SORTING ALGORITHM BENCHMARK REPORT

MODULE: Computational Thinking with Algorithms

AUTHOR: Ante Dujic

This is a sorting algorithm benchmark report, done as a part of a project for Computational Thinking with Algorithms module on GMIT.

There are three parts of this report:

1. INTRODUCTION
2. SORTING ALGORITHMS
3. IMPLEMENTATION AND BENCHMARKING REPORT

1. INTRODUCTION

In the Introduction section it is explained what the definitions of sorting and sorting algorithms are, but also the relevance of concepts such as time and space performance, in-place and out of place sorting, stable and unstable sorting, comparator functions, and what the comparison-based and non-comparison-based sorts algorithms are.

"In mathematics and computer science, an algorithm is a finite sequence of well-defined instructions, typically used to solve a class of specific problems or to perform a computation." (Algorithm - Wikipedia, 2022) There are many different types of algorithms, and the ones analysed in this project are sorting algorithms. According to Collins Dictionary (2022), "**sorting** is the process or operation of ordering items and data according to specific criteria." Sorted items are easier to interpret and to use, so it is not a surprise sorting has been a part of computing since the beginning.

Sorting algorithm is an algorithm that organizes elements of a list so each element in the list is less than or equal to the following one. If there are any duplicate elements in the list, those should appear one after the other, with no other elements in-between. The definition of "*less than*" depends on the elements in the list. If the elements are numbers the ordering will be numerical; if the elements are strings, the ordering could be lexicographical. However, the ordering can also be customized. In general, there is a **comparator** within an algorithm that compares the elements, and the design of a sorting algorithm doesn't depend on it. An example of a comparator function could be: *Function (a, b)* which returns -1 if a < b, 0 if a = b, 1 if a > b (Mannion, lectures [1]).

Sorting algorithms play an important role in computer science since they can reduce the complexity of a problem. "These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more." (Sorting Algorithms Explained with Examples in JavaScript, Python, Java, and C++, 2022)

Numerous algorithms can be designed to solve the same problem. The same applies to sorting algorithms. While there are many different sorting algorithms and their purpose is very similar, they all differ from each other in certain characteristics. Some of the most important characteristics of well-designed sorting algorithms are efficiency, stability, and in-place sorting.

A well-designed algorithm is correct and efficient. The **efficiency** of an algorithm depends on two parameters, time efficiency and space efficiency. *Time efficiency* is defined as the time or number of operations required for the computer to run an algorithm, while *space efficiency* is defined as the amount of memory the computer needs to run an algorithm.

There are two ways to analyse algorithm efficiency, a priory and a posteriori analysis. *A priory analysis* means analysing the efficiency of an algorithm from a theoretical point of view, where we assume all the external factors, such as system architecture or processor speed, are constant. “The relative efficiency of algorithms is analysed by comparing their order of growth” (Mannion, lectures [2]). A priory analysis is a measure of **complexity**. *A posteriori analysis*, on the other hand, is an empirical analysis of an algorithm. The algorithm is implemented and executed on the targeted system and the efficiency of an algorithm is analysed by comparing collected measurements, such as running time and space. A posteriori analysis is a measure of **performance**.

Because the results of testing a certain algorithm on a specific system don’t give a full picture of its performance, the algorithmic complexity is analysed mathematically. **Algorithmic complexity** is a measure of how long an algorithm would take to perform with an input of size n. The algorithm should compute the result within a finite and practical running time, even for large size n (dineshpathak, 2022). In general, algorithmic complexity falls into one of various classes. The ones relevant for the sorting algorithms are:

- **O(1)** – *Constant:* Takes the same number of steps regardless of the input
- **O(n)** – *Linear:* Takes proportionally longer to complete as the input grows
- **O(n^2)** – *Quadratic:* Takes square of the input size time to complete
- **O(log n)** – *Logarithmic:* The number of operations increases by one each time input is doubled
- **O(n log n)** – *Linearithmic:* Takes n times log n to perform

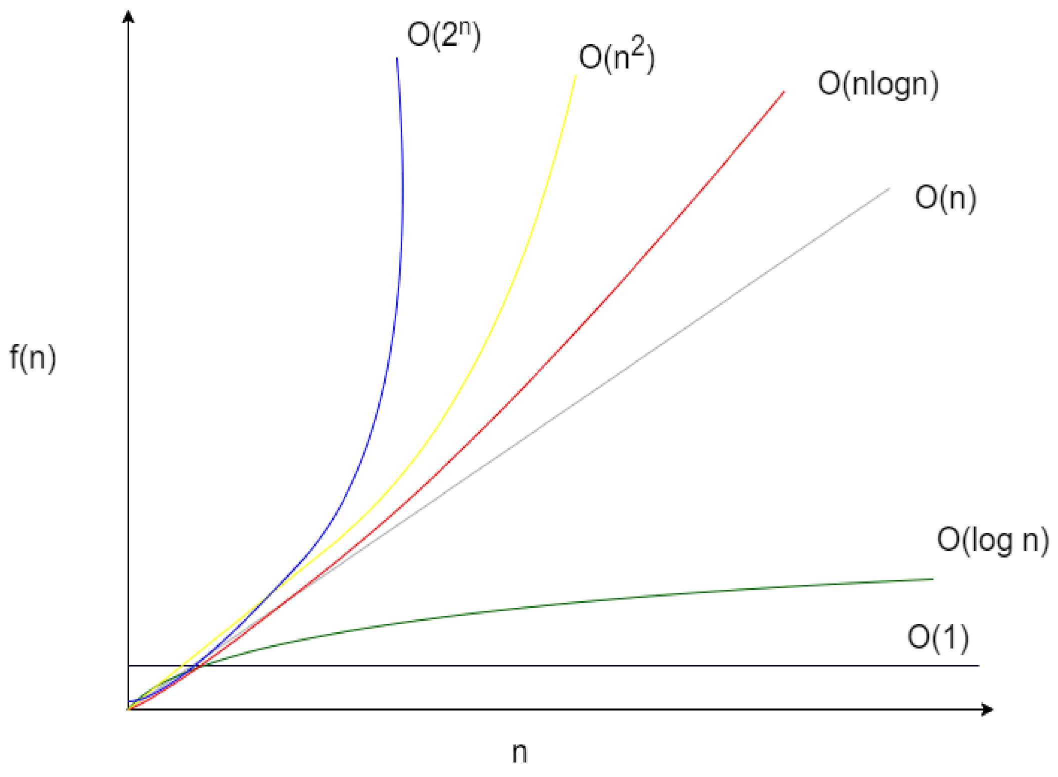


Image 1. Algorithmic complexity (Algorithm Time Complexity, 2022)

In computer science, the resource usage of a given algorithm is expressed through worst, average, and best cases. The *worst case* defines a class in which an algorithm performs the maximum number of steps. It is often the easiest to analyse. The maximum complexity or the upper bound is expressed with Big O notation. It is a typically used notation to represent algorithmic complexity. An *average case* defines a class where the algorithm performs an average number of steps. It defines the expectation an average algorithm user should have. *Θ(theta) notation* is used for analysing the average-case complexity of an algorithm. It represents both the maximum and minimum complexity of an algorithm. The *best case* is the case in which an algorithm makes the minimum number of steps. *Ω(omega) notation* represents the lower bound on complexity and provides the best-case scenario. The best case in reality rarely occurs.

The resource being considered to determine the case categorisation is usually running time, but could also be memory or storage requirements or some other resources.

Another important characteristic on which we can distinguish different sorting algorithms is the **memory requirements**. An algorithm is considered an in-place or an out-of-place algorithm, based on the explicit storage assigned by the algorithm. (In-place vs out-of-place algorithms | Techie Delight, 2022) An *in-place* sorting algorithm uses constant space to create the output. The order of the elements in the list is modified within the same, original list. An algorithm that is not in-place is called a not-in-place or *out-of-place* algorithm. Unlike an in-place algorithm, the extra space used by an out-of-place algorithm depends on the input size. (In-place vs out-of-place algorithms | Techie Delight, 2022)

Sorting algorithms also differ from each other based on stability. The **stability** of a sorting algorithm is defined by how the algorithm sorts the identical elements. *Stable* sorting algorithms keep the relative order of identical elements, while *unstable* sorting algorithms don't. In other words, stable sorting preserves the position of two equal elements relative to each other. Unstable sorting algorithms might maintain relative ordering, but they also might not. Stability is primarily important if we have key value pairs with duplicate keys possible (e.g., people's names as keys and their details as values) and we wish to sort these objects by keys. (Sorting Terminology - GeeksforGeeks, 2022)

As it is established above, there are many different sorting algorithms, but the ones widely applicable are the comparison sorts. A **comparison sorts** algorithm is the algorithm that uses comparison operations to determine which elements of a list should come first in the final, sorted list. Those algorithms use the comparator function mentioned before. Algorithms that sort the lists by comparing elements can't do better than n log n performance in the average or worst cases (Mannion, lectures [2]). Some of the most common comparison-based algorithms are Bubble Sort, Insertion Sort, Selection Sort, and Merge Sort. A sorting algorithm that sorts the elements in the list without comparing is called a **non-comparison sort**. They work by making certain assumptions about the data. Non-comparison-based algorithms are usually faster than comparison-based ones. They can achieve n linear time in the best case and also don't require a comparator. Non-comparison-based algorithms are, however, less flexible. Some examples of non-comparison-based algorithms are Bucket Sort, Counting Sort, and Radix Sort. There are also *hybrid algorithms* that allow minimizing the weaknesses of two or more algorithms.

It is important to note that there is no single algorithm that is a good choice for every possible input. So, even though certain algorithms can be most efficient to solve a given problem, there might be another algorithm that can outperform it for a particular instance of the same problem.

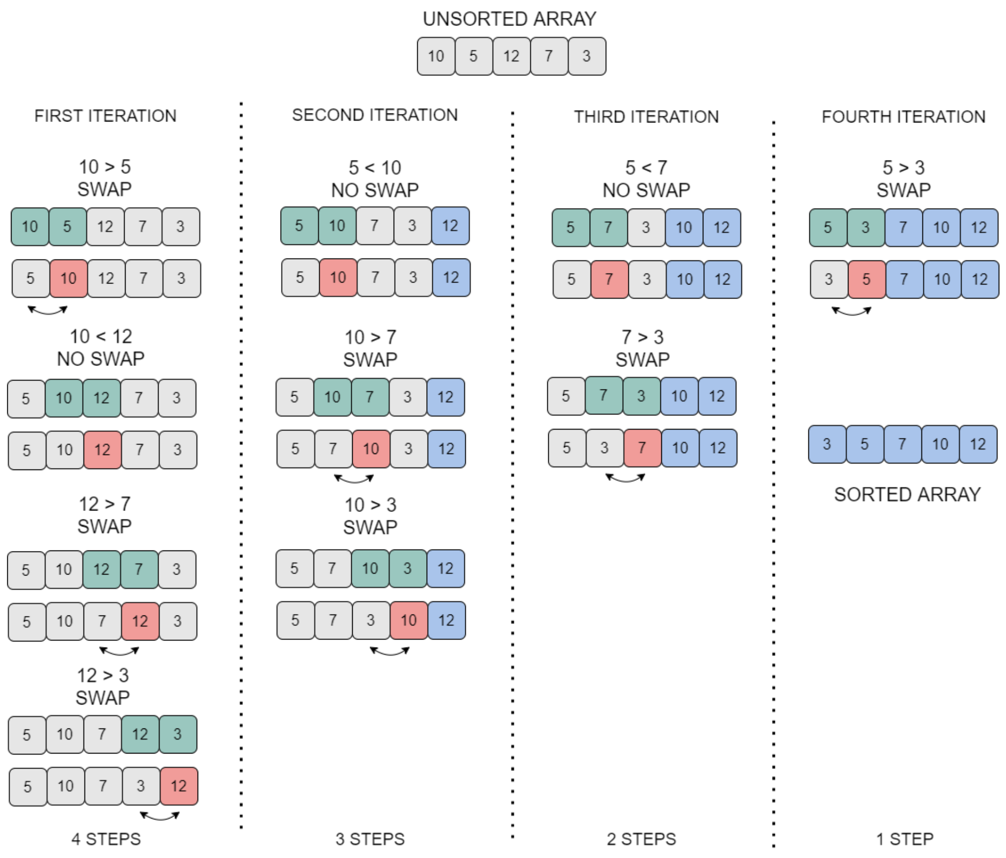
2. SORTING ALGORITHMS

This section gives an overview of 5 different sorting algorithms:

- Two simple comparison-based sorts
    - **Bubble Sort**
    - **Selection Sort**
- Two efficient comparison-based sorts
    - **Merge Sort**
    - **Quick Sort**
- A non-comparison sorts
    - **Counting Sort**

2.1. BUBBLE SORT ALGORITHM

Bubble sort is a simple comparison-based algorithm that iterates through the list comparing the two adjacent elements of the list and swapping ones that are out of order. If the first element is lower than the second one, there is no swapping. On the other hand, if the first element is higher than the second one, they swap the position. This process repeats until all the elements get compared and swapped accordingly. The number of iterations is equal to the number of elements in the list minus one.



There are 5 elements in the array [10, 5, 12, 7, 3], hence algorithm iterates four times through the array. The process starts with the first element of the array. The current element is 10. The algorithm compares the current element with the following one. Since 10 > 5 it swaps them. It continues the comparisons for the following elements. Next, it compares 10 and 12. Since 10 < 12 there is no swap, and the current element is changed to 12. It continues comparing the new current element 12 with the following element, following the same process. The first iteration finishes when the highest element reaches the end of the array. 12 is now in its final position and the second iteration starts. The algorithm does the same process in the second, third, and the fourth iteration, until all the elements are in their final positions; the array is sorted.

Bubble sort is a very slow algorithm and thus recommended to use on rather small data sets. Its worst case time complexity is n^2. The worst case occurs if the list to be sorted is in descending order. The number of iterations is n-1, there are n-1 swaps in the first iteration, n-2 in the second one, and so on until the list is sorted. The average case occurs if the elements of a list are in random order. The total number of swaps averages around $n^2/2$, so the average case complexity is also n^2. The advantage of a bubble sort algorithm is that it is possible to design an algorithm that can determine if the list is already sorted. If this is the case, the algorithm will perform the minimum number of steps. It is still necessary to iterate through the list so the best case is expressed as *n*.

WORST CASE O(n^2)	AVARAGE CASE Θ(n^2)	BEST CASE Ω (n)
[5, 4, 3, 2, 1]	[3, 5, 2, 1, 4]	[1, 2, 3, 4, 5]
[4, 5, 3, 2, 1] [4, 3, 5, 2, 1] [4, 3, 2, 5, 1] [4, 3, 2, 1, **5**]	[3, 2, 5, 1, 4] [3, 2, 1, 5, 4] [3, 2, 1, 4, **5**]	[**1**, **2**, **3**, **4**, **5**] NO SWAPS
[3, 4, 2, 1, **5**] [3, 2, 4, 1, **5**] [3, 2, 1, **4**, **5**]	[2, 3, 1, 4, **5**] [2, 1, 3, **4**, **5**]	
[2, 3, 1, **4**, **5**] [2, 1, **3**, **4**, **5**]	[**1**, **2**, **3**, **4**, **5**]	
[**1**, **2**, **3**, **4**, **5**]	NO SWAPS	

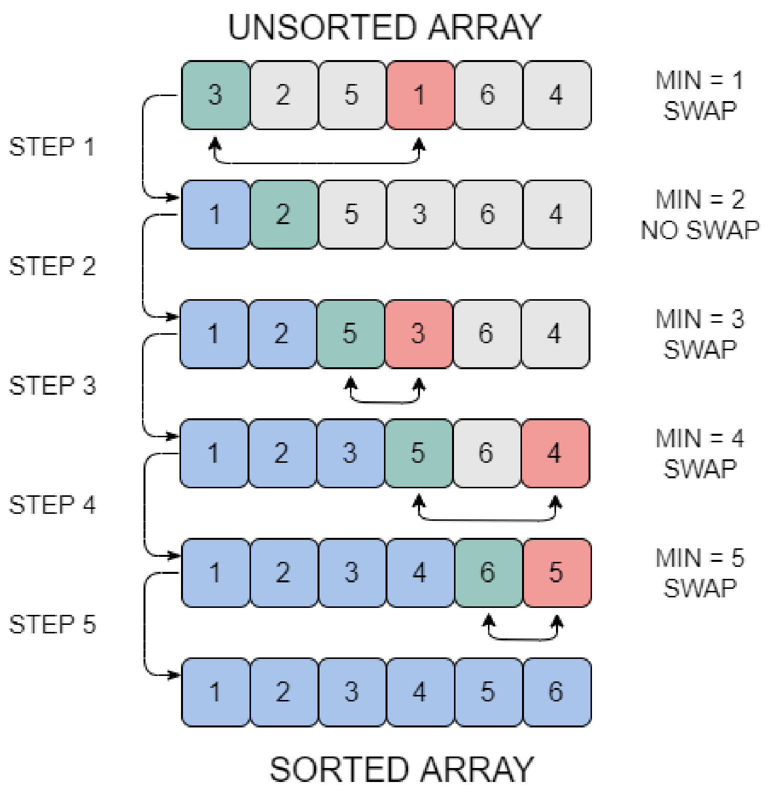
* Number of iterations is always the same, but the number of swaps changes in different cases

** Some variations of the algorithm can recognize an already sorted list

Bubble sort is very efficient in terms of space complexity. It performs the swaps in-place, meaning it uses a constant amount of additional memory to sort. Therefore, the space complexity of a bubble sort algorithm is *O(1).* Bubble sort is a stable sorting algorithm.

2.2. SELECTION SORT ALGORITHM

Selection sort is another simple comparison-based algorithm. It is also an in-place algorithm, but unlike the bubble sort, the selection sort is unstable. The selection sort improves on the bubble sort by making only one exchange for every pass through the list (The Selection Sort, 2022), but is still very impractical for the large input size. The selection sort algorithm divides the list to be sorted into two parts, a subarray that is sorted on the left and a subarray that is unsorted on the right. The algorithm then iterates through the list and selects the smallest element (if the order is ascending), which gets swapped with the left most element. This element now becomes part of the sorted list. The process repeats and is equal to the number of elements in the list minus one.



The array [3, 2, 5, 1, 6, 4] has 6 elements, meaning there are five steps the algorithm takes to sort this array. It starts at the first element of the array, 3. The algorithm looks for the lowest number which then gets swapped with 3. It does it by first storing 3 as a temporary minimum, and then compares every next element to it. If the element compared to 3 is smaller than that number gets stored as a new minimum. This process is done when the algorithm compared the last element with the current minimum. The minimum in the first steps is 3, then 2, then 1, which is the final minimum. It then swaps the position with the first element; 3 and 1 swap. 1 is now in its final position and the same process continues in the following steps, until all the elements are in their final positions; the array is sorted.

The time complexity of the selection sort algorithm is n^2 in the worst, best and average case. The worst-case complexity occurs if the list we want to sort in ascending order is descending, and the other way around. There is a swap done at each step in the worst case. The best case occurs if the list is already sorted. No swaps are being done at each step in this case. If the elements of the list are randomly positioned then the complexity is average. It is necessary to scan all the elements to find the minimum one, regardless of the data in the array. In the first iteration there are n-1 comparisons done, in the second one n-2, and so on until the last comparison gets done.

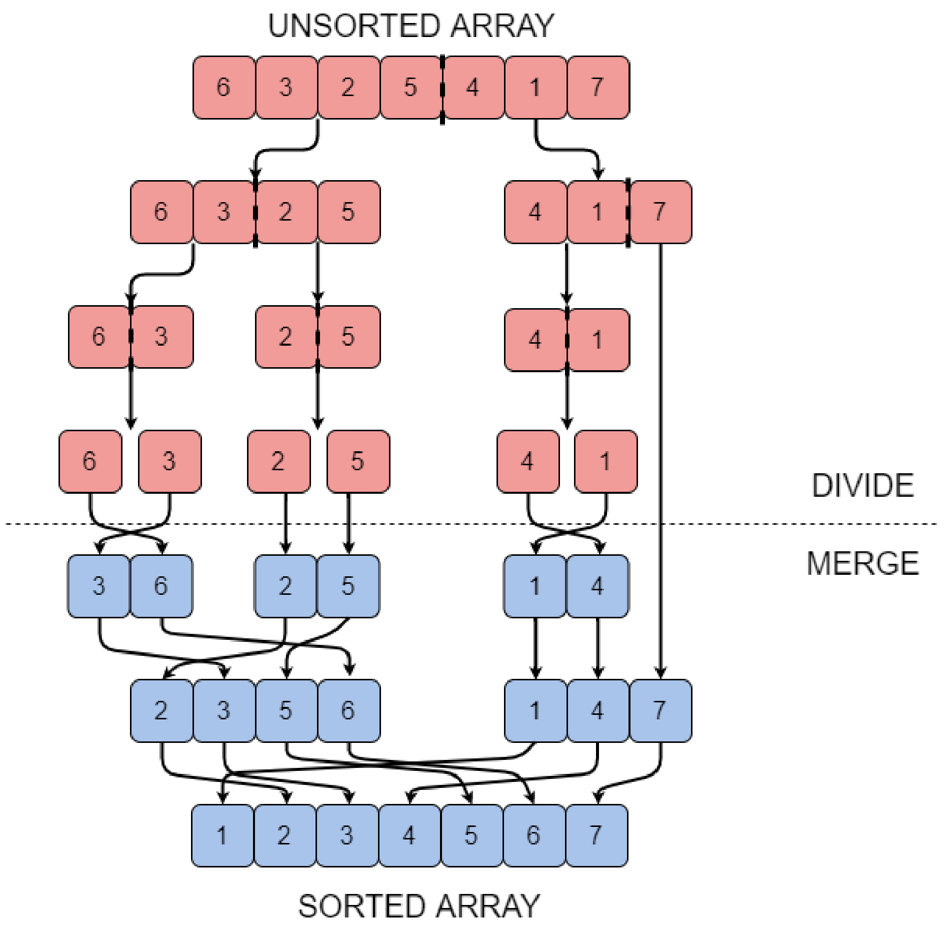
WORST CASE $O(n^2)$	AVARAGE CASE $\Theta(n^2)$	BEST CASE $\Omega(n^2)$
[5, 4, 3, 2, 1]	[3, 2, 4, 1, 5]	[1, 2, 3, 4, 5]
5, 4, 3, 2 > 1 [**1**, 4, 3, 2, 5]	3, 2, 4 > 1 1 < 5 [**1**, 2, 4, 3, 5]	1 < 2, 3 ,4 ,5 [**1**, 2, 3, 4, 5]
4, 3 > 2 2 < 5 [**1**, **2**, 3, 4, 5]	2 < 4, 3, 5 [**1**, **2**, 4, 3, 5]	2 < 3, 4, 5 [**1**, **2**, 3, 4, 5]
3 < 4, 5 [**1**, **2**, **3**, 4, 5]	4 > 3 3 < 5 [**1**, **2**, **3**, 4, 5]	3 < 4, 5 [**1**, **2**, **3**, 4, 5]
4 < 5 [**1**, **2**, **3**, **4**, **5**]	4 < 5 [**1**, **2**, **3**, **4**, **5**]	4 < 5 [**1**, **2**, **3**, **4**, **5**]

* Number of the compares is always the same, but the number of swaps changes in different cases

Same as bubble sort, the space complexity of selection sort is also *O(1).* This is because it uses constant extra memory for its operations, such as 2 variables for swapping and 1 variable to keep track of the smallest element.

2.3. MERGE SORT ALGORITHM

Merge sort algorithm is an efficient comparison-based sorting algorithm. It works by recursively splitting the input list into two or more sub-lists of the same or related type, until it cannot be split any further. Those elements are then merged, sorting them at the same time. This approach is called divide and conquer. To simplify, merge sort first divides the array into equal halves and then combines them in a sorted manner (A Simplified Explanation of Merge Sort, 2022). Merge sort is a stable sort algorithm.



The array [6, 3, 2, 5, 4, 1, 7] has seven elements and it gets divided into two shorter arrays of almost equal length, [6, 3, 2, 5] with four elements and [4, 1, 6] with three elements. The same algorithm is then recursively called on the two smaller arrays, doing the same. [6, 3, 2, 5] gets divided into [6, 3] and [2, 5], and the other array [4, 1, 7] is divided into [4, 1] and [7]. The same process continues until all the arrays are broken into single elements. At the end of the dividing process, there are the following individual elements: 3, 6, 2, 5, 4, 1, and 7. The merging process starts here. The algorithm first compares the individual elements and then merges them accordingly (lower ones of the two compared ones going to the left position) forming the final sorted array.

Merge sort algorithm is one of the most efficient algorithms. As such, it is particularly good for sorting data with slow access times, such as data that cannot be held in internal memory (RAM) or are stored in linked lists. (Mannion, lectures [4]). Its worst, best and average cases are all very similar. The overall time complexity of a merge sort algorithm is *n log n*. The worst case complexity occurs when the list to be sorted is in reverse order. If the elements of the list are in random order, then the time complexity is average. The best case occurs if the list is already sorted..

WORST CASE O(n log n)	AVARAGE CASE Θ(n log n)	BEST CASE Ω(n log n)
[7, 6, 5, 4, 3, 2, 1]	[7, 3, 2, 6, 4, 1, 5]	[1, 2, 3, 4, 5, 6, 7]
[7, 6, 5, 4] | [3, 2, 1]	[7, 3, 2, 6] | [4, 1, 5]	[1, 2, 3, 4] | [5, 6, 7]
[7, 6] | [5, 4] | [3, 2] | [1]	[7, 3] | [2, 6] | [4, 1] | [5]	[1, 2] | [3, 4] | [5, 6] | [7]
[7] | [6] | [5] | [4] | [3] | [2] | [1]	[7] | [3] | [2] | [6] | [4] | [1] | [5]	[1] | [2] | [3] | [4] | [5] | [6] | [7]
[6, 7] | [4, 5] | [2, 3] | [1]	[3, 7] | [2, 6] | [4, 1] | [5]	[1, 2] | [3, 4] | [5, 6] | [7]
[4, 5, 6, 7] | [1, 2, 3]	[2, 3, 6, 7] | [1, 4, 5]	[1, 2, 3, 4] | [5, 6, 7]
[1, 2, 3, 4, 5, 6, 7]	**[1, 2, 3, 4, 5, 6, 7]**	**[1, 2, 3, 4, 5, 6, 7]**

* Number of steps is always the same, but the number of compares changes in different cases

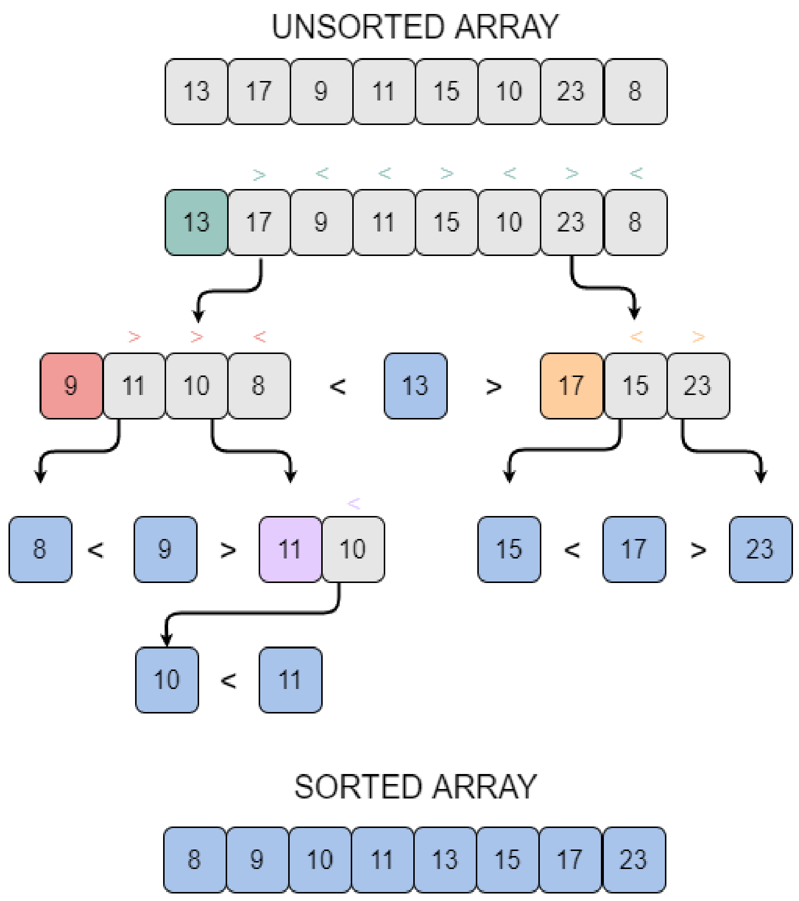
Merge sort algorithm takes a lot of space. It requires additional memory for swapping; therefore, its space complexity is *O(n).*

2.4. QUICK SORT

Quick sort algorithm is another efficient comparison-based sorting algorithm. It uses recursion and the divide and conquer approach, same as the merge sort algorithm. Quick sort works by picking an element of a list as a pivot and divides the list into two sub-lists; one list gets filled with the elements that have smaller values than the pivot, and the other list with the elements with higher values. The two sub-lists are then further divided using the same approach. This process continues until there is a single element in each sub-list. There are many different versions depending on which element of a list is set as a pivot:

1. Set the first element as a pivot
2. Set the last element as a pivot
3. Set random element as a pivot
4. Set median as a pivot

Efficient implementations of a quick sort are not stable sort because swapping is done based on the position of the pivot, ignoring the original positions. However, stable versions do exist. Because it uses extra memory only for storing recursive function calls but not for manipulating the input, it is also considered an in-place sorting algorithm. (QuickSort - GeeksforGeeks, 2022)



In this instance, the first element is selected as the pivot. In the array [13, 17, 9, 11, 15, 10, 23, 8], 13 gets selected as the pivot. This array is then partitioned into two subarrays, in a fashion so that all elements lower than 13 get placed in one, and the ones higher than 13 in the other array. 13 is now in its final position while elements 9, 11, 10, and 8 form one sub-array and 17, 25, and 23 form the other one. The same algorithm is then called on those two sub-arrays. The process continues until all the sub-arrays become individual elements, which are at this stage all at their final positions, forming the sorted array.

Quick sort algorithm is one of the fastest known sorting algorithms. It runs in *n log n* on average. This occurs when the list elements are neither properly ascending nor descending. The worst case rarely occurs with a quick sorting algorithm. It is because it can be implemented in different ways, just by picking a different element for a pivot. The worst case is the case when the selected pivot is either the greatest or the smallest element of the list. This leads to the situation where the pivot lies at the extreme end of the sorted array. One sub-array is always empty and the other sub-array contains n - 1 elements. The result is quicksort being called only on this sub-array (QuickSort (With Code in Python/C++/Java/C), 2022). The worst-case complexity is n^2. When the algorithm chooses the middle element or near the middle one as a pivot, then the best-case occurs. The best-case complexity is *n log n*. If compared to the merge sort algorithm, the quick sort is more efficient when sorting smaller array size, while merge sort is more efficient and works faster in case of larger array sizes (Quick Sort vs Merge Sort - GeeksforGeeks, 2018).

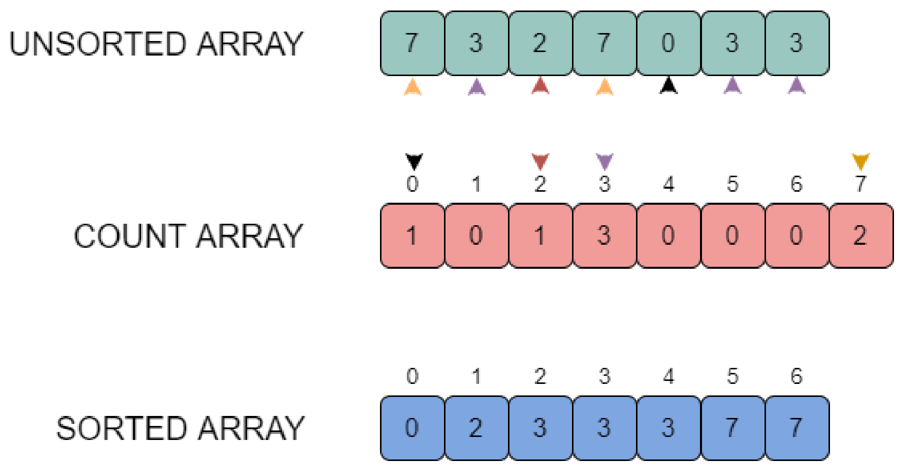
WORST CASE $O(n^2)$	AVARAGE CASE $\Theta(n \log n)$	BEST CASE $\Omega(n \log n)$
[7, 6, 5, 4, 3, 2, 1]	[5, 3, 7, 2, 6, 1, 4]	[4, 1, 3, 2, 6, 5, 7]
[6, 5, 4, 3, 2, 1] **7** [ ]	[3, 2, 1, 4] **5** [7, 6]	[1, 3, 2] **4** [6, 5, 7]
[5, 4, 3, 2, 1] **6** [ ]	[2, 1] **3** [4] [7] [6]	[2] **1** [3] [5] **6** [7]
[4, 3, 2, 1] **5** [ ]	[2] [1]	**[1, 2, 3, 4, 5, 6, 7]**
[3, 2, 1] **4** [ ]	**[1, 2, 3, 4, 5, 6, 7]**	
[2, 1] **3** [ ]		
[1] **2** [ ]		
[1, 2, 3, 4, 5, 6, 7]		

* Selecting different pivot would improve the worst case

The out-of-place version of quick sort has a space complexity of *O(n).* The in-place version of quick sort uses *O(log n)* space for working memory (Quicksort, 2022).

2.5. COUNTING SORT

Counting sort is a non-comparison sorting algorithm, which means it doesn't perform the sorting by comparing the elements. It does it by iterating through the input data, checking how many times each item occurs, and using those counts to calculate an item's index in the final, sorted array. While comparison sorts compare all elements against each other without making any assumptions, counting sort makes assumptions about the data. For example, it assumes that values are going to be in a certain range. Some other assumptions counting sort makes are input data will be all real numbers (Counting Sort - GeeksforGeeks, 2022). Counting sort algorithm uses a temporary array to do the sorting, hence it belongs to the out-of-place types of algorithms. If implemented in the correct way it is stable. Counting sort algorithm is not a general-purpose algorithm, it is most effective when there are multiple smaller integers in a list. Because of its characteristics, counting sort algorithm can be used as a subprogram in other sorting algorithms, such as radix sort which can be used for sorting numbers having a large range (Time & Space Complexity of Counting Sort, 2022).



The array [7, 3, 2, 7, 0, 3, 3] has eight elements in the range 0 to 7. Because of the element range, there is a count array initiated with 8 elements (because the maximum index is 7). Values at each index in the count array are initially set to 0. The algorithm then checks if there are any values in the unsorted array that are the same as the index value of the count array, and counts their occurrences. In this example, there are 1 x 0's, 0 x 1's , 1 x 2's, 3 x 3's, 0 x 4's, 0 x 5's, 0 x 6's and 2 x 7's. Finally, the algorithm constructs an output array [0, 2, 3, 3, 3, 7, 7] using mentioned information.

Counting sort works the same way regardless if the array to be sorted is already sorted, reversed, or if the elements are in random order. The time complexity for all such cases is the same, *n+k*. n is the number of elements in the list and k is the range of elements in the list. If all the elements of the list are of the same range, then the best-case complexity occurs. Counting how many times each element occurs in the list is constant and then finding the index value takes n time, which makes the total time complexity (almost) linear. The average time complexity occurs if the elements in the list are random. Worst-case time complexity occurs if the largest element of the list is much larger than the other elements. The worst case happens when the range k is large.

Space complexity depends on the size of the input. As the range of the list gets bigger the space complexity becomes larger. It is expressed as *O(k)* and this is the reason the counting sort algorithm is at its best when the list range is small.

SORTING ALGORITHM CHARACTERISTICS OVERVIEW

ALGORITHM	BEST CASE	WORST CASE	AVARAGE CASE	SPACE COMPLEXITY	STABLE	IN-PLACE
Bubble Sort	n	n^2	n^2	1	Yes	Yes
Selection Sort	n^2	n^2	n^2	1	No	Yes
Merge Sort	n log n	n log n	n log n	n	Yes	No*
Quick Sort	n log n	n^2	n log n	n*	No*	Yes*
Counting Sort	n + k	n + k	n + k	n + k	Yes	No

* Standard version

3. IMPLEMENTATION AND BENCHMARKING

As previously mentioned, there are two ways to analyse algorithm efficiency, a priory and a posteriori analysis. Benchmarking is a posteriori analysis, meaning it is an empirical method used to compare the relative performances of the algorithms. The performance of an algorithm depends on the various hardware and software characteristics. For this reason, there are multiple statistical runs done on the same machine. This provides the best possible representation of the algorithm performance for an average user.

Benchmarking for this project is done using an application created in Python – *benchmark.py.* The Five sorting algorithms introduced above are contained in this benchmark. Each algorithm code is written in a separate script (*bubbleSort.py, selectionSort.py, mergeSort.py, quicksort.py,* and *countingSort.py*) which is imported into the main application. The code included in each is shown below.

- bubbleSort.py (Python Program for Bubble Sort - GeeksforGeeks, 2022)

```
def bubbleSort(arr):

    # Defining the size of the array
    n = len(arr)

    # Loop to access each array element
    for i in range(n-1):

        # Loop to compare array elements
        for j in range(0, n-i-1):

            # Compare the two adjacent elements
            if arr[j] > arr[j + 1] :

                # Swap the elements if the above comparison is true
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

- selectionSort.py (Selection Sort in Python, 2022)

```
def selectionSort(arr):

    # Defining the size of the array
    n = len(arr)

    # Loop to access each array element
    for i in range(n-1):

        # Set the first element to be the minimum
            #other elements will be compared to it
        min_index = i

        # Loop throught the remaining array elements
        for j in range(i+1, n):

            # If the element at j is lower then the minimum
            if arr[j] < arr[min_index]:
                # Assign it as a new minimum
                min_index = j

        # After finding the lowest element, swap with the first element
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

- mergeSort.py (The Merge Sort, 2022)

```
def mergeSort(arr):
    if len(arr)>1:

        # Find the middle of the array
        mid = len(arr)//2

        # Create the two halves
            # Left from the mid point
        lefthalf = arr[:mid].copy()
            # Right from the mid point
        righthalf = arr[mid:].copy()

        # Calling the function on each half (recursion)
        mergeSort(lefthalf)
        mergeSort(righthalf)

        # SORTING AND MERGING TWO HALVES

        i=0  # Initial index of the left array
        j=0  # Initial index of the right array
        k=0  # Initial index of the merged array

        # Loop until the left and right array are empty
        while i < len(lefthalf) and j < len(righthalf):

            # Compare the elements of the array
                # If the left element is smaller
            if lefthalf[i] <= righthalf[j]:
                # place this element in the merged array on index k
                arr[k]=lefthalf[i]
                i=i+1   # move to next index (left array)

                # if the right element is smaller
            else:
                # place this element in the merged array on index k
                arr[k]=righthalf[j]
                j=j+1   # move to next index (right array)
            k=k+1 # move to next index (merged array)

        # No elements left in right array
        # Loop until the left array is empty
        while i < len(lefthalf):
            # Place remaining elements of left array in merged array
            arr[k]=lefthalf[i]
            i=i+1   # move to next index (left array)
            k=k+1   # move to next index (merged array)

        # No elements left in left array
        # Place remaining elements of right array in merged array
        while j < len(righthalf):
            arr[k]=righthalf[j]
            j=j+1   # move to next index (right array)
            k=k+1   # move to next index (merged array)

    return (arr)
```

- quickSort.py (The Quick Sort, 2022)

```
def quickSort(arr):
    # calling the recursive function
   quickSortHelper(arr,0,len(arr)-1)

# Recursive function
def quickSortHelper(arr,first,last):
    # Base case
        # if the array length < 1, list is already sorted
   if first<last:

       # Defining a split point
       splitpoint = partition(arr,first,last)

       # Call the recursive function on the sub-arrays
       quickSortHelper(arr,first,splitpoint-1)
       quickSortHelper(arr,splitpoint+1,last)

# Function for partitioning the array around the pivot
def partition(arr,first,last):
   # Defining pivot (first array element)
   pivotvalue = arr[first]
   # Left partition marker is the first element remaining
   leftmark = first+1
   # Right partition marker is the last element remaining
   rightmark = last

   done = False
   while not done:
       # Move the left marker to the right until the element > pivot
       while leftmark <= rightmark and arr[leftmark] <= pivotvalue:
           leftmark = leftmark + 1
       # Move the right marker left until the element < pivot
       while arr[rightmark] >= pivotvalue and rightmark >= leftmark:
           rightmark = rightmark -1
       # Stop if the second marker is less then the first one
       if rightmark < leftmark:
           done = True
       else:
       # Swap the values that are out of place
           temp = arr[leftmark]
           arr[leftmark] = arr[rightmark]
           arr[rightmark] = temp

   # Swap the values
   temp = arr[first]
   arr[first] = arr[rightmark]
   arr[rightmark] = temp

   return rightmark
```

- countingSort.py (Counting Sort in Python, 2022)

```
def countingSort(arr):
    # Finding the maximum element in the array
    maxElement= max(arr)

    # Set the lenght of the temp array to the array lenght + 1
    countArrayLength = maxElement+1

    # Initialize the counting array
    countArray = [0] * countArrayLength

    # Iterate throught the array
    for el in arr:
        # Set the count for every element by 1
        countArray[el] += 1

    # Loop the range of the count array
    for i in range(1, countArrayLength):
        # For each element in the countArray,
        # sum up its value with the value of the previous
        # element, and then store that value
        # as the value of the current element
        countArray[i] += countArray[i-1]

    # Calculate element position based on the count array values
    outputArray = [0] * len(arr)
    i = len(arr) - 1
    while i >= 0:
        currentEl = arr[i]
        countArray[currentEl] -= 1
        newPosition = countArray[currentEl]
        outputArray[newPosition] = currentEl
        i -= 1

    return outputArray
```

For each algorithm in the application:

- There are different input (n) sizes in the range 100 – 10 000
- There are 10 tests (runs)
- The running time for each run is measured in ms
  - The average is calculated for each input size
- Input in range 0 – 100 is randomly generated for each run

The end result is printed out on the terminal showing the average running time for each input size for each sorting algorithm. Same data is also plotted to give a visual representation for the easier comparison and analysis. More details on how this was implemented are in the code comments in *benchmark.py.*

AVARAGE RUNNING TIME (10 RUNS) IN MILLISECONDS													
Input size	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	2.291	11.761	45.459	100.781	186.019	493.883	1603.147	2668.367	4644.961	7167.366	10040.692	13677.714	17777.758
Selection Sort	1.173	6.219	22.237	49.917	90.149	138.971	553.237	1244.131	2233.392	3430.996	4911.488	6670.933	8675.457
Merge Sort	0.607	1.748	3.704	4.713	5.981	7.878	17.141	26.111	36.478	46.361	57.105	67.521	78.389
Quick Sort	0.297	1.297	1.793	3.623	3.784	4.744	12.523	20.633	30.841	43.990	57.991	75.849	91.001
Counting Sort	0.399	0.200	0.408	0.819	0.396	0.889	1.600	1.820	2.255	2.590	3.190	3.590	4.287

The table above shows the application output. The output are average times in milliseconds calculated running each algorithm 10 times for different input sizes, in the range from 100 elements to 10 000 elements.

It is noticeable how Bubble sort has by far the worst performing times. While the running time isn't significantly higher on the input size of a 100, where it takes approx. 2 sec to sort, it grows very fast as the input increases. To sort an array of 10 000 elements, Bubble sort takes almost 18 sec.

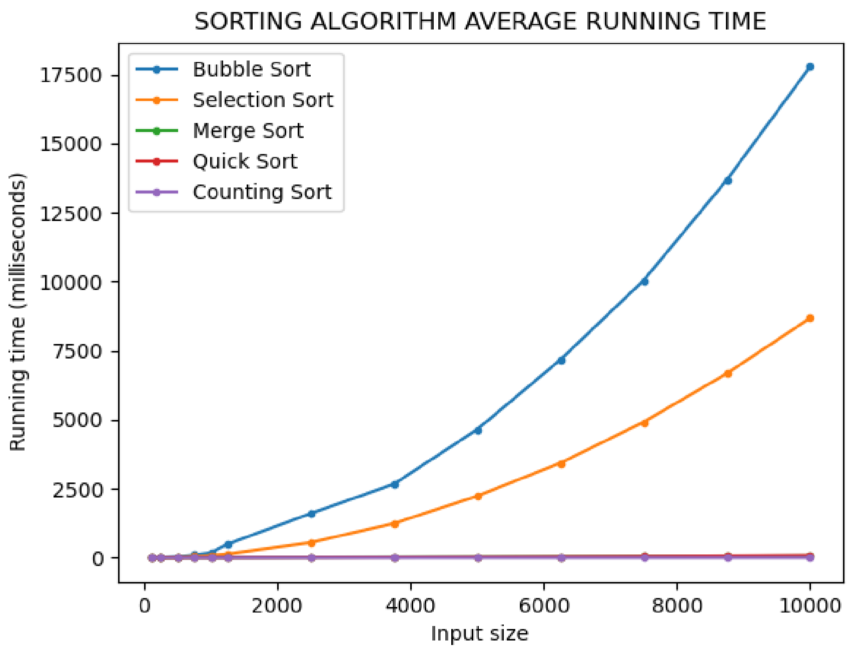
The other simple sorting algorithm, Selection sort, performs better than the Bubble Sort, but it is noticeable that its running time is still very high. It takes approx. 1 sec to sort an array of 100 elements, but already on 250 elements, it gets up to 6 sec. Its running time continues to increase significantly as the input size grows. Selection sort is more than two times faster than the Bubble sort on the input size of 10 000, but is still very slow compared to the other three sorting algorithms.

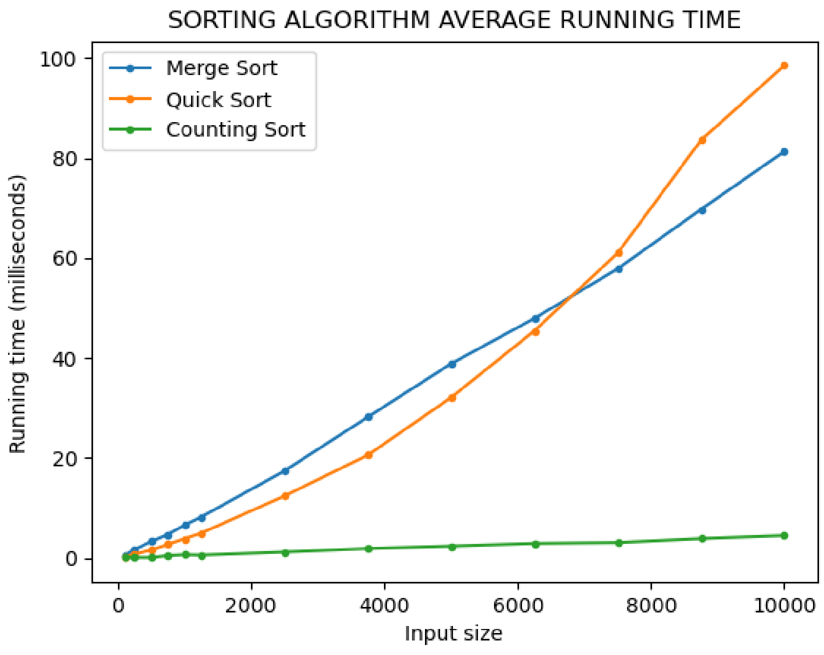
Merge sort and Quick sort are the two efficient sorting algorithms introduced in this project. They both have much faster running times than the simple sort algorithms, but are still considered slow compared to the Counting sorting algorithm. Both Merge and Quick Sort algorithms perform relatively well for the small input sizes. In fact, Quick sort has the best running time for the input size of 100 elements, but as the input increases the running time gets higher. It is noticeable how Quick sort has better running time on the input sizes from 100 to 6250 elements than the Merge sort. It takes approx. the same time to sort the array of 6250 elements, but then its running time gets worse for the higher input sizes. Quick sort takes approx. 9 sec to sort an array of 10 000 elements, while Merge sort takes approx. 7 sec to sort an array of the same input size. All this confirms the statement above; Quick sort is more efficient for the smaller input sizes, while Merge Sort performs better on the larger input sizes, when compared to each other.

The only non-comparison algorithm introduced in this project is the Counting sort algorithm. Overall, it is the fastest algorithm tested in this project. Its running time growth is insignificant as the input size increases, compared to the other tested algorithms. There are instances where this algorithm sorts elements faster on the higher input sizes than on the lower ones. For example, it takes 0.2 ms to sort an array of 250 elements, while it takes 0.39 ms to sort an array of 100 elements. Same for the input size of 1000 elements. It takes approx. 0.4 ms to sort the array of the mentioned size, while 0.8 ms for the array with an input size of 750. Overall the running time of the Counting sort algorithm gets higher as the input size grows. This difference in not very noticeable in real time (for this benchmark purposes). For comparison, for an array with 100 elements, Counting sort takes 0.0003 sec to perform and for an array with 10 000 elements, it takes 0.004 sec. It is important to note that the range of the elements for all instances of the input sizes is the same, from 0 to 100.

Below is a visual representation of this table. It gives an insight on the time complexity of each algorithm.

The graphs show the average running times for all Five tested algorithms, discussed above. Because the running times of the Simple sorting algorithms are much higher than the other tested algorithms, and it is impossible to tell the difference between the remaining algorithms, there are 2 separate graphs, the first showing all five sorting algorithms, and the second one with Merge, Quick and Counting Sort algorithms only.





It is clearly visible from the first graph that Bubble sort is by far the slowest of the Five sorting algorithms. It is followed by a slightly faster, but still very slow Selection sort. As stated before, both Bubble and Selection sort have an average running time of n^2. If we compare this graph to the illustration introduced at the start of this report, we can confirm this statement. Merge sort and Quick sort algorithms have average time complexity of n log n. Same as for the simple sorting algorithm, this is clearly visible from the graph above. We can also see how Quick sort performs better than the Merge sort on the smaller input sizes, while Merge sort is more efficient on the input sizes with larger values. It is also visible how the efficient sorting algorithms have better running times than the simple sort ones, but are still much slower than the Counting sort. Counting sort has the average time n+k. Same as for the other sorting algorithms, this statement can be confirmed. It is the fastest of all Five sorting algorithms introduced in this project.

Finally, we could say these benchmark results are in line with the expectations. Each of the five sorting algorithms performed in the manner explained in the sorting algorithm introduction.

REFERENCES:

- Algorithm Time Complexity. (2022). [online] Available at: https://justin-miguel-fernandez.medium.com/algorithm-time-complexity-9bf96bc84d2d [Accessed 1 May 2022].
- Collins Dictionary 2022. [online] Available at: <https://www.collinsdictionary.com/dictionary/english/sorting> [Accessed 25 April 2022].
- Counting Sort - GeeksforGeeks. (2022). [online] Available at: https://www.geeksforgeeks.org/counting-sort/ [Accessed 5 May 2022].
- dineshpathak, a., 2022. *Algorithmic Complexity*. [online] Devopedia. Available at: <https://devopedia.org/algorithmic-complexity> [Accessed 28 April 2022].
- En.wikipedia.org. 2022. *Algorithm - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Algorithm> [Accessed 25 April 2022].
- En.wikipedia.org. 2022. *Quicksort - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Quicksort> [Accessed 4 May 2022].
- freeCodeCamp.org. 2022. *Sorting Algorithms Explained with Examples in JavaScript, Python, Java, and C++*. [online] Available at: <https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/> [Accessed 27 April 2022].
- GeeksforGeeks. 2022. *QuickSort - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/quick-sort/> [Accessed 4 May 2022].
- GeeksforGeeks. 2022. *Sorting Terminology - GeeksforGeeks*. [online] Available at: <https://www.geeksforgeeks.org/sorting-terminology/> [Accessed 1 May 2022].
- Medium. 2022. *A Simplified Explanation of Merge Sort*. [online] Available at: <https://medium.com/karuna-sehgal/a-simplified-explanation-of-merge-sort-77089fe03bb2> [Accessed 3 May 2022].
- Patrick Mannion. *Analysing algorithms Part 1 (lectures),* GMIT, Ireland [2].
- Patrick Mannion. *Sorting Algorithms Part 1 (lectures),* GMIT, Ireland [1].
- Patrick Mannion. *Sorting Algorithms Part 2 (lectures),* GMIT, Ireland [3].
- Patrick Mannion. *Sorting Algorithms Part 3 (lectures),* GMIT, Ireland [4].
- QuickSort (With Code in Python/C++/Java/C). (2022). [online] Available at: https://www.programiz.com/dsa/quick-sort [Accessed 4 May 2022].
- QuickSort (With Code in Python/C++/Java/C). (2022). [online] Available at: https://www.programiz.com/dsa/quick-sort [Accessed 4 May 2022].
- Runestone.academy. 2022. *The Selection Sort — Problem Solving with Algorithms and Data Structures*. [online] Available at: <https://runestone.academy/ns/books/published/pythonds/SortSearch/TheSelectionSort.html> [Accessed 2 May 2022].
- Techie Delight. 2022. *In-place vs out-of-place algorithms | Techie Delight*. [online] Available at: <https://www.techiedelight.com/in-place-vs-out-of-place-algorithms/> [Accessed 1 May 2022].
- Time & Space Complexity of Counting Sort. (2022). [online] Available at: https://iq.opengenus.org/time-and-space-complexity-of-counting-sort/ [Accessed 5 May 2022].

CODE:

- Python Program for Bubble Sort - GeeksforGeeks. (2022). [online] Available at: https://www.geeksforgeeks.org/python-program-for-bubble-sort/ [Accessed 1 May 2022].
- The Merge Sort — Problem Solving with Algorithms and Data Structures. (2022). [online] Available at: https://runestone.academy/ns/books/published/pythonds/SortSearch/TheMergeSort.html [Accessed 1 May 2022].
- The Quick Sort — Problem Solving with Algorithms and Data Structures. (2022). [online] Available at: https://runestone.academy/ns/books/published/pythonds/SortSearch/TheQuickSort.html [Accessed 1 May 2022].
- Counting Sort in Python. (2022). [online] Available at: https://stackabuse.com/counting-sort-in-python/ [Accessed 1 May 2022].
- Selection Sort in Python. (2022). Retrieved 1 May 2022, from https://stackabuse.com/selection-sort-in-python/ [Accessed 1 May 2022].

CREDITS:

- Diagrams created using Diagram (2022.) Available at: https://app.diagrams.net/