Computational Thinking - Sorting Algorithms Project

Student: Andrew Walker

Introduction

What is sorting?

The history when it comes to sorting algorithms, comes from manual methods all the way to machine computer algorithms today. With roots coming from Herman Hollerith's card sorting machines, to the development of Radix Sort, this was a milestone for paving the way for automated data processing (GeeksforGeeks, n.d.). Before computers, algorithms were sorted manual by hand, using sorting type for type cases for printing and sorting cards for things like census data.

As computers evolved areas like bubble sort, insertion sort, and selection sort emerged in the 20th century. When it came to merge sort and Quicksort, John von Neumann invented merge sort back in 1945, this was seen as a divide and conquer sort of approach when it came to sorting. Quicksort, introduced by Tony Hoare back in 1960, this was seen as an efficient sorting algorithm and is still seen being used today (Cormen et al., 2009).

Why sorting matters?

Sorting would be seen as a fundamental operation in computer science. Such tasks as searching, database operations, and data presentation. Sorting matters with Improved and search efficiency. With sorted data finding certain items can be done at a much faster pace. Such algorithms as binary, which requires sorted data, can go through large datasets at a quicker pace which saves time then on searching through an unsorted dataset (W3Schools, n.d.).

With sorting it facilitates data analysis, with sorting you can reveal patterns and other trends, that can be difficult when data is unsorted. Many other algorithms such as merge and search rely on sorted algorithms to function most efficiently (Sedgewick & Wayne, 2011.

Why performance Matters

Performance is important for many reasons, sorting is crucial for optimizing algorithms and data structures, with these it leads to faster searches, improved data retrieval and system efficiency. With real world applications having an efficient algorithm can save on time, energy and resources. Having poor time/space efficiency ends up leading to slow apps, wasted processing and delays (TutorialsPoint, n.d.; W3Schools, n.d.).

Performance also helps with database indexing, this also for faster query execution and data retrieval, load balancing which is the distribution of tasks in a distributed system, this leads to better resource utilization. Caching is another reason why performance matters with sorting, with this performing to its best capacity it will have faster retrieval pf frequently used data.

But the area than can be affected most is the time and space complexity, the choice of what sorting algorithm is used impacts this area. Then in turn can directly affect the performance of the sorting process and the whole application. (**TutorialsPoint** and **W3Schools).**

To really measure a performance, we use the time and space complexity analysis. Using this idea is to measure the order of growths and then put the in terms of input size.

First let's talk about the time complexity aspect of the analysis. This is when the algorithm quantifies the amount of time that is taken to run the algorithm as a function of the length of the input. This is not to get confused with the execution time of the machine that the algorithm is running on.

The time it actual takes to solve the problem within the algorithm is called time complexity. This is the actual time that would be needed for the completion on the algorithm. To estimate this the cost of each fundamental instruction is to be understood and how many times these instructions are executed in the algorithm.

Another area that comes with sorting algorithms is In-Place vs. Out-of-Place sorting. With In-Place sorting, this sorts the array by using something called constant extra memory. What this does is modifies the input directly, its most common use would be bubble sort or insertion sort. This matters for algorithms that are ideal when memory can be limited. Then with Out-of-Place sorting it can use additional memory with creates copies or helper structures. An example of this would be merge sort, what this does is stores temporary arrays to store halves. This matters because it is safer and can be easy to implement the downfall would be it cost more memory (GeeksforGeeks, nd).

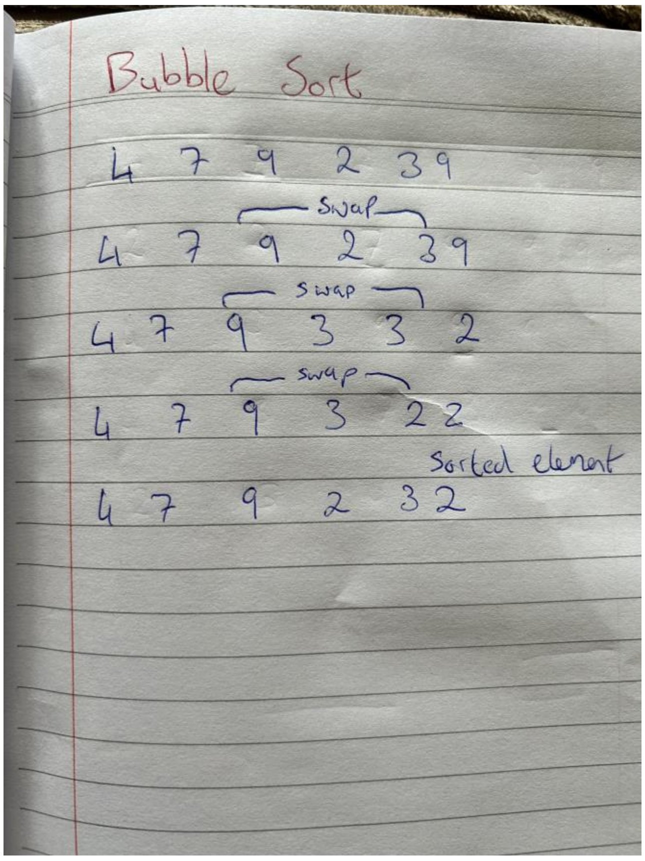
Then we move onto Stable vs Unstable sorting. With stable sorting it can preserve the relative order of equal elements. For example, if two students end up with the same result, this means that the original order is kept. This is used with merge and insertion. Unstable sort is something that rearranges equal elements. This is more common with selection sort and quick sort. This area matters because stability is crucial in multi-level sorting. Then finally we have Comparison-Based vs N0n-Comparison Based sorting.

With Comparison-base sorting it compares elements using symbols such as <, >, or ==. If you compare it to time complexity, at least with this it’s just 0(n log n). Examples of this again would be bubble sort, merge sort and quick sort. With non-comparison-based sorting, this doesn't compare elements it uses keys, counts, or digits. This type can be faster than comparison sorts. Example of this sort would be Counting Sort, Radix Sort, and Bucket Sort. Some of these sorting algorithms will be explained below (GeeksforGeeks, n.d.).

Sorting Algorithms

Bubble Sort

First sorting algorithm I used was Bubble sorting algorithm, this is a simple comparison-based algorithm. This compares adjacent elements and then it swaps them if there in the wrong order. What it does is it “bubbles the largest number to the end of the array with each pass. The process will be repeated until the list of numbers is sorted into the correct order. The term bubble comes from how the largest value in the array bubble up to the end. Bubble sort has a best case: O(n) (if already sorted), Average Case: O(n2), Space: O (1)-in-place and it is stable (GeeksforGeeks, n.d.).

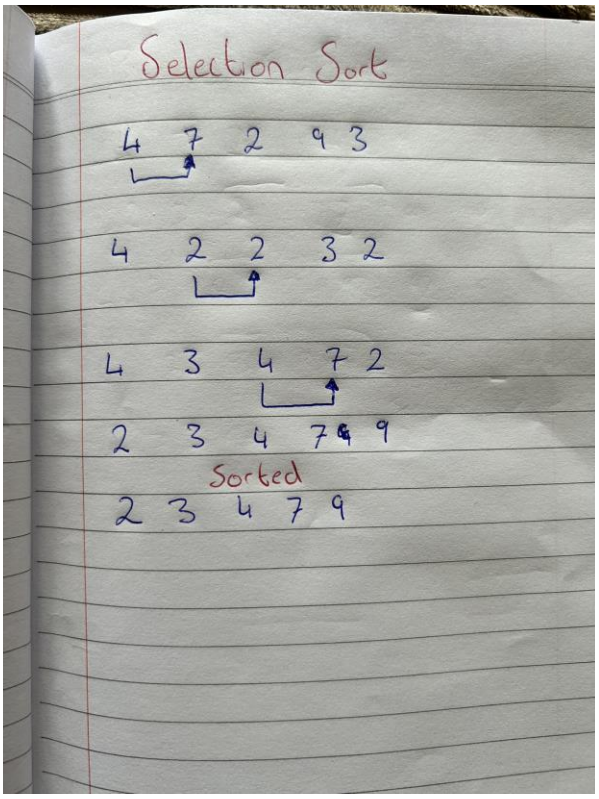


The advantages of Bubble sort are that it's a simple sorting algorithm to understand and implement. The method is a good example of teaching how sorting works. As mentioned, it’s in-place which means it doesn’t require extra memory.

Selection Sort

Next algorithm used was Selection Sort, this algorithm is done by selecting the smallest elements and moving them to the front. It will repeatedly select the smallest number from the unsorted until sorted. This is a simple comparison-based algorithm. The array gets divided into 2 parts the sorted on the left and the unsorted on the right. It works by starting on 0 then finding the smallest number in the array, swap it the number that's at the current index, then move to the next index and continue to repeat the process until sorted. For the time complexity the best case is O(n2), Average, O(n2), and Worst O(n2). It swaps up to n –1, the space complexity is O(1)- in-place, but it is not stable because it can swap non-adjacent equal values (GeeksforGeeks, n.d.).

Diagram of the steps:



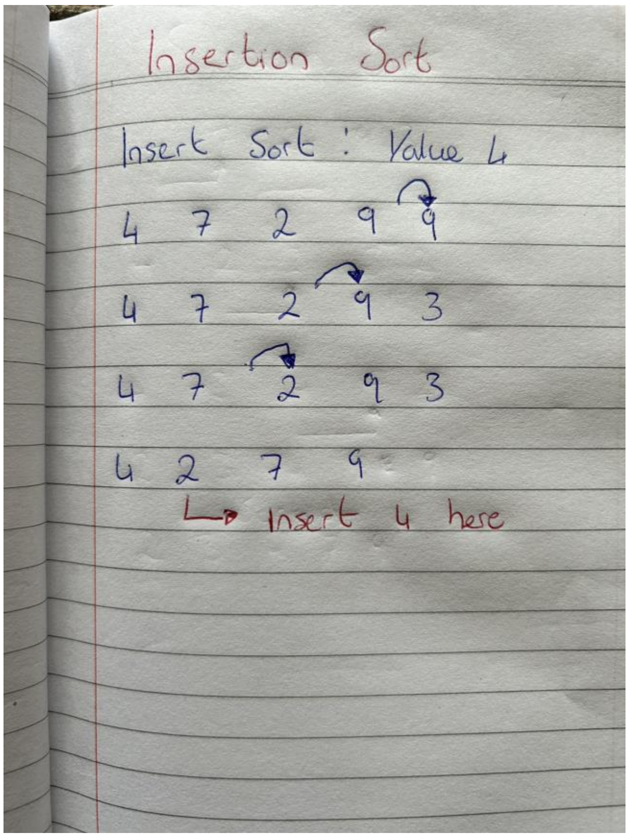
The advantage of this sort is, easy to understand and implement, and will perform fewer swaps than the likes of Bubble and insertion Sorts.

The Disadvantages would be that it is inefficient when it comes to large data sets, will always be O(n2), this is even the case if the array is already sorted. And as mentioned it is not stable.

Insertion Sort

Third algorithm used was Insertion Sort. This is a simple and intuitive way of using an algorithm, some references say it's like the way you might sort cards in your hands. The method of doing it is taking one number from the unsorted array, and it inserts it into the correct position back into the sorted array. Starting at the second number, compare this with the numbers before it, then we shift all bigger numbers to the right. Then insert the correct number into its right position. Repeat this process till all numbers are in a sorted array. It's time and space complexity would be best case O(n) which is already sorted, Average case is O(n2), worst case is O(n2) which would be reverse sorted and then its space is O(1) which is (in-place). This is a stable algorithm (GeeksforGeeks, n.d.).

Example using: [4, 7, 2, 9, 3, 9]



The advantages of this sorting algorithm would be that, with small dtat it is efficient. Performs well when arrays are nearly sorted. Like Bubble sort, insertion sort is easy and simple to implement. As mentioned above it is stable because the equal elements maintain order and in-place because extra memory is not needed.

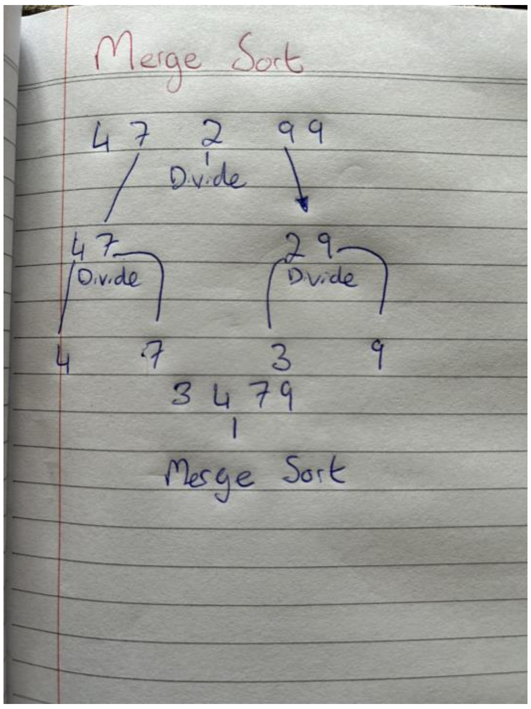
The disadvantages would be that, with large datasets iit would be more inefficient and is slower than the likes of merge sort, Quick Sort and Counting Sort with is a non-comparison.

Merge Sort

The fourth algorithm used was Merge sort. This type of sorting is efficient comparison-based algorithm, and its method is to divide and conquer. With the array you have it splits it into two halves, then it recursively sorts each half, and finally after all this it will merge the sorted halve, which makes it one full sorted array. This is seen as the most reliable and consistent sorting algorithm when it comes to using large datasets.

Step one is to divide, you divide the unsorted array into halves and keep doing this till the subarray has one number. Then conquer by sorting the subarrays and finally merge the subarrays back into one sorted array. The time and space complexity of this algorithm would be at best case O(n log n), average would be O(n log n) worst case O(n log n) and space O(n) this is due to the auxiliary arrays. It is stable.

Example using: [4, 7, 2, 9, 3, 9]

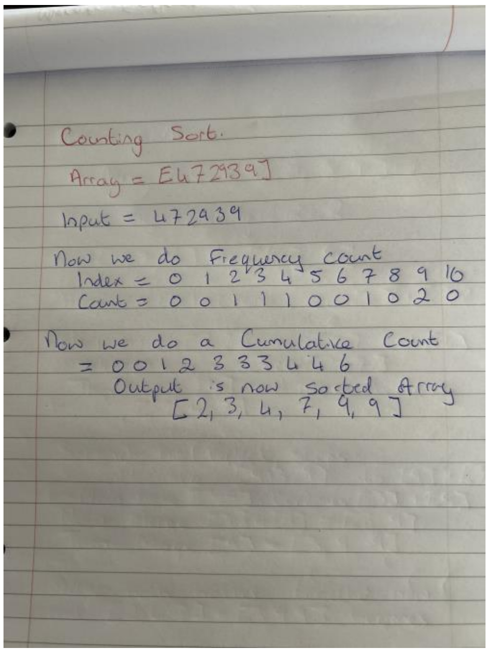


With this sorting algorithm there is advantages and disadvantages, the advantages of this algorithm are its more efficient for large data sets. Stable which means it maintains the order of equal numbers. The performance stays the same on worst-case input. Divide and conquer is something used in many algorithms.

The disadvantages would be that it requires O(n) additional space which would go as not in-place. When it comes to smaller data it would be slower compared to something like quick sort. When it comes to implement it, is does become more complex than such algorithms like Bubble and Insertion Sort (GeeksforGeeks, n.d.).

Counting Sort

Then the final algorithm that was used for this was Counting Sort. This type of sorting is a non-comparison-based algorithm. Instead of comparing the numbers in the array, it counts the occurrences of each number. Using this information to place the correct numbers in their correct positions turning it into a sorted array. When having a smaller range of numbers, it makes Counting sort exceptionally fast when sorting. It works by first finding the max number within the input array. Then we create a count array of + 1 and set all other values in the array to 0. after this count how many times the number appears in the array. To know the position of the number you go from count array to cumulative count. With the output array you create it by positioning number in the correct place. Finally copy the sorted result into the original array. The time and space Complexity for this would best 0(n + k), average 0(n + k), worst 0(n + k), space 0(n + k) and it is stable.



Counting Sort.

Array = [472939]

Input = 472939

Now we do Frequency count

```
Index = 0 1 2 3 4 5 6 7 8 9 10
Count = 0 0 1 1 1 0 0 1 0 2 0
```

Now we do a Cumulative Count

```
= 0 0 1 2 3 3 3 4 4 6
```

Output is now sorted Array
[2, 3, 4, 7, 9, 9]

The advantages of Counting Sort would, when it comes to a small range it is very fast, would-be linear time complexity (O(n)) this is when k is not much greater than n. Stable which means it maintains order of equal numbers. This is an algorithm that you don't need to involve comparisons. This is ideal if you have special use cases.

Disadvantages would be that without a modification it would not be suitable for negative numbers. Can't be used for large ranges. Then uses extra memory (GeeksforGeeks, n.d.).

Implementation and Benchmarking

With Java application I created five algorithms that would use the same benchmark for each. These were Bubble, Selection, Insertion, Merge and Counting Sort. To gain the educational resources for all five algorithms I used GeeksforGeeks as my logic-based implementation, with the English statements coming from my own knowledge of what each step in the algorithms would be doing. Each algorithm was placed in the same class, using a method for each. The main method was used to get random arrays of integers of different sizes, then run each algorithm about 10 time and System.nanoTime was used to measure how long each algorithm would take. Once this was done it calculated and printed the average times from 1-10 in milliseconds.

The implementation of each sorting algorithm followed a structed, modular approach in Java, using the function design, gained from GeeksforGeeks to encapsulate each sort. (GeeksforGeeks, n.d.).

When it comes to the benchmarking method, each algorithm was tested with the same method. For the arrays an increasing size of numbers was generated: so, for example (n = 100, 200, 300, 400, ……, up to a 1000). With each algorithm the array in them the sort then ran 10 times. To ensure fair testing a copy of the unsorted array was used. When the time was calculated it was done in milliseconds (Oracle, 2014).

When it came to analysing the result, it was clear that counting and merge were the most efficient to use out of all five algorithms. As seen below in the diagram. With Merge sort it was clear that it was consistently faster, even when the inputs were much larger. With counting, the performance was clear to work on all input sizes because it's a non-comparison based. The other algorithms with them being easy to implement they had much slower run times to show. The results from these aligned with the theoretical time complexities. It was clear that Merge Sorts 0(n log n) was much faster in performance compared to the simpler 0(n2). With counting sort being linear complexity, it performed well with small range for numbers (Big-O Cheat Sheet, n.d.).

Table Results

Size	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Bubble	47.566	37.584	32.944	36.684	34.375	31.035	32.806	31.604	31.250	31.532
Counting	4.163	1.592	1.605	0.948	0.954	0.864	0.894	0.905	0.906	0.856

Insertion	62.539	44.836	11.564	9.250	9.999	9.993	13.357	9.560	11.155	10.904
Merge	0.003	0.001	0.001	0.001	0.000	0.001	0.001	0.001	0,000	0.000
Selection	34.418	45.067	40.618	26.816	25.880	26.390	26.989	25.761	28.765	25.829

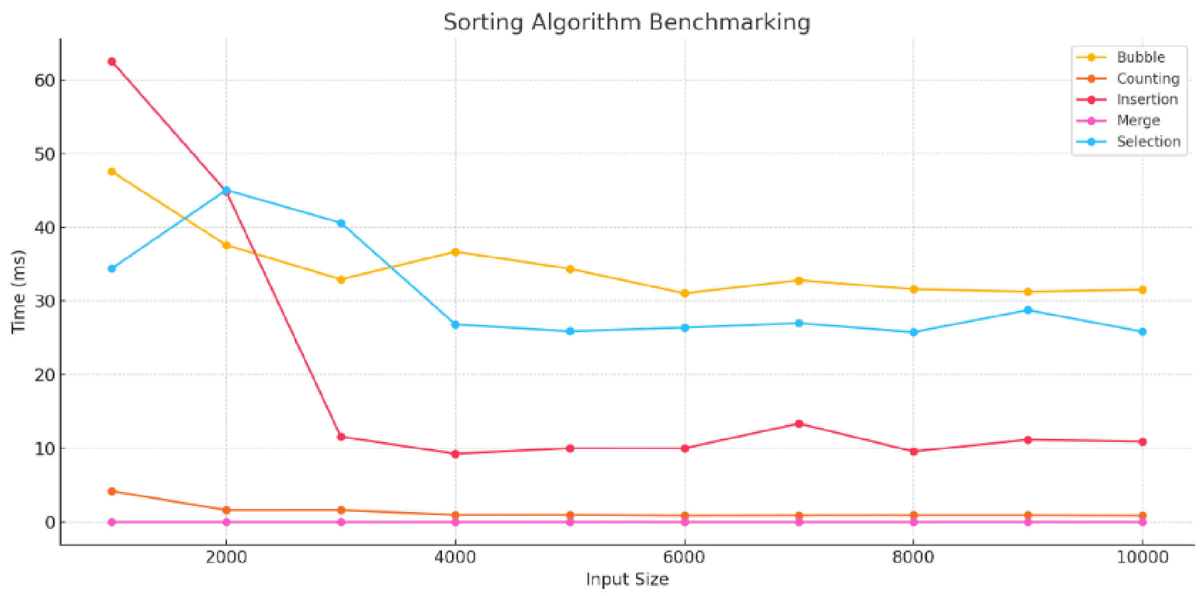
Bubble Average: 10 runs= 0.034738 seconds

Counting Average: 10 runs= 0.001369 seconds

Insertion Average: 10 runs=0.019316

Merge Average: 10 runs= 0.000001

Selection Average: 10 runs= 0.030653



Reference List:

Introduction to algorithms third edition (2009) : By Thomas H Cormen (author), Charles E Leiserson (author), Ronald L Rivest (author), Clifford Stein (author) : Free Download, borrow, and streaming (no date) *Internet Archive*. Available at: https://archive.org/details/introduction-to-algorithms-third-edition-2009 (Accessed: 14 April 2025).

Know thy complexities! (no date) *Big*. Available at: https://www.bigocheatsheet.com/ (Accessed: 14 April 2025).

(2025) *System (java platform SE 8)*. Available at: https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime-- (Accessed: 14 April 2025).

GeeksforGeeks (2025) *Sorting algorithms*, *GeeksforGeeks*. Available at: https://www.geeksforgeeks.org/sorting-algorithms/ (Accessed: 14 April 2025).

Online content. (no date) *Princeton University*. Available at: https://algs4.cs.princeton.edu/home/ (Accessed: 14 April 2025).

Tutorials on technical and non technical subjects (no date) *Tutorials point*. Available at: https://www.tutorialspoint.com/ (Accessed: 23 April 2025).

W3schools.com (no date a) *W3Schools Online Web Tutorials*. Available at: https://www.w3schools.com/ (Accessed: 14 April 2025).