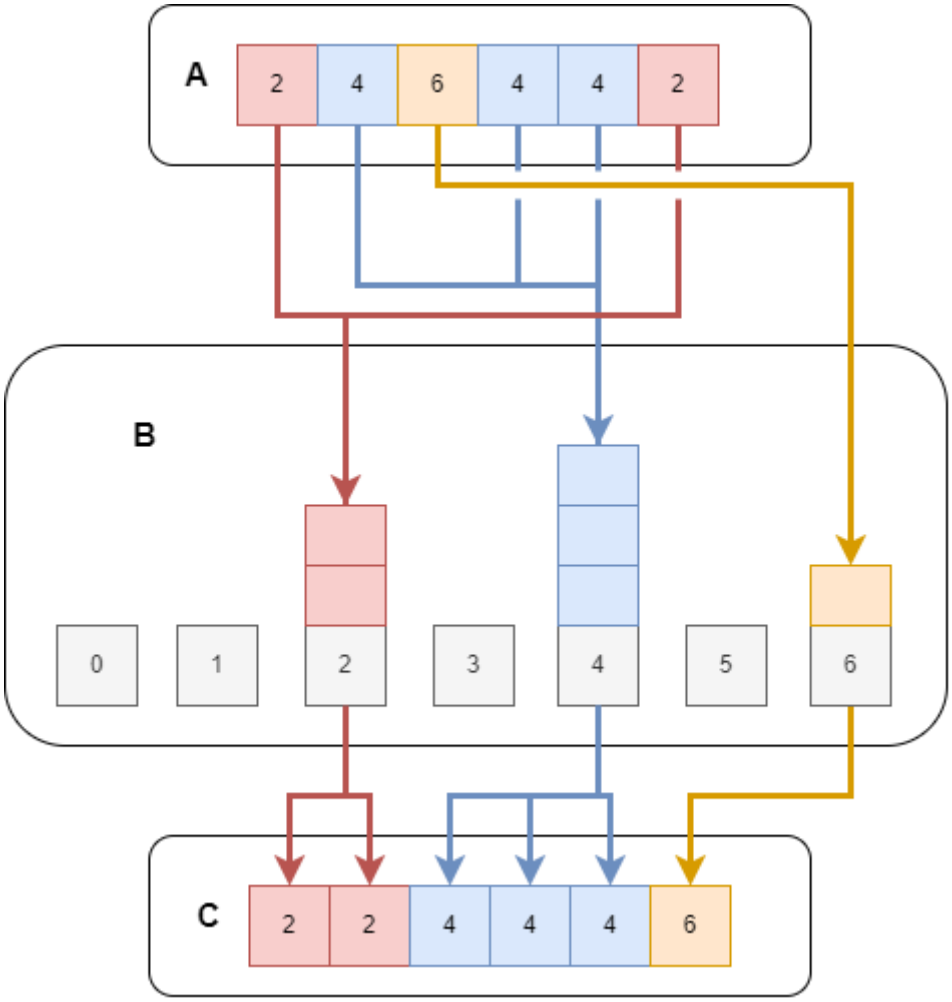
The original unsorted array is iterated through to find the largest element that will be used to initialize the next step in section **B.** This could be removed if a prior knowledge of the range was known.

Section B:

A new array (count array) is made with the size of largest element plus one, that will be used to count how many times an element occurs in the original array. All indexes in the array are first initialized with a zero value and as can be seen are incremented on each occurrence. This stage is often referred to as a histogram and can also be viewed as a form of key value pairs.

Section C:

All elements are copied into the original array in order by iterating through the count array which are already in sorted order.


A
2
4
6
4
4
2
B
0
1
2
3
4
5
6
C
2
2
4
4
4
6

Implementation & Benchmarking

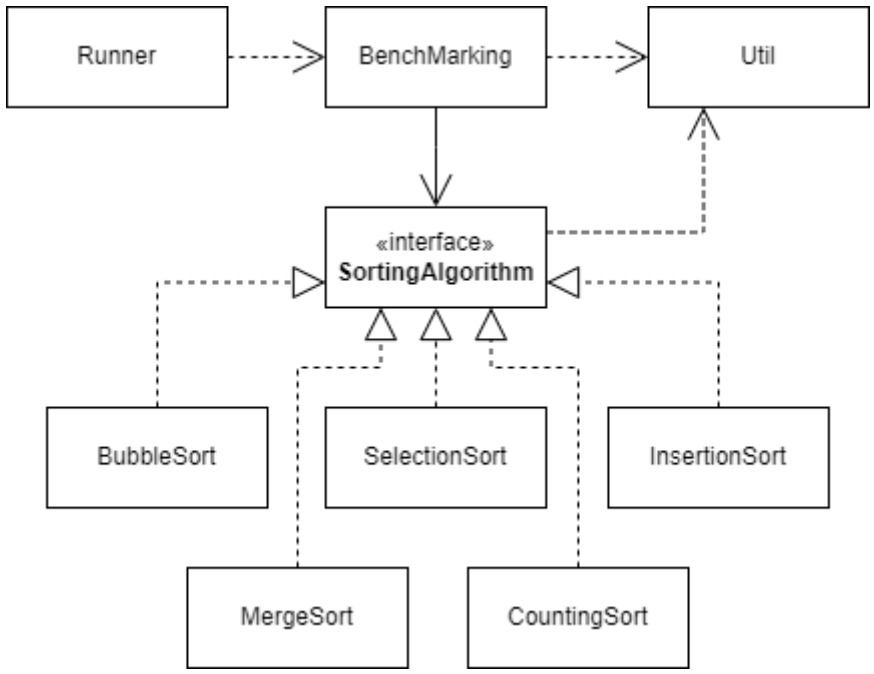
Implementation

To do a posteriori analysis of the different algorithms discussed in this project, a Java application was created. The main goal is testing each algorithm with a variety of different size input values and get a performance running time of each. All the inputs are created to be arrays of positives numbers with randomly generated values, with a new array being used for every test. Each algorithm is run ten times in total and the average is returned in the value of milliseconds. To get as accurate as possible results the timing was measured in nanoseconds and then converted to milliseconds. All results are outputted as a table with the values rounded to three decimal places in milliseconds.

It is important to note at this point that the application when run will be affected by elements out of control of the application and could influence the results of bench marking process. This can include but is not limited to system architecture, CPU design, operating system, and background processes to name a few. This is where complexity and performance collide, where complexity influences performance but not the other way around.

To make the application as efficient as possible and with as little code bloat as possible, a strategy design pattern was implemented. Each algorithm is implemented in its own Java class, which inherits the methods of a sorting algorithm interface designed for benchmarking. With the Liskov Substitution Principle then put into practice in a bench marking Java class.

The following is a simplified UML diagram of the application:



Benchmarking Results

Two runs of the application with different sets of inputs were run. The following tables and matching graphs are a representation of the results. In the tables Sizes refers to the input size which would be the size of the array with random values being sorted. All times are in milliseconds.

Table 1:

Sizes	100	1000	2000	4000	5000	6000	7000	8000	9000	10000
Bubble Sort	0.213	1.606	2.971	12.782	20.42	31.064	44.522	60.495	78.507	100.494
Selection Sort	0.075	1.031	1.069	3.99	6.2	8.955	12.095	15.913	19.977	25.011
Insertion Sort	0.045	0.737	0.307	1.197	1.796	2.612	3.667	4.582	5.847	7.228
Merge Sort	0.052	0.201	0.384	0.463	0.581	0.733	1.307	0.656	0.742	1.236
Counting Sort	0.01	0.076	0.11	0.094	0.081	0.095	0.028	0.026	0.047	0.067

Graph 1:

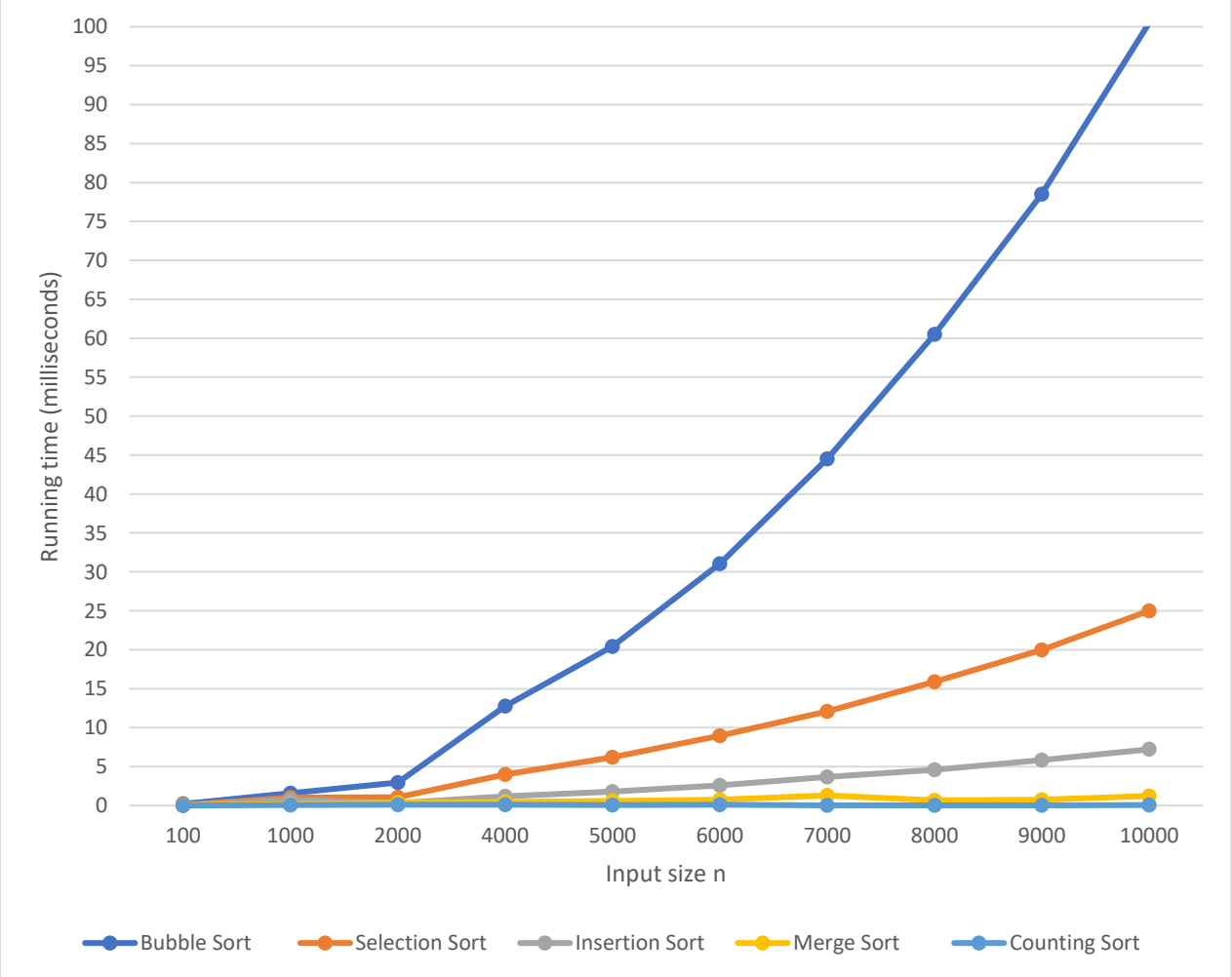
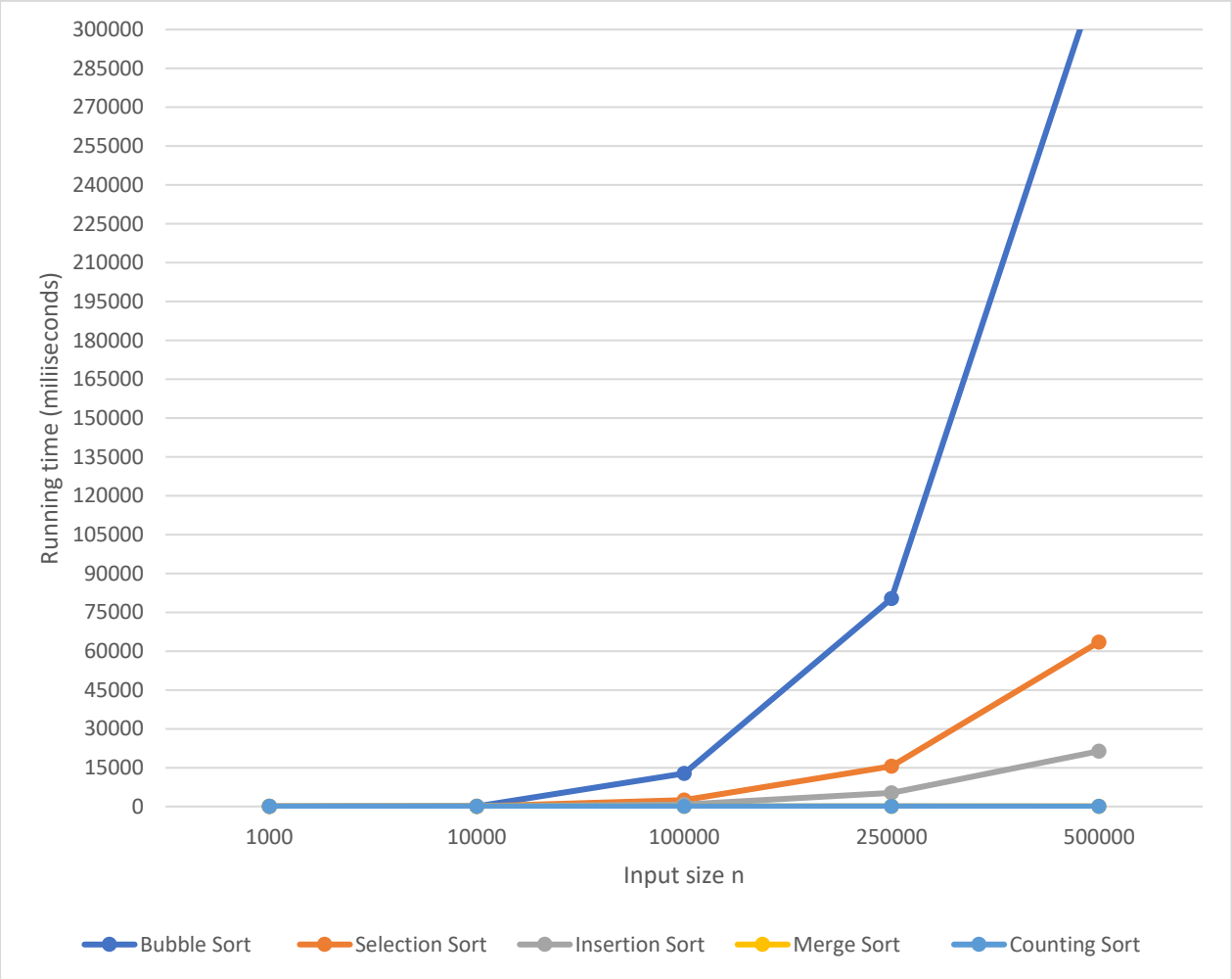


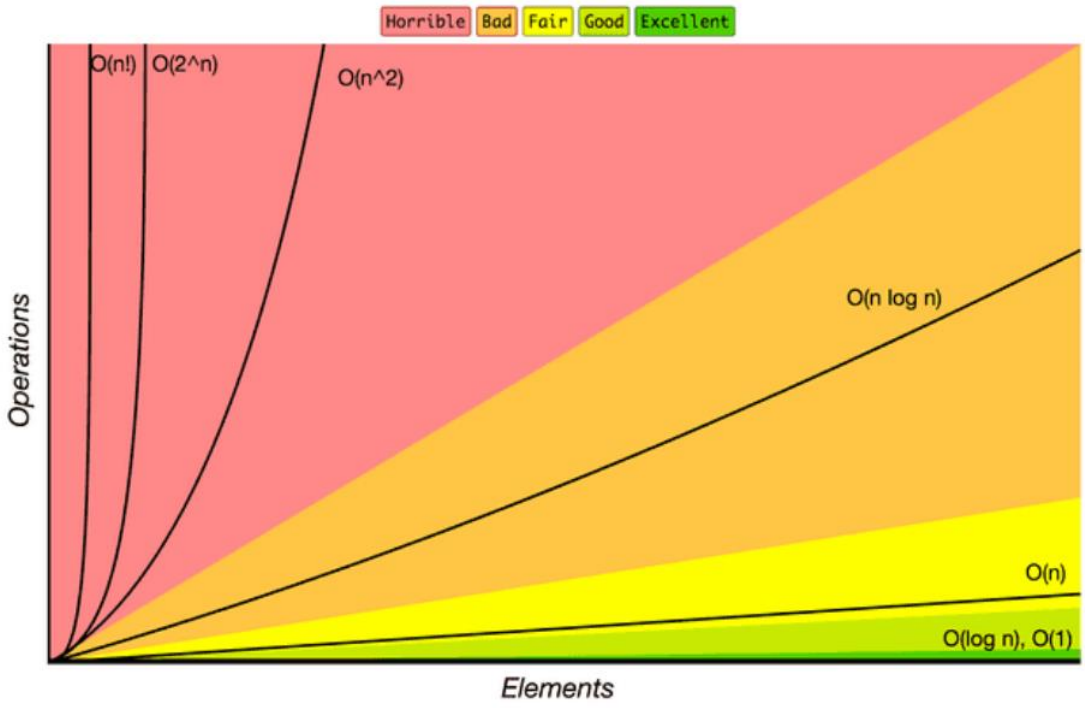
Table 2:

Sizes	1000	10000	100000	250000	500000
Bubble Sort	1.721	103.717	12702.019	80235.291	322215.4
Selection Sort	1.789	24.819	2447.423	15584.709	63524.805
Insertion Sort	0.744	7.307	817.279	5205.661	21367.252
Merge Sort	0.227	1.192	15.414	27.664	65.93
Counting Sort	0.075	0.229	1.413	1.003	2.045

Graph 2:



Looking at the results in table 1 and graph 1, both merge sort and counting sort are outperforming the rest and are what would have been expected, with counting being more efficient than merge. Looking back at the graph presented early in the report merge sort has a time complexity of n log n and counting sort n + k with merge taking more time than counting similarly to the graph. While the curves of results are far different than the diagram below it is important to note that this is representational of the expected growth in theory. Looking at the final column in table 1 with the input size of 10000, each result is significantly different to the next. Order of magnitudes represent ranges in that 1, 10, 100, 1000 are different orders of magnitude and with that in mind each result is significantly different.



The results of bubble, selection and insertion sort were the most surprising and somewhat unexpected since they all share a time complexity of n^2. This is what lead to running the application a second time with different inputs of much greater value shown in table 2 and graph 2. Merge and counting sort again are as expected, significantly outperforming the rest again. While the other three are still performing very differently, graph 2 show them starting rise more significantly with bubble sort showing just how inefficient at large data sets it is.

At this point it worth going back to what was discussed in the bubble sort section in relation to Astrachan's research that bubble sort is roughly one fifth as fast as insertion sort and seventy percent as fast as selection sort. This may account for the significant difference as well as out of the control of aspects affecting the application. They may all still fit within their order of magnitude which table 2 seems to reflect in the last column of input size 500000, which overall before the average was calculated, the time taken for the application to complete the test was very long due to bubble, selection and insertion all taking most of the total running time all in five figure minimum average runtimes.

Overall while bubble, selection and insertion showed some surprising results each algorithm preformed as expected despite not necessarily being same to the theory. Merge and counting both matched their complexity.

References

Astrachan, O., 2023. *Bubble Sort: An Archaeological Algorithmic Analysis.* [Online]
Available at: https://users.cs.duke.edu/~ola/bubble/bubble.html
[Accessed May 2023].

Carr, D., 2023. *Analysing Algorithms,* s.l.: Atlantic Technological University.

Carr, D., 2023. *Analysing Algorithms Part Deux,* s.l.: Atlantic Technological University.

GeekforGeeks, 2023. *Bubble Sort Algorithm.* [Online]
Available at: https://www.geeksforgeeks.org/bubble-sort/
[Accessed May 2023].

GeekforGeeks, 2023. *Counting Sort.* [Online]
Available at: https://www.geeksforgeeks.org/counting-sort/
[Accessed May 2023].

GeekforGeeks, 2023. *In-Place Algorithm.* [Online]
Available at: https://www.geeksforgeeks.org/in-place-algorithm/
[Accessed May 2023].

GeekforGeeks, 2023. *Insertion Sort – Data Structure and Algorithm Tutorials.* [Online]
Available at: https://www.geeksforgeeks.org/insertion-sort/
[Accessed May 2023].

GeekforGeeks, 2023. *Merge Sort Algorithm.* [Online]
Available at: https://www.geeksforgeeks.org/merge-sort/
[Accessed May 2023].

GeekforGeeks, 2023. *Selection Sort Algorithm – Data Structure and Algorithm Tutorials.* [Online]
Available at: https://www.geeksforgeeks.org/selection-sort/
[Accessed May 2023].

GeekforGeeks, 2023. *Stable and Unstable Sorting Algorithms.* [Online]
Available at: https://www.geeksforgeeks.org/stable-and-unstable-sorting-algorithms/
[Accessed May 2023].

GeekforGeeks, 2023. *Types of Asymptotic Notations in Complexity Analysis of Algorithms.* [Online]
Available at: https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/
[Accessed May 2023].

Gupta, D., 2023. *Comparison Among Selection Sort, Bubble Sort, and Insertion Sort.* [Online]
Available at: https://www.interviewkickstart.com/learn/comparison-among-bubble-sort-selection-sort-and-insertion-sort
[Accessed May 2023].

Healy, J., 2022. *Big-O Notation,* s.l.: Atlantic Technological University.

Khandelwal, V., 2023. *What is Merge Sort Algorithm: How does it work, and More.* [Online]
Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/merge-sort-algorithm
[Accessed May 2023].

Tiwari, A., 2023. *Counting Sort Algorithm.* [Online]
Available at: https://www.interviewkickstart.com/learn/counting-sort
[Accessed May 2023].

Upadhyay, S., 2023. *Time and Space Complexity in Data Structure: Complete Guide.* [Online]
Available at: https://www.simplilearn.com/tutorials/data-structure-tutorial/time-and-space-complexity
[Accessed May 2023].

Wigmore, I., 2023. *Sorting Algorithm.* [Online]
Available at: https://www.techtarget.com/whatis/definition/sorting-algorithm
[Accessed May 2023].

Wikipedia, 2023. *Bubble Sort.* [Online]
Available at: https://en.wikipedia.org/wiki/Bubble_sort
[Accessed May 2023].

Wikipedia, 2023. *Counting Sort.* [Online]
Available at: https://en.wikipedia.org/wiki/Counting_sort
[Accessed May 2023].

Wikipedia, 2023. *Insertion Sort.* [Online]
Available at: https://en.wikipedia.org/wiki/Insertion_sort
[Accessed May 2023].

Wikipedia, 2023. *Merge Sort.* [Online]
Available at: https://en.wikipedia.org/wiki/Merge_sort
[Accessed May 2023].

Wikipedia, 2023. *Selection Sort.* [Online]
Available at: https://en.wikipedia.org/wiki/Selection_sort
[Accessed May 2023].