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 2](#_Toc165135053)

[Time and Space Complexity 2](#_Toc165135054)

[A graphical representation of growth in time complexity of algorithms (Aguilera, 2018, para.2): 3](#_Toc165135055)

[In-place and Stable Sorting Algorithms 3](#_Toc165135056)

[Comparison and Non-comparison Based Sorting Algorithms 4](#_Toc165135057)

[Sorting Algorithms 5](#_Toc165135058)

[Bubble Sort 5](#_Toc165135059)

[Selection Sort 6](#_Toc165135060)

[Insertion Sort 7](#_Toc165135061)

[Merge Sort 7](#_Toc165135062)

[Counting Sort 7](#_Toc165135063)

[Implementation and Benchmarking 8](#_Toc165135064)

[Benchmarks results table 8](#_Toc165135065)

[Graphical representation of time performance of sorting algorithms 8](#_Toc165135066)

[References 9](#_Toc165135067)

# 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 (n2) - quadratic time complexity (running time increases quadratically)
* O (2n) - exponential time complexity
* O (n!) - factorial time complexity

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

A diagram of a complexity

Description automatically generated

### 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 data set 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 the 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.

# 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(n2) 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

***First iteration***

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 3 | 8 | 8 | 3 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 8 | 3 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 3 | 8 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 3 | 8 | 9 |

***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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 3 | 8 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 3 | 8 | 8 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 3 | 8 | 8 | 9 |

***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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 3 | 8 | 8 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 4 | 8 | 8 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 4 | 8 | 8 | 9 |

## 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(n2), 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 Ω(n2) as well. Due to its in-place sorting nature, space complexity is constant - O(n).

Demonstration of Selection Sort

***First iteration***

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 3 | 8 | 8 | 3 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 3 | 8 | 8 | 3 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 3 | 8 | 8 | 3 | 9 |

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 8 | 8 | 4 | 9 |

***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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 8 | 8 | 4 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 8 | 8 | 4 | 9 |

*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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 8 | 8 | 4 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 8 | 8 | 4 | 9 |

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

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 4 | 8 | 8 | 9 |

*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(n2) 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

***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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 3 | 8 | 8 | 3 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 8 | 3 | 9 |

***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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 8 | 3 | 9 |

*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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 8 | 3 | 9 |

*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*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 4 | 8 | 8 | 3 | 9 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 3 | 4 | 8 | 8 | 9 |

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

## Merge Sort

## Counting Sort

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4 | 3 | 8 | 8 | 3 | 9 |

# Implementation and Benchmarking

This section will describe the process followed when implementing the application, and present the results of benchmarking. Discuss how the measured performance of the algorithms differed – were the results similar to what you would expect, given the time complexity of each chosen algorithm? In this section you should use both a table and a graph to summarise the results obtained (see samples below).

#### Benchmarks results table

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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SIZE | 100 | 250 | 500 | 750 | 1000 | 2500 | 5000 | 7500 | 10000 |
| Bubble Sort | 0.107 | 0.180 | 0.282 | 0.445 | 0.733 | 4.150 | 18.764 | 49.203 | 90.896 |
| Selection Sort | 0.067 | 0.117 | 0.204 | 0.217 | 0.294 | 1.765 | 6.179 | 13.132 | 22.951 |
| Insertion Sort | 0.035 | 0.111 | 0.138 | 0.236 | 0.141 | 0.706 | 2.110 | 5.030 | 7.119 |
| Merge Sort | 0.086 | 0.031 | 0.074 | 0.125 | 0.182 | 0.363 | 0.913 | 0.537 | 1.240 |
| Counting Sort | 0.010 | 0.019 | 0.038 | 0.054 | 0.047 | 0.036 | 0.075 | 0.096 | 0.211 |

#### Graphical representation of time performance of sorting algorithms

# 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 algorithms 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.* Baeldung.

<https://www.baeldung.com/java-insertion-sort>