Atlantic Technological University
Higher Diploma in Science in Software Development

Computational Thinking with Algorithms

Sorting Algorithms - Report

Student: Stjepan Tadic

Student ID: G00438839

Academic Year: 2023-2024

Lecturer: Dr. Dominic Carr

Table of Contents

Introduction

A sorting algorithm is a set of instructions used to arrange a collection of items, such as numbers or strings, in a particular order. The algorithm takes an unordered list or array of elements as an input, rearranges it using step by step approach, and outputs an ordered sequence of items.

Sorting algorithms are valuable tools in computing as they make it easier to organize, analyse and search for information.

Even though they are all used to accomplish the same task of ordering data, sorting algorithms may differ significantly from one another in their approach and efficiency at sorting. When measuring an algorithm's performance we typically look at its time and space complexity.

Time and Space Complexity

Time complexity, often considered the most important aspect for determining an algorithm's performance, describes the time required to execute an algorithm as a function of its input size. In other words, time complexity helps us to understand how the algorithm's efficiency scales as the input size increases.

Space complexity, on the other hand, is a measure of memory usage of a sorting algorithm based on the size of the input.

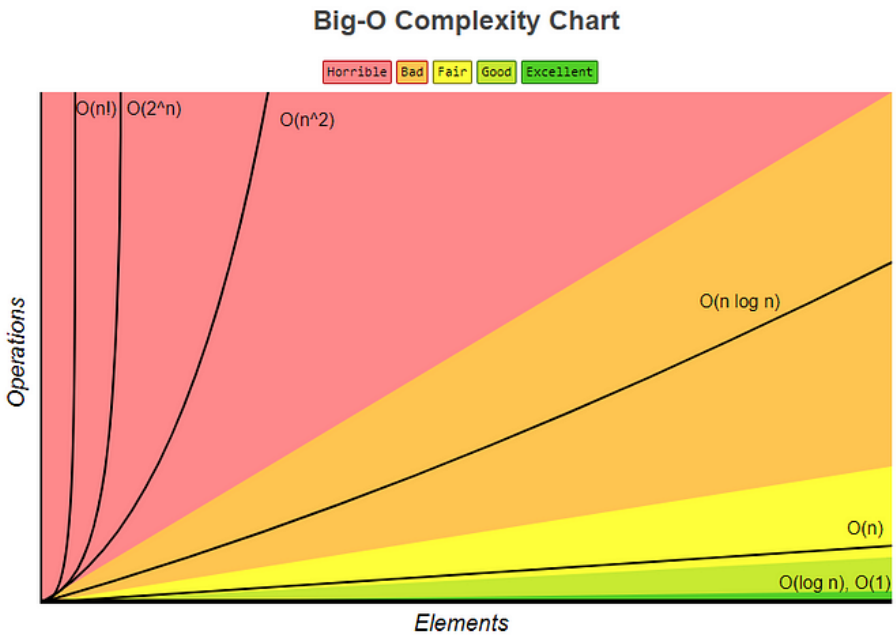
The complexity of an algorithm is typically expressed by the *Big O notation* - a mathematical notation that represents an upper bound on the growth rate of the given algorithm, that is, its worst case performance in terms of the time complexity (running time), and the space complexity (memory usage).

Despite Big O being the most commonly used notation to determine the time and space complexity of a sorting algorithm, since it provides information about algorithm efficiency in the worst case scenario, for a comprehensive analysis of a particular sorting algorithm, *Big Theta (Θ)* and Big Omega (Ω) notations are used to measure average and best case performances.

Common Big O running times, from fastest to slowest:

- O (1) - running time does not depend on the size of the input (constant time)
- O (log n) - runtime increases logarithmically with the input size (log time)
- O (n) - runtime increases linearly with the size of the input
- O (n log n) - linear increase of runtime with the logarithm of the input size
- O (n^2) - quadratic time complexity (running time increases quadratically)
- O (2^n) - exponential time complexity
- O (n!) - factorial time complexity

A graphical representation of growth in time complexity of algorithms (Aguilera, 2018, para.2):



In-place and Stable Sorting Algorithms

The type of algorithm that arranges items within the original data structure is called the **in-place sorting algorithm**. It is often used when space efficiency is the main concern since it doesn't require significantly more extra memory beyond the amount needed for the original input array or list. Some algorithms that fit into this category are Bubble Sort, Insertion Sort, Selection Sort and Quick Sort.

Another category of algorithms is **stable sorting algorithms** implemented in a way that preserves the relative order of equal elements. In other words, if two or more elements have the same value (key) within the input data structure, once sorting is completed, their relative order will stay the same.

As an example of this, we can take a list of students, sorted alphabetically, by their names and then by their grades. If a stable sorting algorithm is used, then the students with the same name will stay in the same order after the list is sorted, even if their grades change (Thakrani, 2023, para.2).

Looking at the example above, we can say that stable sorting algorithms are useful when data holds more than one property, and we want to sort that data by only a specific property while maintaining the order of another.

Common stable sorting algorithms are Bubble Sort, Insertion Sort, Merge Sort, Tim Sort and Counting Sort.

Comparison and Non-comparison Based Sorting Algorithms

The most common approach to sorting elements of a given dataset is by comparing them with one another and placing them at the appropriate position. **Comparison-based sorting algorithms** utilize what is called a *comparator function* to achieve this result. The comparator function is used to determine which of two elements should occur first in the final sorted list (Wikipedia, n.d., para.1) by returning a value indicating their relative order.

Algorithms that use comparison-based sorting are Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort and others.

Although versatile and widely used, comparison-based sorting algorithms have some limitations. For example, they may perform poorly on partially or nearly sorted data sets, resulting in unnecessary comparisons and swaps. Also, some of them are unstable, not preserving the relative order of the equal elements. Another disadvantage is that, in the average and worst case scenarios, comparison-based sorting algorithms cannot perform faster than O(n log n).

Non-comparison based algorithms, on the other hand, can carry out sorting without comparing elements of a data set, but rather by making assumptions about the data to be sorted. Common assumptions include the data types of input set or assumptions about the range of possible values.

By having near linear time and space complexity, non-comparison based sorting algorithms can offer better performance for specific data type, but are not as versatile as comparison based ones.

Common non-comparison based sorting algorithms are Counting Sort, Radix Sort and Bucket Sort.

Which sorting algorithm should we use?

When it comes to the best sorting algorithm, there is no single right answer to that question, but it rather depends on a specific use case.

Is the only thing we care about simplicity of implementation. Go with the likes of Bubble and Selection Sort.

Do we have a large dataset and want it to be sorted considerably fast, but memory usage is what concerns us the most? An in-place, highly efficient sorting algorithm such as Quick Sort would be an excellent choice.

Sorting Algorithms

In this section, I will introduce and discuss five commonly used sorting algorithms, their respective time and space complexities, and provide an illustration of how they work with the following sequence of numbers: 4, 3, 8, 8, 3, 9.

Bubble Sort

One of the simplest sorting algorithms, bubble sort works by repeatedly iterating over a collection of elements, swapping adjacent elements that are out of order until they are sorted.

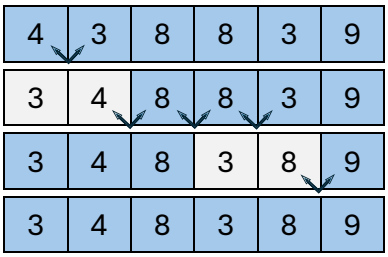
It is *a stable sorting algorithm* since it preserves the relative order of equal elements. Bubble sort is also an *in-place sorting algorithm* because it doesn't require extra memory space - sorting is carried out directly on the original data structure.

Bubble sort time complexity is $O(n^2)$ for the average and the worst case scenarios which makes it very inefficient, especially for large data sets. However, since it is an in-place sorting algorithm, it has a constant space complexity of O(1).

Demonstration of Bubble Sort Algorithm

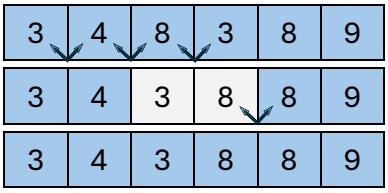
First iteration

Starting from the first element, compare adjacent elements and swap them if necessary.



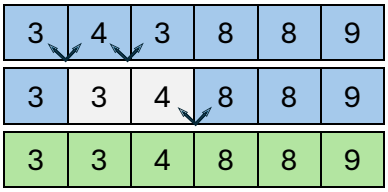
Second iteration

Continue to compare adjacent elements, swapping them if they are out of order. The largest element, after each iteration, is placed at the end (sorted) so there is no need to compare it with the preceding one.



Third iteration

At the end of this iteration, all elements are sorted, but before stopping, the algorithm will run one more pass to ensure that all elements are in the correct place.



Selection Sort

Yet another simple and straightforward sorting algorithm, selection sort starts with the first element of the unsorted portion of the array, compares it to every other element, and then, at the end of each iteration, swaps it if a smaller (or a larger) element is found. This repeats for the remaining unsorted portion until the entire array is sorted.

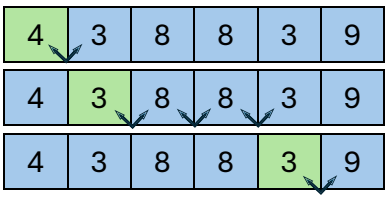
Selection sort is in-place sorting algorithm but is not stable, meaning that after sorting is completed items with equal keys may not preserve their relative order.

Similar to the bubble sort, selection sort has the quadratic time complexity, $O(n^2)$, for worst and average case as there are two nested loops, making it inefficient for large datasets. Interestingly, if array is already sorted, algorithm still needs to go over entire array to find smallest (or largest) element which makes its best case scenario quadratic $\Omega(n^2)$ as well. Due to its in-place sorting nature, space complexity is constant - O(n).

Demonstration of Selection Sort Algorithm

First iteration

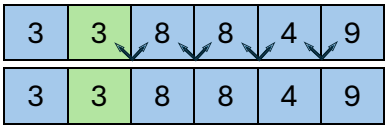
Set the first element as the smallest one and compare it with every other element.



After iteration finishes, swap the first element of the unsorted part with the smallest one.

Second iteration

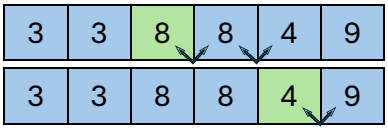
Set the first element of the unsorted portion of the array as the smallest one (3) and compare with the rest of the elements.



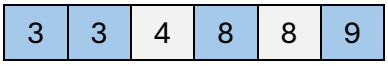
No element was smaller than the first one, so no swaps were made.

Third iteration

Assign the first element of the unsorted part as the smallest one (8) and compare it with the rest of the elements on the right.



Swap the first element of the unsorted portion of the array with the smallest one found.



Even though the array is sorted at this point and no further swaps will be made, the algorithm will keep assigning the next element of the unsorted part of the array as the smallest and compare it with all of the elements on the right until the penultimate element is reached and finally compared with the last one.

Insertion Sort

Insertion sort works in a similar way as players would organise a hand of playing cards (Baeldung, 2024, para.1). It is a simple algorithm that builds a sorted portion of an array or a list by iteratively inserting each element of the unsorted part into its correct position within the sorted portion.

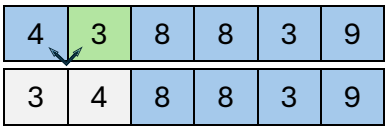
Just like a bubble sort, insertion sort is a stable and in-place sorting algorithm that preserves the relative position of equal elements and requires almost no additional memory space to operate. It's space complexity is O(1).

With time complexity of $O(n^2)$ for the worst and average case, and O(n) for the best case, insertion sort proves good on smaller and nearly sorted datasets.

Demonstration of Insertion Sort Algorithm

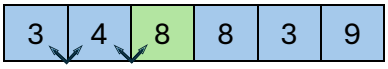
First iteration

The first element of the array is assumed to be sorted, so take the second one and declare it as a key. If key is greater than the first element, move key to the front.



Second iteration

Now when the first two elements are sorted, third element is assigned to the key. Compare current key with the elements to the left.



No swaps were needed since key is the largest element.

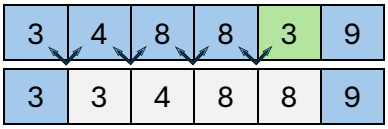
Third iteration

Assign next element of the unsorted part as a key and compare with elements of sorted portion.

No swaps were necessary, assign next element of unsorted part to the key.

Forth iteration

Key (3) is compared to the elements on the left. Since key is smaller, element on the left is moved one step to the right. After all elements that are smaller than the key are moved a step to the right, place the key in front of them.



One more iteration follows, but no elements were moved since the last element was already at the correct position.

Merge Sort

One of the most popular sorting algorithms, merge sort utilises a divide-and-conquer approach. It recursively divides an input array into smaller subarrays, sorting each one individually (sorting is faster on small arrays) and combining them back together to form a compete sorted array.

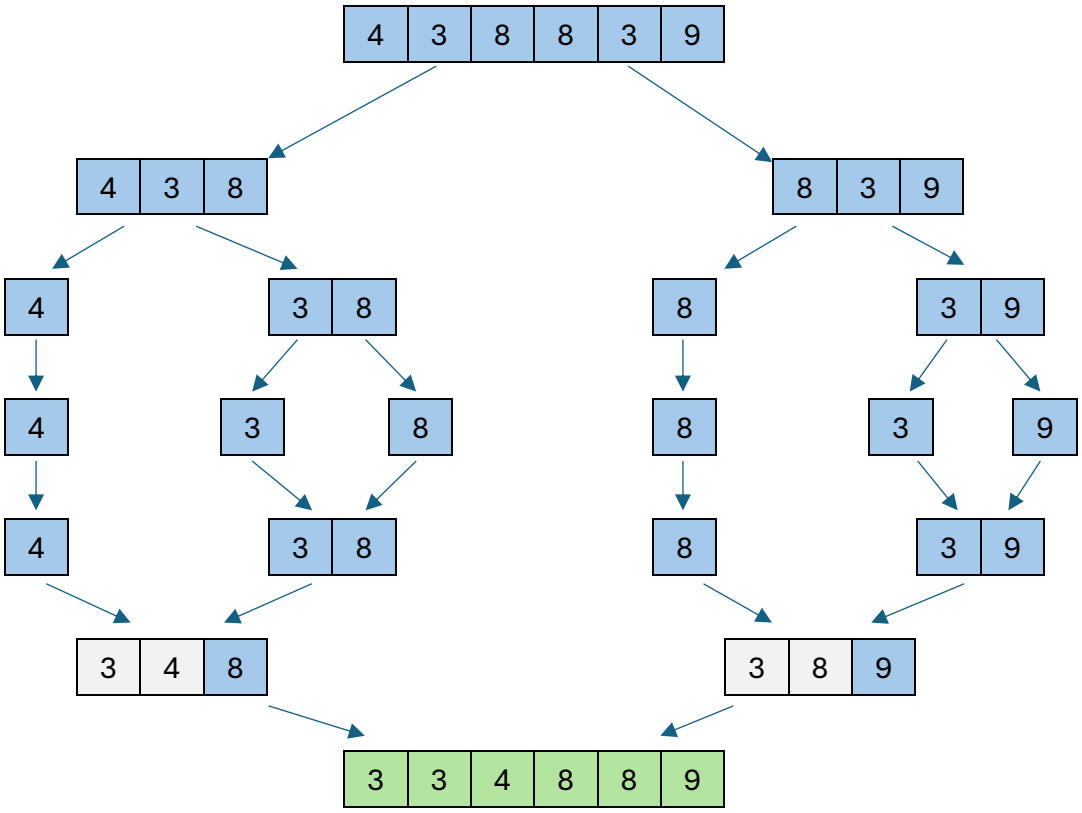
Merge sort is a stable algorithm, it maintains a relative order of equal elements. However, merge sort is not an in-place algorithm, since it requires additional memory to store temporary subarrays during the sorting process. The space complexity of a merge sort algorithm is linear O(n), meaning it linearly grows with the input size.

The time complexity of the merge sort algorithm is O(n log n) for best, average and worst case. It is a highly efficient and versatile algorithm with a consistent time complexity which makes merge sort one of the most popular choices for sorting large datasets.

The drawbacks of merge sort are its space and implementation complexity when compared to some other sorting algorithms. Also, it performs slower on small datasets than simpler algorithms like insertion sort. Another drawback is that merge sort does not possess a mechanism to check whether the sequence of numbers is already sorted, rather it goes through the entire process of sorting and merging, therefore time complexity for the best case is identical to the one for the worst case.

Demonstration of Merge Sort Algorithm

The original array is divided into two halves, and then those halves are further divided until each subarray holds only a single element. The algorithm proceeds to combine those subarrays back together in a sorted order until everything merges back into a complete sorted array.



Counting Sort

The counting sort algorithm works by counting the frequency of unique elements in the array. It stores the count into an additional frequency array by mapping the count as an index of that frequency array, and by using that count, algorithm sorts the data to the output array.

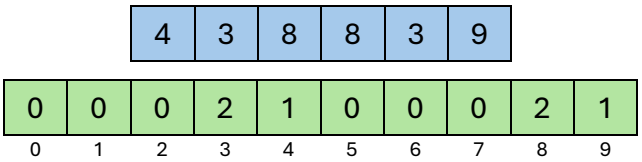
It is a stable sorting algorithm as it preserves the relative order of equal elements. Counting sort requires additional memory for an auxiliary array that stores the count of unique elements, therefore it is not an in-place sorting algorithm.

As a non-comparison based algorithm, counting sort performs sorting in near-linear time if it can make certain assumptions about input, particularly about the range of integer values within the given array. Its efficiency is even more pronounced if the range of integer values is much smaller than the number of elements in the array. Consequently, counting sort has the time complexity of O(n + k) for worst, average and best case where the ‘k’ represents the range of values in the input array.

Demonstration of Counting Sort Algorithm

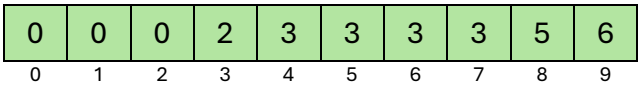
STEP ONE

Create an auxiliary (count) array with the size of the largest element from the original array plus one. Count the frequency of unique elements and store the count by mapping it as an index of the count array.



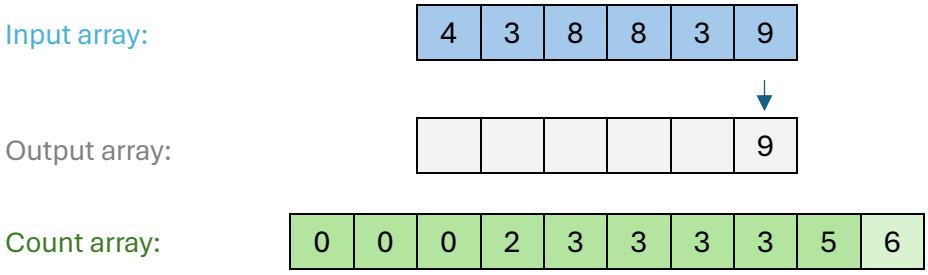
STEP TWO

Calculate the cumulative sum of elements in the count array so we can determine how many elements are less or equal to each of the input elements.



STEP THREE

Iterate over the input array in reverse, and store the element of the input array at the particular index of the output (sorted) array.

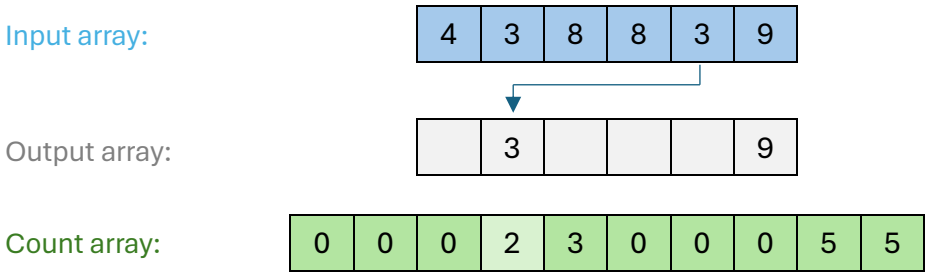


In the step above, the index of the output array is determined by the decremented value of the element at the current index of the count array which, in return, is calculated by the value of an element of the input array. Implementation in code (for i=5):

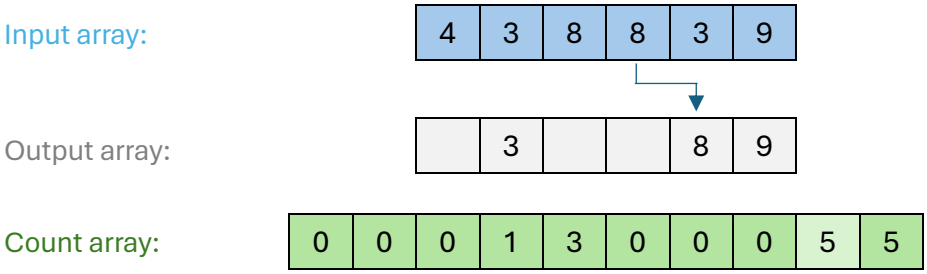
output[count[input[i]] – 1] = input[i] → output[count[9] – 1]] = 9 → output[6 - 1] = 9

meaning algorithm is storing the last element of input array (9) at index 5 of output array. After each iteration we decrement the value at index input[i] of the count array → 6–1= 5.

Next iteration *(i=4)* → output[2 - 1] = 3



Next iteration (i=3) → output[5 - 1] = 8



By iterating and placing elements at the appropriate position within the output array, all of the elements will eventually get sorted:

3	3	4	8	8	9

Once sorting is completed, the algorithm copies all of the elements back into the original input array.

Implementation and Benchmarking

Up to this point, I have discussed and explained different sorting algorithms, how they work, their characteristics, and their efficiency from a theoretical perspective (a priori analysis). I will now conduct a posteriori analysis, which includes the implementation of benchmarking application and sorting algorithms in a programming language, in this case Java, by measuring and comparing their performances, more specifically their time complexity on a number of input sizes.

The application is entirely implemented within a single class consisting of benchmark method, utility methods, and sorting algorithms. To automate the process of benchmarking, two arguments are passed into the benchmark method, one specifying the input size, and the other defining the actual sorting algorithm to be tested. Arrays of input sizes and sorting algorithms are declared within the main method.

The benchmark method, on each sorting algorithm, runs ten times for different input sizes. The current time is taken before and after each benchmark, and then differences of those times are added and stored into a separate variable. Once all ten passes are completed, the benchmark method returns the average time in milliseconds, which is then outputted to the console and stored in a CSV file. The process is repeated for each input size and sorting algorithm.

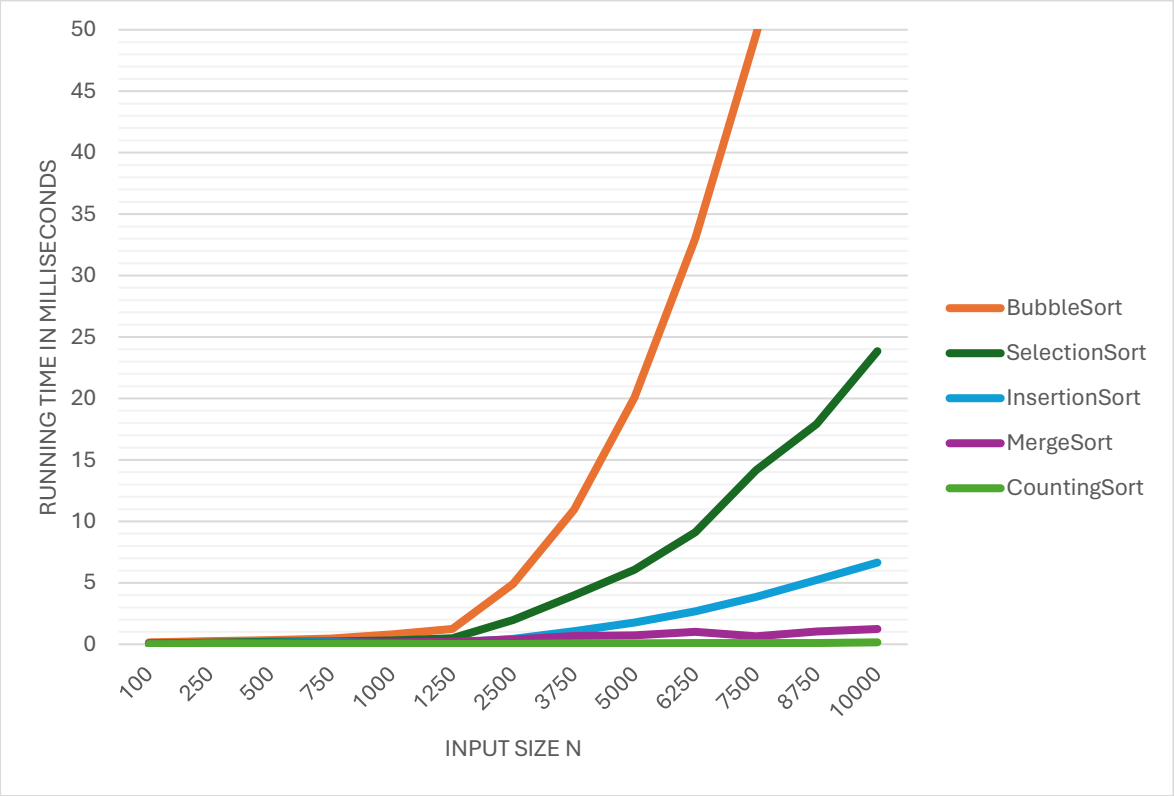
Benchmarking results and graphical representation are provided on the following page.

Benchmark Results

All values are in milliseconds, and represent the average of 10 repeated runs

SIZE	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble	0.153	0.267	0.346	0.472	0.809	1.243	4.917	10.94	20.067	33.019	49.613	68.574	94.916
Selection	0.065	0.13	0.214	0.225	0.304	0.503	1.975	3.983	6.066	9.108	14.169	17.913	23.844
Insertion	0.036	0.107	0.136	0.174	0.107	0.134	0.432	1.054	1.765	2.677	3.851	5.222	6.647
Merge	0.045	0.039	0.074	0.114	0.165	0.206	0.375	0.677	0.716	1.004	0.656	1.039	1.243
Counting	0.011	0.021	0.039	0.053	0.038	0.025	0.033	0.064	0.071	0.089	0.092	0.109	0.158

Graphical Representation of Time Performance of Sorting Algorithms



Looking at the table and graph representing time performance of each algorithm, it can clearly be seen that Bubble Sort evaluates as the most inefficient, while Counting Sort scales the best with the increase of the input size.

Interestingly, even though Bubble, Selection and Insertion Sort have the same time complexity of $O(n^2)$, benchmarking results show that Selection, and especially Insertion sort performs slightly better as the input size increases. Although that difference won't matter much in the long run, since time complexity increases quadratically with the input size for these sorting algorithms, it seems that Bubble Sort, as opposed to Selection and Insertion sort, performs more operations during each iteration, particularly more comparisons and swaps, causing this deviation to manifest in the benchmark results.

On the other hand, the difference in performance between Counting Sort, with a time complexity of O(n +k), and Merge Sort, with a time complexity of O(n log n), despite being visible, is not overly emphasized at first. By looking at the benchmark results table, at smaller input sizes of up to 1000, Counting Sort performs only around twice as well as Merge Sort. But looking further at larger input sizes, the gap between the two becomes very apparent.

The near linear nature of Counting Sort would become much more evident on even bigger datasets, which makes this algorithm, if some assumptions can be made, a very powerful tool for sorting data of large data structures.

References

Aguilera, R. (Sep 7, 2018). *Big O Notation and Sorting Algorithms*. Medium. https://medium.com/@raul.aguilera/big-o-notation-and-sorting-algorithms-2b21115a1c96

Thakrani, S. (Jul 21, 2023). *What are stable sorting and in-place sorting algorithms*. Medium. https://medium.com/@suhailthakrani12/what-are-stable-sorting-algorithms-and-in-place-sorting-algorithms-672820a8e36c

Wikipedia (Jan 4, 2024). *Comparison Sort* https://en.wikipedia.org/wiki/Comparison_sort

Baeldung (Mar 17, 2024). *Insertion Sort in Java* https://www.baeldung.com/java-insertion-sort

Martin, Eric. (Jan 8, 2024). *Formatting Output with printf() in Java*. Baeldung. https://www.baeldung.com/java-printstream-printf

Carr, Dominic. (n.d.). *Source code of benchmarking algorithm* https://vlegalwaymayo.atu.ie/pluginfile.php/1183960/mod_resource/content/0/Main.java

GeeksForGeeks (Apr 8, 2024). *Merge Sort – Data Structure and Algorithms Tutorial* https://www.geeksforgeeks.org/merge-sort/?ref=header_search

Programiz (n.d.). *Counting Sort Algorithm* https://www.programiz.com/dsa/counting-sort