Benchmarking Sorting Algorithms

ANDRES PENAS PALMEIRO
GMIT - 2020
HIGHER DIPLOMA IN SOFTWARE DEVELOPMENT
COMPUTATIONAL THINKING WITH ALGORITHMS

Table of Contents

Introduction:

Sorting can be defined as the operation of arranging a collection of items according to some pre-defined ordering rules. A sorting algorithm specifies a particular way to perform this operation in a step-by-step approach. The idea of sorting is particularly relevant in computer science as numerous tasks become simple by properly sorting the information in advance (T. Heineman, Pollice, & Selkow, 2016) and its study dominated the early days of computing.

The sorting operation is considered successfully achieved when each item in the collection is less or equal to its successor, being reorganised such that if "A[i] < A[j]" then "i < j", *i.e.*, items valued less than other items must appear before –or greater later–, in the sorted result. Also, duplicate elements must be contiguous: if "A[i] = A[j]", there cannot be "k" such that "i < k < j" and "A[i] != A[k]", that is, no element with different value can appear between elements with the same value. And finally, the sorted collection must be a permutation of the elements that originally formed the collection.

Some types of data can have a natural comparability (the possibility to be compared, or being comparable elements), such as numbers, or even characters –lexicographical order–, but other types of data must have a customized pre-defined (before the sorting operations takes place) ordering scheme –requisite illustrated by the "Dutch National Flag Problem"– specified by a comparator function that ensures that "the elements in the collection being compared […] admit a total ordering. That is, for any two elements *p* and *q* in a collection, exactly one of the following three predicates is true: *p = q, p < q*, or *p > q*" (T. Heineman et al., 2016, p. 55). Comparator functions define the ordering rules that enable custom objects to be sorted following its definition of "greater than", "equal to" or "less than" for this particular collection of objects. Sorting algorithms are independent of the comparator functions: they use them, but its definition falls outside sorting algorithms' boundaries, being just a pre-requisite for a collection to be eligible for the sorting operations. Sorting algorithms assume their existence and implementation.

There are numerous sorting algorithms, varying in code complexity, performance, and other characteristics. The performance of an algorithm can be measured in terms of its efficiency –resource consumption or computational cost– and it can be affected by multiple factors such as the input size, the relation between the inputs –e.g. if they are in a specific range– or the extent of pre-sorted items, among others. No algorithm is the best for all possible situations, and numerous factors must be taken into consideration when deciding which to algorithm use. It is important to understand the strengths and weaknesses of the different algorithms to decide which one would be more suitable for the specific task in mind. Usually, the criteria followed to select one or another sorting algorithm is based on the following characteristics: stability, efficiency, in-place sorting and suitability.

The stability of a sorting algorithm depends on its treatment of duplicated values. An algorithm is considered stable if it maintains in the ordered collection the relative order of two equal elements in the unordered collection. An unstable sorting algorithm does not respect the relative order of duplicated elements.

The efficiency of an algorithm is directly related to the resources used by it. There are many ways in which the resources used by an algorithm can be measured. Speed[1] and memory usage are the two most common measurements of algorithms' efficiency, and these are typically expressed in Big-O notation. Big-O notation classifies algorithms' complexity –computational cost– according to the

[1] Even if terms like "speed" or "time" are used, they refer not to the time that it takes for an algorithm to perform –as it depends on multiple variables– but the running time, that is, the number of operations executed, understanding an unit of time as a simple statement (B. Koffman & A.T. Wolfgang, 2005).

asymptotic upper bound in relation to the input growth rate. That is, describes how the computational cost grows as the input size grows (in the long term, or tending to the infinite), in the worst-case scenario, and taking only in consideration the higher order of magnitude (ignoring constants and constants multipliers, as it looks at performance for larger sizes of "n", where the influence of constants is minimum) of the function. For example, an algorithm with a simple statement in a nested loop, five simple statements inside another loop and 25 simple statements, would have the following function of growth: $T(n) = n^2 + 5n + 25$, which in Big-O notation would result in $O(n^2)$, meaning that the time complexity for this algorithm, in the worst case scenario, would increment directly proportional to the squared size of the input data with the input "n" tending to the infinite. For an algorithm with O(n) time complexity –also called linear– its time complexity will double as the input size "n" doubles, *i.e.*, its time complexity will increase proportionally to the input size. Other notations used to classify algorithms are the Big Omega (Ω) or Big Theta(Θ), expressing the best-case scenario –lower bound– and average-case scenario, respectively.

In relation to memory usage[2], a sorting algorithm is considered as "in-place" if it only uses a fixed additional amount of working space independently of the input size –space complexity of O(1)–, while a non in-place sorting algorithm may require additional working memory depending on the input size.

As for suitability, the properties of the algorithm should match the class of expected input, as different algorithms perform in different ways depending on the class of input on which the algorithm will operate with, and it needs to be assessed individually for any particular task and algorithm.

In general, we can group sorting algorithms into four different categories: comparison-based, effective comparison-based, non-comparison based and hybrid. Comparison-based algorithms perform the sorting of the input data by comparing the elements between them and deciding its ordering; the efficient comparison-based algorithms implement further strategies to make the comparison more efficient (for example, Merge sort follows a "divide and conquer" strategy, dividing the set of inputs in smaller units to improve comparison performance). Non-comparison sorting algorithms rely on arithmetic rather than comparison by pairs. Hybrid algorithms combine two or more different algorithms to solve the problem.

In the following sections the sorting algorithms that have been chosen for our benchmarking will be presented, along with the implementations selected and the results of our tests.

[2] The memory used by an algorithm can be expressed by two different concepts: space complexity and auxiliary space. On one hand, the auxiliary space of a sorting algorithm refers to the additional memory usage required by the algorithm. On the other hand, the space complexity refers to the total space taken by the algorithm with respect to the input size, including both auxiliary space and space used by the input ("What does 'Space Complexity' mean? - GeeksforGeeks," n.d.). The term "space complexity" is often used to express the "auxiliary space", as it is not possible to improve performance over the space used by the input itself. However, in this project "space complexity" will be used in the common general meaning to avoid confusions, as the specific differentiation between both concepts falls outside of the boundaries of this report.

Sorting algorithms

In this section, the selection of algorithms chosen to perform the benchmarking will be described and its operations explained. The time and space complexity will also be presented, along with multiple examples that illustrate their behaviour in different scenarios.

Bubble sort

Bubble sort is a sorting algorithm that belongs to the group of simple comparison-based sorting algorithms. The first printed appearance of the term "Bubble Sort" was in 1962, although it seems to have been referred by other names –as "sorting by exchange algorithm"– previously (Astrachan, 2003). Its name comes from the way this algorithm operates: iterating repeatedly through the list to be sorted comparing adjacent values and exchanging them if they are in the wrong order, *i.e.*, "the smaller values *bubble* up to the top of the array (toward the first element) while the larger values sink to the bottom of the array" (B. Koffman & A.T. Wolfgang, 2005, p. 513).

Bubble sort operates under the following structure:

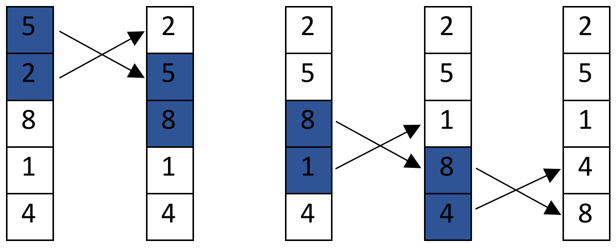
Step 1: Compare two adjacent values in the array.

Step 2: If they are out of order, exchange them.

Step 3: Repeat as many times as total pair-comparisons in the array (one less than the length of the array).

To illustrate how the algorithm works, we can follow the next example:

Given an array with five elements {5, 2, 8, 1, 4}, Bubble sort would:



- Compare the first two adjacent elements: 5 <-> 2
- Exchange them if they are out of order.
- Compare the next two adjacent elements: 5 <-> 8
- They are not out of order.
- Compare the next two adjacent elements: 8 <-> 1
- ...

At this moment, one iteration would be completed, with four[3] more still to be performed until the array is finally sorted. This example illustrates the "bubbling" mechanism, as the largest element has been *bubbled* to the bottom of the array, and links us to another observation: the last element is already sorted, but the iteration will still irrevocably check its order, leading to an obvious reduction in efficiency. This problem can be fixed by setting the iteration to check one item less each time. Another optimization would be to set a "flag" as a Boolean value to check if an exchange has been made through the iteration, breaking the sort once no exchanges have been made –as that would

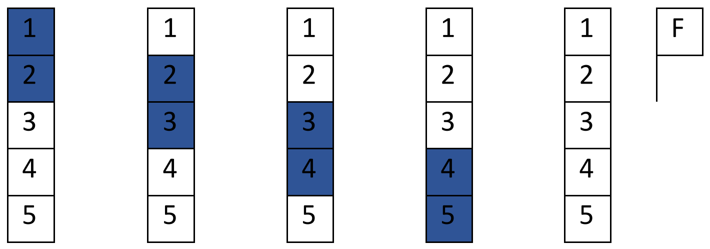
[3] Further optimizations can set boundaries in the number of comparison-iterations over de array. For example, setting the limit as "n-1", avoiding one last iteration in worst-case scenarios as the elements must be already sorted.

mean that the array is already sorted. This implementation would end the iterative process as soon as the array is sorted, saving several iterations in nearly sorted arrays (B. Koffman & A.T. Wolfgang, 2005).

The best performance of Bubble sort would be when the given array is already sorted, as there would not be any exchanges to be made; in this situation its performance would be Ω(n), as the loop would only be performed once, checking that the elements are in order and not exchange has been made, setting the flag to stop the iterations. The opposite scenario would lead to its worst performance, O(n^2), as the array would be in inverted order, and the loop would iterate over all[4] the array performing pair-comparisons each time. The statements inside of the nested loop –the exchange statements– do not affect the time complexity of the algorithm, as they have a constant time complexity: O(1). The average case sets this sorting algorithm as a quadratic algorithm as well, being its time complexity Θ(n^2).

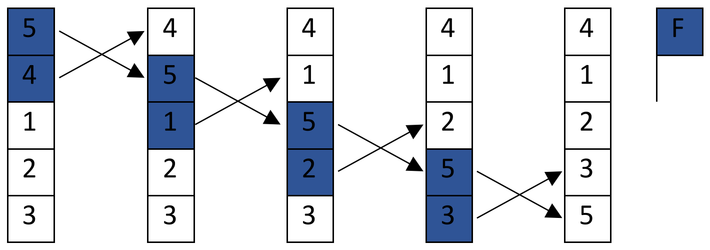
The following examples illustrate the behaviour of Bubble sort in the best, average and worst scenario.

Best-case scenario (elements already sorted). Given an array with five elements {1, 2, 3, 4, 5}, Bubble sort would:

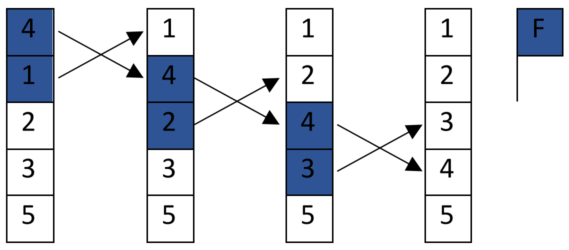


- After one iteration over the array the Flag "F" did not switch to True, as there has not been exchanges between values. The array is considered sorted.

Average-case scenario (elements partially sorted). Given an array with five elements {5, 4, 1, 2, 3}, Bubble sort would:

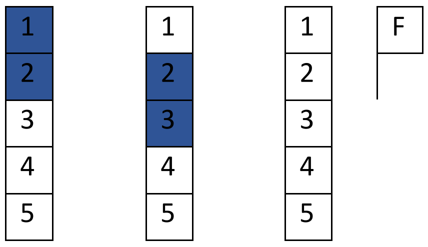


- The first iteration placed the value "5" in place. Exchanges have been made, so the Flag is set to true and the next iteration begins:



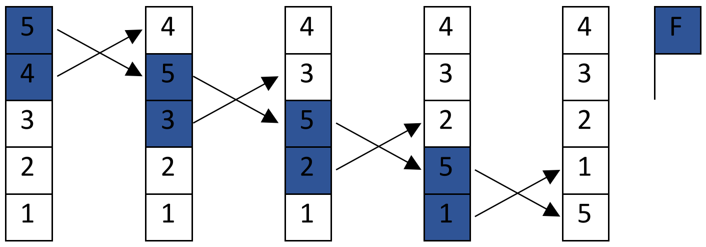
[4] When talking about iteration over arrays in comparison-based scenarios, "all" means "n-1", as the operation is performed by pairs.

- The array is already sorted, but the Flag has been set to True again as there had been exchanges, so the algorithm iterates one last time over the pair-comparison:

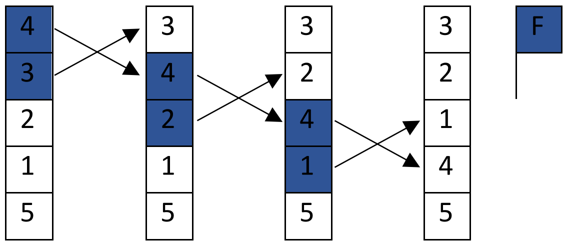


- The Flag has not been switched, so the algorithm understands that the array is already sorted and stops.

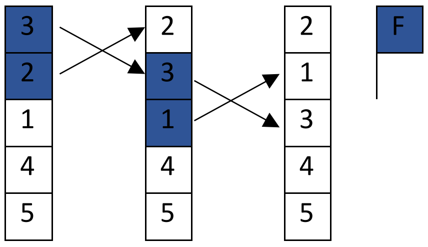
Worst-case scenario (inverse order of the elements). Given an array with five elements {5, 4, 3, 2, 1}, Bubble sort would:



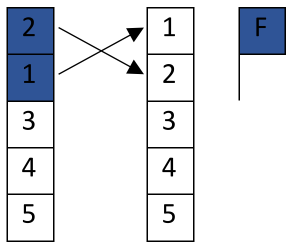
- The first iteration places the value "5" in its place, "sinking" the larger value to the bottom. The flag was activated, so the algorithm performs the next iteration.



- Two values have been placed correctly and exchanges have been made. Note that the second loop compares one element less each time, as it is considered sorted. Next iteration:



- Three of five values have been placed correctly.



- The last two values have been placed correctly. The algorithm stops even if exchanges have been made, as optimized implementation details[5].

The space complexity for Bubble sort is O(1), because only a single additional memory space is necessary (variable used to exchange elements), being an in-place algorithm. Bubble sort maintains duplicates in their relative order, for what it is considered a stable sorting algorithm.

Selection sort

Selection sort is a sorting algorithm that belongs to the group of simple comparison-based sorting algorithms. The operation of this algorithm is based on the following idea: dividing the collection to be sorted into two different subsections of it –already sorted elements and unsorted elements–, select the minimum value of the unsorted collection and replace it for the value on the next available place of the sorted collection. In a "step by step" approach:

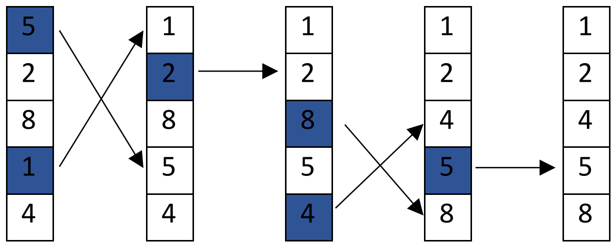
Step 1: Compare all elements in the collection to find the minimum value.

Step 2: Exchange the minimum value with the value at the beginning of the collection.

Step 3: Repeat comparing the unsorted elements of the collection and replacing it for the next value from the previously sorted elements.

The following example will illustrate the idea:

Given an array with five elements {5, 2, 8, 1, 4}, Selection sort would:



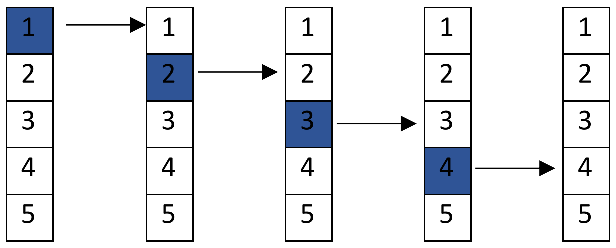
- Compare elements to find the minimum value: M→1
- Exchange the minimum value with the value of the first element: 5 <-> 1
- Compare the unsorted elements of the collection to find the minimum value: M → 2
- Exchange it with the value of the first element of the unsorted elements: 2→2
- …

[5] See "Implementation and Benchmarking: Bubble Sort Implementation" for further details.

At this point, the collection would be sorted[6].

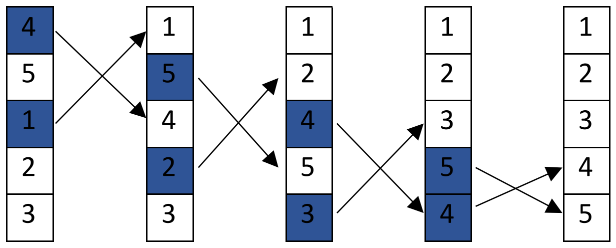
The time complexity for the Selection sort algorithm is: $\Omega(n^2)$, $\Theta(n^2)$ and $O(n^2)$, as it iterates through the whole collection performing comparisons of inputs to locate the minimum value each time no matter what –even in the best-case scenario, with an already sorted collection. The following examples will illustrate these situations.

Best-case scenario. Given an already sorted array with five elements {1, 2, 3, 4, 5}, Selection sort would:



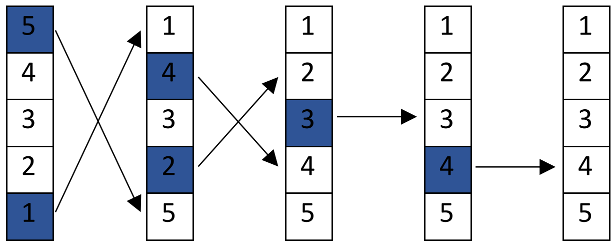
- The best-case scenario presents 4 iterations over the array, selecting in each one the minimum value and exchanging it with the one in the correspondent place in the subarray (with itself, in this example), with no changes in time complexity over other scenarios.

Average-case scenario. Given a partially sorted array with five elements {4, 5, 1, 2, 3}, Selection sort would:



- The average-case scenario presents 4 iterations as well, each one with selection and exchange of values, with no change in time complexity over best-case scenario.

Worst-case scenario. Given an array with five elements in reverse order {5, 4, 3, 2, 1}, Selection sort would:



[6] This representation could lead to misunderstandings: each column with shaded values represents one value of the first level iteration and one whole second level iteration, as the algorithm selects the minimum value of the array and exchanges it with the value in the first position of the "sorted" subarray –empty at the beginning. This example presents a case with 4 different values in first level iterations over the array, performing its operations in each of them. The first level iteration selects the correspondent place in the sorted subarray, while the second level iteration selects the minimum value by pair-comparison.

- The worst-case scenario presents 4 iterations, each of them with selection and exchange of values, with no change in its time complexity over other scenarios.

Because Selection sort only uses a single additional memory space to perform exchanges, is an in-place algorithm –space complexity O(1). Selection sort does not take into consideration the relative order of duplicates, for what it is considered as unstable or non-stable sorting algorithm.

Insertion sort

Insertion sort is another algorithm that belongs to the family of the simple comparison-based algorithms. The operations made by this sorting algorithm are often compared to the technique used by card players to arrange a hand of cards: "The player keeps the cards that have been picked up so far in sorted order. When the player picks up a new card, the player makes room for the new card and then *inserts* in its proper place" (B. Koffman & A.T. Wolfgang, 2005, p. 516). Insertion sort divides the collection of inputs in two subsections separated by a "key" element, being the first subsection formed by the first element at the beginning, and the second one by the rest of the original collection. A comparison between the "key", the value of the first element from the second subsection, and the values of the sorted subsection is made, placing the new element in its ordered place.

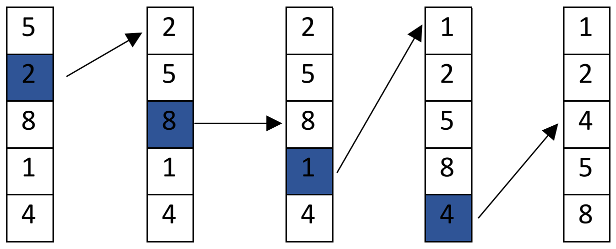
Step 1: Select the key as the second element of the collection.

Step 2: Compare the key value with the elements before it.

Step 3: place the key in its correct position.

The following illustration explains the operations made by the Insertion sort:

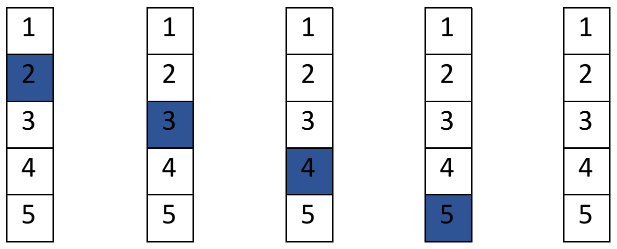
Given an array with five elements {5, 2, 8, 1, 4}, Insertion sort would:



- Select the key value as the second element of the collection: K → 2
- Compare to the elements situated before the key: 2 <-> 5
- Place the key element in its correct position shifting the elements larger than itself.
- Select the next value as key.
- …

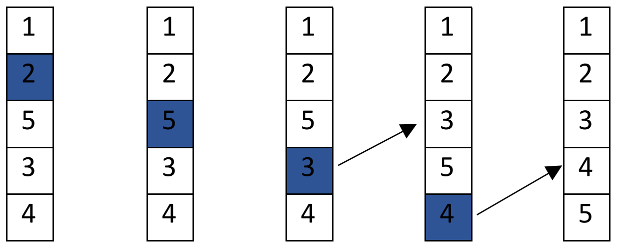
The performance of Insertion sort in the best-case scenario is Ω(n) time complexity, as there are no inversions to be made when the collection is already sorted and just iterates through the elements of the collection. In its worst-case scenario, with the collection in reverse order, the time complexity is quadratic O(n^2), displacing all elements of the collection in each iteration. The time complexity in an average scenario is also Θ(n^2), as it is considered to displace half of the inputs. The following examples illustrate these scenarios.

Best-case scenario. Given an already sorted array with five elements {1, 2, 3, 4, 5}, Insertion sort would:



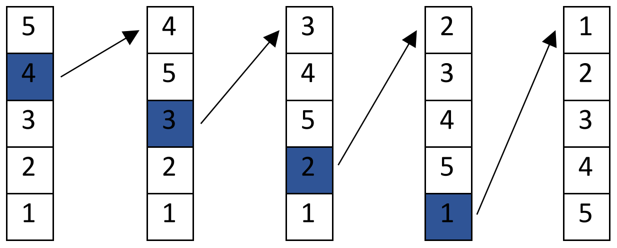
- In the best-case scenario, the second level of iterations does not run, as its condition is not meet. Only the first level of iterations is activated, with Ω(n) time complexity. The algorithm selects its key value in each iteration of the first iteration level, looking for larger elements to shift; as it does not find any, the second iteration level is not triggered.

Average-case scenario. Given a partially sorted array with five elements {1, 2, 5, 3, 4}, Insertion sort would:



- In the average-case scenario, the second level of iterations runs half of the times, giving to the algorithm, in this scenario, Θ(n^2) time complexity.

Worst-case scenario. Given an array in reverse order with five elements {5, 4, 3, 2, 1}, Insertion sort would:



- In the worst-case scenario, the second level of iterations always runs. The algorithm always finds a larger element to shift its position.

The space complexity for Insertion sort is O(1), using only a single space of memory to store the key; for this reason we can consider it as an in-place algorithm. Also, it is stable, as it maintains duplicates in their relative order.

Merge sort

Merge sort is an efficient comparison-based sorting algorithm proposed by John von Neumann in 1945, which uses the “divide and conquer” strategy to approach the task of sorting the input data. Merge sort divides the given collection in half recursively[7] and then merges the sorted halves.

Step 1: Divide phase:

Step 1.1: Divide the collection in half.

Step 1.2: Repeat recursively.

Step 2: Merge phase:

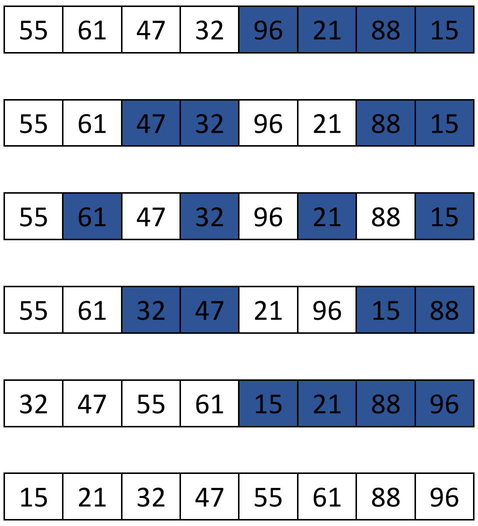
Step 2.1: Sort the halves by pair-comparison.

Step 2.2: Merge the halves.

Step 2.3: Repeat recursively.

The following example will illustrate the idea:

Given an array with five elements {55, 61, 47, 32, 96, 21, 88, 15}, Merge sort would:



1. Split the 8-element array
2. Split the 4-element arrays
3. Split the 2-element arrays
4. Merge the 1-element collections into 2-element arrays
5. Merge the 2-element collections into 4-element arrays
6. Merge the 4-element collections into an 8-element arrays

- Recursively split the array in halves.
- Recursively merge the halves sorting the elements.

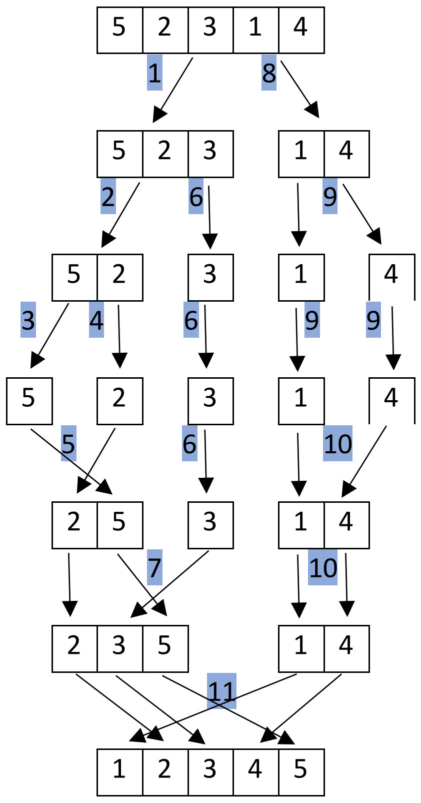
At this moment, the given array would be sorted.

Our illustrative example may lead to misunderstanding following the path of execution of the algorithm. The “Divide phase” divides the given array in halves recursively, splitting it in half from the first divided array, until reaching the base case. Then, the call to the “Merge phase” would be done giving these two 1-element arrays. With the merge called (and done), the retroactive path would end up in the last 2-element array which has not been split yet (second half of the last 4-element array) , to reach the two 1-element arrays and merge them, and then, continue with the retroactive path with the divided arrays until getting to the first original half sorted. After that, the same process would start

[7] An iterative version is also available. The recursive version is the one chosen for our exposition as it is the most commonly used.

on the second original half. The Recursion in this method is binary, calling twice for itself, always following the path of the first recursive call, to start with the second recursive call in the retroactive path of the first recursive call. It is an inverted tree structure, and the path of execution always leads first to the left of the tree, until reaching the base case, when it continues with the first right-path from the bottom of the leftmost path (unique "branch" at the moment). The following example observes the path of execution:

Given an array with five elements {5, 2, 3, 1, 4}, Merge sort would:



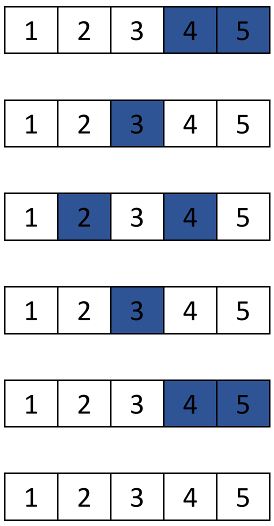
The order of the steps has been shaded. The steps 5, 7, 10 and 11 are the "Merge phase".

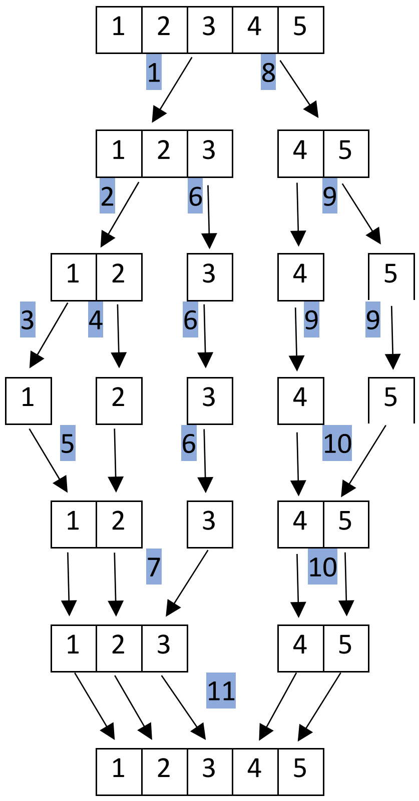
It is relevant to note that the implementation of the algorithm used in for this project[8] operates with the indexes, not with the values, until the sorting stage of the merge phase. The original unsorted array carries over the method signature splitting its index, and each merge is done based on the virtual subarrays of the original array (based on the indexes "l" and "r"), working between the indexes indicated. The indexes act as the limits defining the subarrays of –and on– the original array, carrying the partially sorted array over.

The time complexity for this algorithm is: Ω(n log n), Θ(n log n) and O(n log n), as it splits the given array O(log n) times, and the merge phase is O(n). Merge sort does not have any mechanism to change its behaviour when in a best-case scenario –array already sorted–, because of this it would still split and merge the input array. The following examples will illustrate these scenarios.

[8] See "Implementation and Benchmarking" bellow.

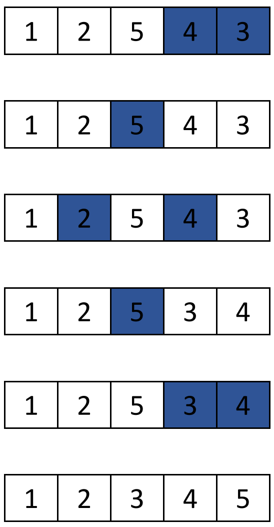
Best-case scenario. Given an already sorted array with five elements {1, 2, 3, 4, 5}, Merge sort would:

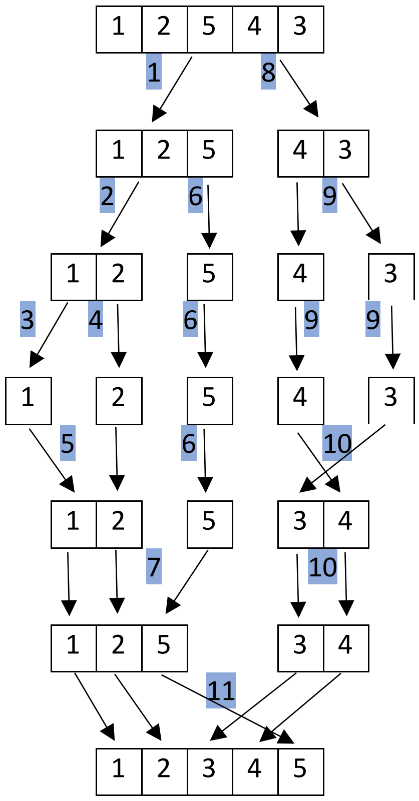




- In the best-case scenario Merge sort splits all subarrays until getting to 1-element array Ω (n log) and merges them Ω(n), with Ω(n log n) time complexity.

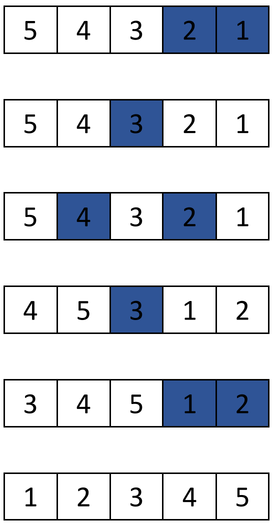
Average-case scenario. Given a partially sorted array with five elements {1, 2, 5, 4, 3}, Merge sort would:

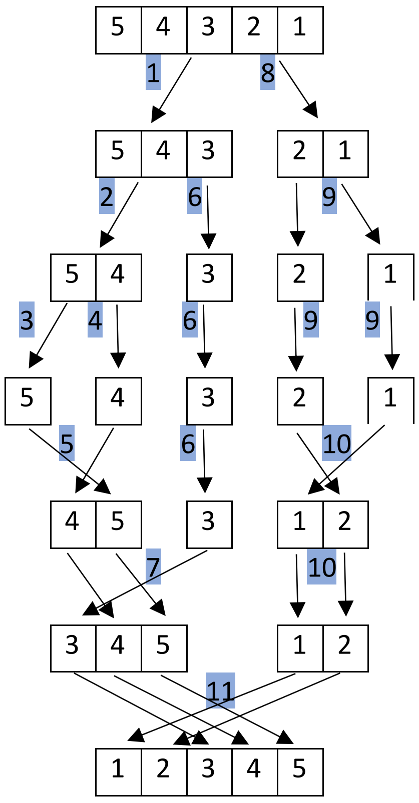




- In the average-case scenario Merge sort splits all subarrays until getting to 1-element array Θ (n log) and merges them Θ(n), with Θ(n log n) time complexity.

Worst-case scenario. Given an array with five elements in reverse order {5, 4, 3, 2, 1}, Merge sort would:





- In the worst-case scenario Merge sort splits all subarrays until getting to 1-element array O (n log) and merges them O(n), with O(n log n) time complexity.

As far as for space complexity, it is important to notice that the implementation studied is an optimized version of the original naïve implementation that, instead of operating with indexes –as the optimized version does–, operates with actual subarrays of the unsorted array[9]. The naïve implementation would have a O(n log n) space complexity, as "each recursive call of *sort* will require space equivalent to the size of the array, or O(n), and there would be O(log n) such recursive calls; thus the storage requirement for this naïve implementation is (n log n)" (T. Heineman et al., 2016, pp. 81-83). The space complexity for the optimized implementation is O(n). Also, Merge sort is stable, as it maintains duplicate values in their relative order.

Radix sort

Radix sort is a non-comparison-based sorting algorithm that was developed for the first time in 1954, by Harold H. Seward, adapting the Radix sort proposed by Herman Hollerith in 1887 to work on tabulating machines. It is closely related to Counting sort, also developed by Seward, as Counting sort is usually used as subroutine of Radix sort. This algorithm operates by creating and distributing elements into buckets according to their value in relation to their radix, and iterating over the collection as many times as significant digits have the greatest element. Its operation allows the algorithm to sort the elements of the collection in just one iteration (one per significant digit of the

[9] Further information about the naïve version can be found in (T. Goodrich & Tamassia, 2006, pp. 488ss).

maximum element), as it positions the elements by their own properties –with some arithmetical operations– rather than comparing the elements between them. This sorting algorithm can be presented sorting from the Most Significant Digit or the Least Significant Digit. In our approach to the sorting algorithm, the LSD version will be presented, as it is the most used for integer representations, as the MSD version –mainly used for Strings or Integers with fixed-length– falls outside of the boundaries of this exposition. In the same way –for simplicity and our boundaries project–, "Radix: 10" will be the default value in our approach to the algorithm's operations. Step by step, the algorithm operates as follows:

Step 1: Get the maximum value of the collection.

Step 2: Iterate over the collection as many times as significant digits have the maximum value.

Step 3: Apply Counting sort isolating the significant value analysed in the iteration.

Step 3.1: Count how many times each element is present in the collection.

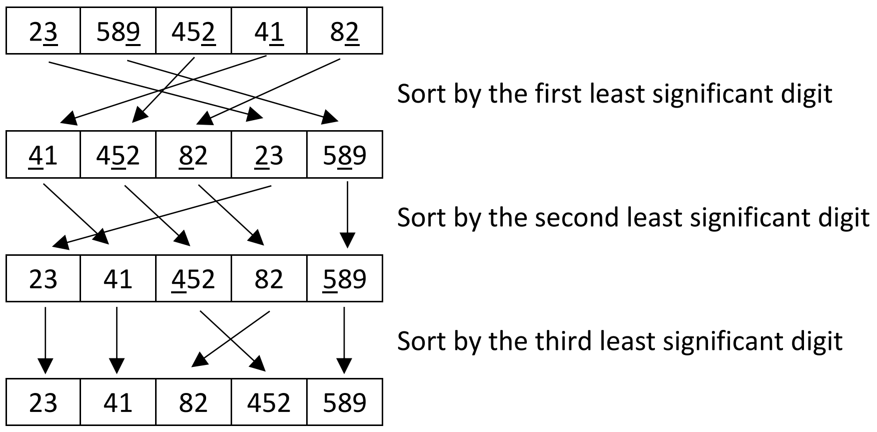
Step 3.2: Store the count of each different element in its correspondent value-index-relation position in a radix-length array: the value of the element as the index of the count array, and the value of the count array as the count of each different element.

Step 3.3: Add each value of the count array to its predecessor to calculate its sorted position.

Step 3.4: Position the index of count array in the position of the sorted array indicated by the value of count array: index of count array as value in the sorted collection in the index of the sorted collection corresponding to the value of the count array. Each time a value is positioned, "1" is subtracted from its "added count" value; this allows duplicate values to be positioned.

The following example will illustrate the structure of the algorithm:

Given an array with five elements {23, 589, 452, 41, 82}, Radix sort would:


23 589 452 41 82
Sort by the first least significant digit
41 452 82 23 589
Sort by the second least significant digit
23 41 452 82 589
Sort by the third least significant digit
23 41 82 452 589

The operation would continue as many times as significant digits the greatest element of the collection has. This collection would be already sorted. It is worth mentioning that elements with less significant digits than the greatest are padded with "0" as values to the right.

The following illustration approaches the sorting operation (Counting sort).

Given an array with five elements {5, 2, 5, 1, 4, 8, 9}, Counting sort, as subroutine of Radix sort would:

5	2	5	1	4	8	9

0	1	2	3	4	5	6	7	8	9	Index of the array to count values of the original array (Radix = length)
	1	1		1	2			1	1	Count of appearances of values in the original array
0	1	2	2	3	5	5	5	6	7	Sum of the value with its predecessor to calculate position

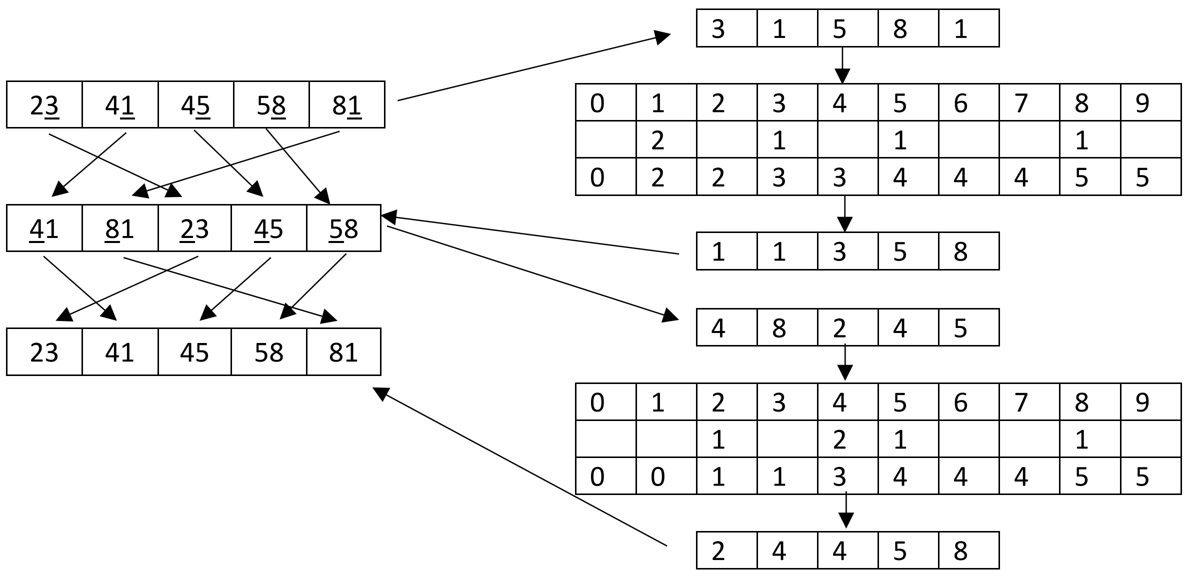
Storage of the values of the count array in the position where its value indicates: first position with value 1, second position with value 2, third position with value 4, fifth position with value 5, fourth position also with value 5 (duplicate element positioned after the subtraction of "1" in the "added-count" value, sixth position with value 8, and seventh with value 9

1	2	4	5	5	8	9	Sorted array

The operations of Counting sort show how an array can be sorted in one iteration. Counting sort as subroutine of Radix sort performs this operation analysing the correspondent significant digit.

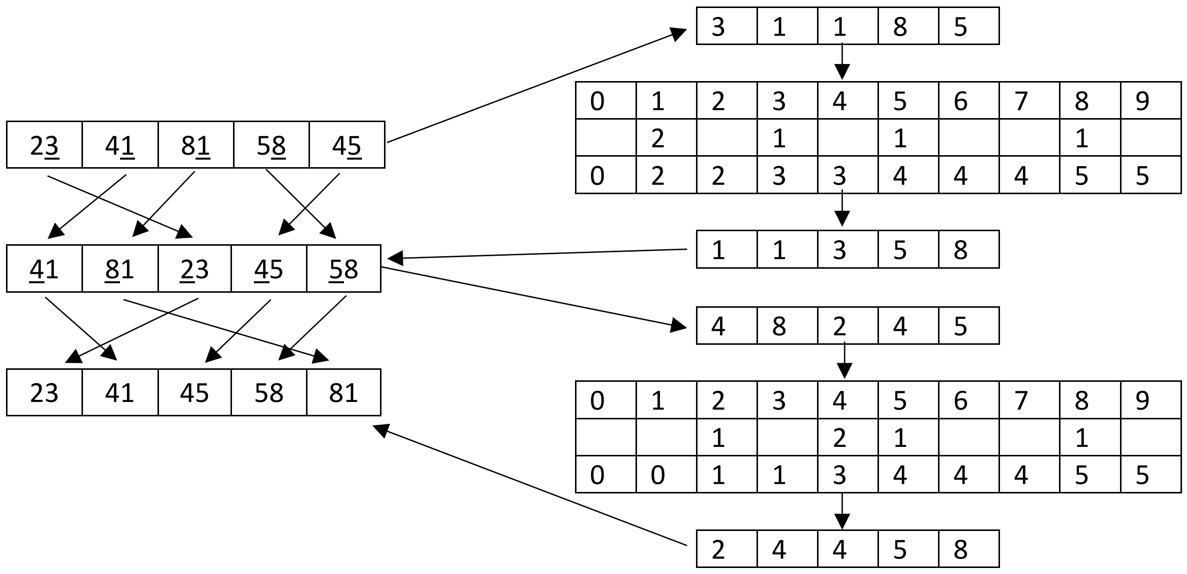
The performance of Radix sort is: Ω(nk), Θ(nk) and O(nk). The variable "k" represents the number of digits of the maximum element to sort, which also represents the number of iterations that the algorithm needs to sort the collection. The time complexity is the same in the best, average and worst scenarios as the number of iterations does not change in relation to the scenario; if the sorting operation relies on the properties of the elements themselves rather than in the relation between elements, the order of their presentation cannot alter the performance of the algorithm's behaviour. The following examples illustrate these scenarios.

Best-case scenario. Given an already sorted array with five elements {23, 41, 45, 58, 81}, Radix sort would:



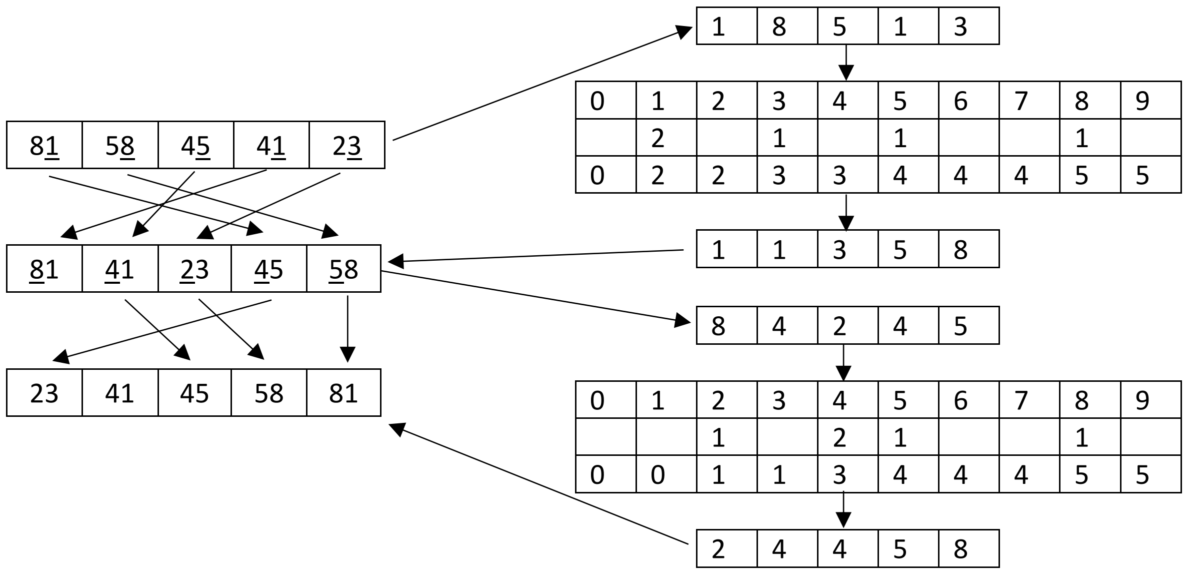
- In the best-case scenario Radix sort still performs its digit-distribution. Because of that, its time complexity does not vary.

Average-case scenario. Given a partially sorted array with five elements {23, 41, 81, 58, 45,}, Radix sort would:



- In the average-case scenario Radix sort performs the same operations, not varying its time complexity.

Worst-case scenario. Given an array with five elements in reverse order {81, 58, 45, 41, 23}, Radix sort would:



- In the worst-case scenario Radix sort performs its digit-distribution, without varying its time complexity.

The space complexity for this version of Radix sort is O(n + k), but it depends on the subroutine used to sort. In our exposition Counting sort has been used, so the space complexity of Radix sort relies on the space complexity of Counting sort. This sorting algorithm uses "n" plus 1 additional space of memory per value in the range of elements to sort("k"), which in Radix sort is usually equivalent to the range of the radix on use. The stability of Radix sort also relies on the subroutine used to sort, and being Counting sort stable, the Radix sort studied can be considered as stable too.

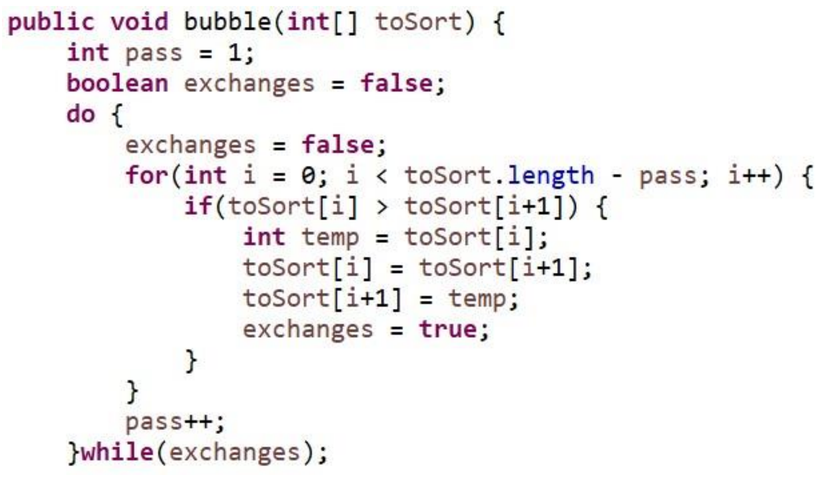
Implementation and Benchmarking

To compute the performance of the algorithms studied in this report a suite of 10 independent trials for each different “n” input size chosen was constructed. Each trial is formed by a collection of “n” randomly generated integers in the range of 0-99. The same block of 10 independent collections is used to execute the 5 different sorting algorithms for the correspondent “n” input size to study. The execution of the different algorithms with the same independent collections is made to control potential differences in the percentage of pre-sorted elements in the randomly generated collection, measuring the different algorithms with the same percentage of pre-sorted elements. Each algorithm operation is measured 10 times per “n” input size, with millisecond-level timing taken before and after the observable behaviour –sorting the collection of integers. The score of the algorithm is calculated by the average of the 10 different timings and rounded to 3 decimals. The “n” input sizes chosen for this study are: 100, 500, 1000, 2500, 5000, 7500, 10000, 15000, 20000 and 25000. That is, 10 different randomly generated collections of “n” size with integer values in the range of 0-99 are produced; these 10 unsorted collections are tested in each sorting algorithm taking their timings and calculating the average time of the 10 trials for each sorting algorithm. This process is repeated for each different “n” value.

The following sections show the implementation –and its explanation– of each sorting algorithm used for this report. This section presents the code used[10] for each algorithm studied followed by a short and concise explanation of each variable or structure –and its purpose.

Bubble sort Implementation

The following Java implementation[11] is the optimized version of Bubble sort used to test its performance.



```java
public void bubble(int[] toSort) {
    int pass = 1;
    boolean exchanges = false;
    do {
        exchanges = false;
        for(int i = 0; i < toSort.length - pass; i++) {
            if(toSort[i] > toSort[i+1]) {
                int temp = toSort[i];
                toSort[i] = toSort[i+1];
                toSort[i+1] = temp;
                exchanges = true;
            }
        }
        pass++;
    }while(exchanges);
```

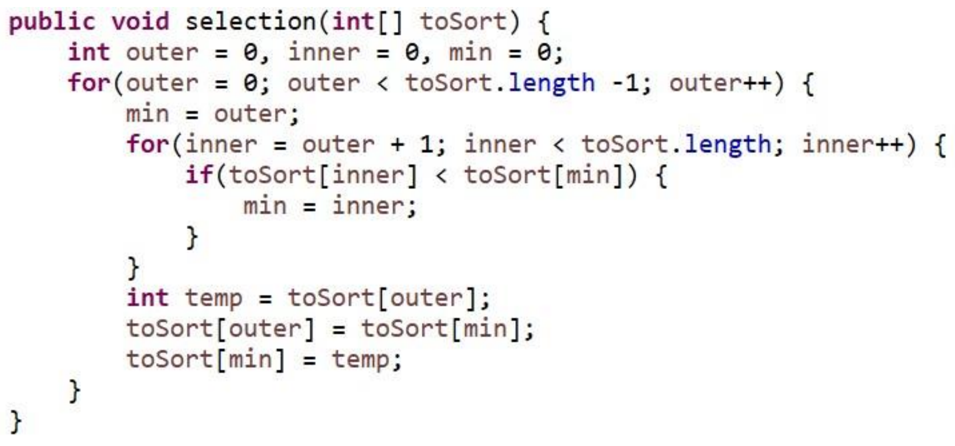
- Variable “pass” decreases the number of pair comparisons in each iteration.
- Variable “exchanges” works as the flag to indicate when no exchanges have been made in the input array.
- The “Do-While” loop performs based on the flag.
- The “inner” loop represents the iteration of pair-comparison effectuated between values.
- The conditional clause compares the values.

[10] The code presented in this report is a comments-free version of the original code used in our application. The subtraction of comments in the exposition is due to an attempt of improving the code readability, explaining each step after the code itself. The code in the Java application preserves all comments.
[11] Code source: (B. Koffman & A.T. Wolfgang, 2005).

- The code inside the conditional clause exchanges the values if necessary and operates the flag variable.

Selection sort Implementation

The Java implementation[12] of Selection sort used in this report is as follows:

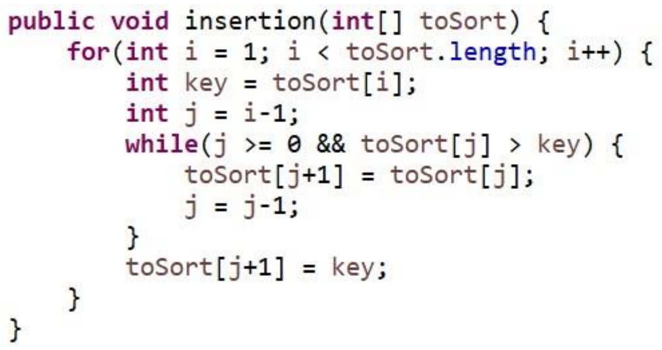


```
public void selection(int[] toSort) {
    int outer = 0, inner = 0, min = 0;
    for(outer = 0; outer < toSort.length -1; outer++) {
        min = outer;
        for(inner = outer + 1; inner < toSort.length; inner++) {
            if(toSort[inner] < toSort[min]) {
                min = inner;
            }
        }
        int temp = toSort[outer];
        toSort[outer] = toSort[min];
        toSort[min] = temp;
    }
}
```

- Variable "outer" fixes the position of the first element available of the sorted subsection.
- Variable "inner" fixes the position of the first element of the unsorted subsection.
- Variable "min" fixes the position of the value considered as the minimum value at that moment of time.
- The first "For" loop increases as the sorted subsection increases and performs the exchange after the definitive minimum value for this iteration has been found.
- The first appearance of "min" stablishes the hypothetical minimum value to start pair comparisons against it.
- The inner "For" loop performs pair-based comparisons to locate the minimum value.

Insertion sort Implementation

The Java implementation[13] for Insertion sort used is:



```
public void insertion(int[] toSort) {
    for(int i = 1; i < toSort.length; i++) {
        int key = toSort[i];
        int j = i-1;
        while(j >= 0 && toSort[j] > key) {
            toSort[j+1] = toSort[j];
            j = j-1;
        }
        toSort[j+1] = key;
    }
}
```

- The "For" loop keeps track of the key, increasing by one in each iteration.
- The variable "key" keeps the value of the element selected as the key.
- The variable "j" fixes the last position of the first subsection.
- The "While" loop compares the value of the key with all elements of the first subsection.
    - It iterates in reverse and stops once the first value that is less than the key is found, or the subsections reaches its end.

[12] Code source: Class notes.

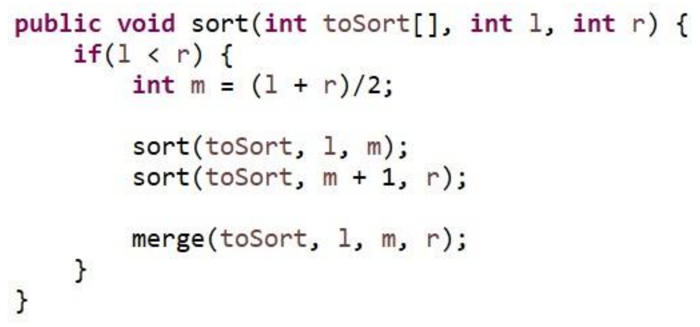
[13] Code source: Class notes.

- Statements inside the “While” loop displace the values towards the key position as the elements are being compared placing the element compared in the key’s position, making room for the key in case it is its place.
    - Note: There is no exchange, just displacement -virtual displacing, as it is a duplication made by overwriting the value towards the key–, while the value of the key is fixed by the “key” variable.
- After the “While” loop, the “For” loop places the key in its position and the next iteration begins.
    - Note: the position is “j+1” as the index of comparison has been moved to the previous element in the collection at the end of the last “While” iteration. The correct position of the key would be the next position from where the condition of the “While” loops is false, that holds the duplicated value of the last “displaced” element, and it is overwritten by the key’s value.

Merge sort Implementation

The following provides the implementation[14] used for the optimized version of Merge sort. The chosen implementation of this algorithm is formed by two different methods: the first on to recursively split the given array, and the second one to sort and merge the split arrays.

Divide phase:



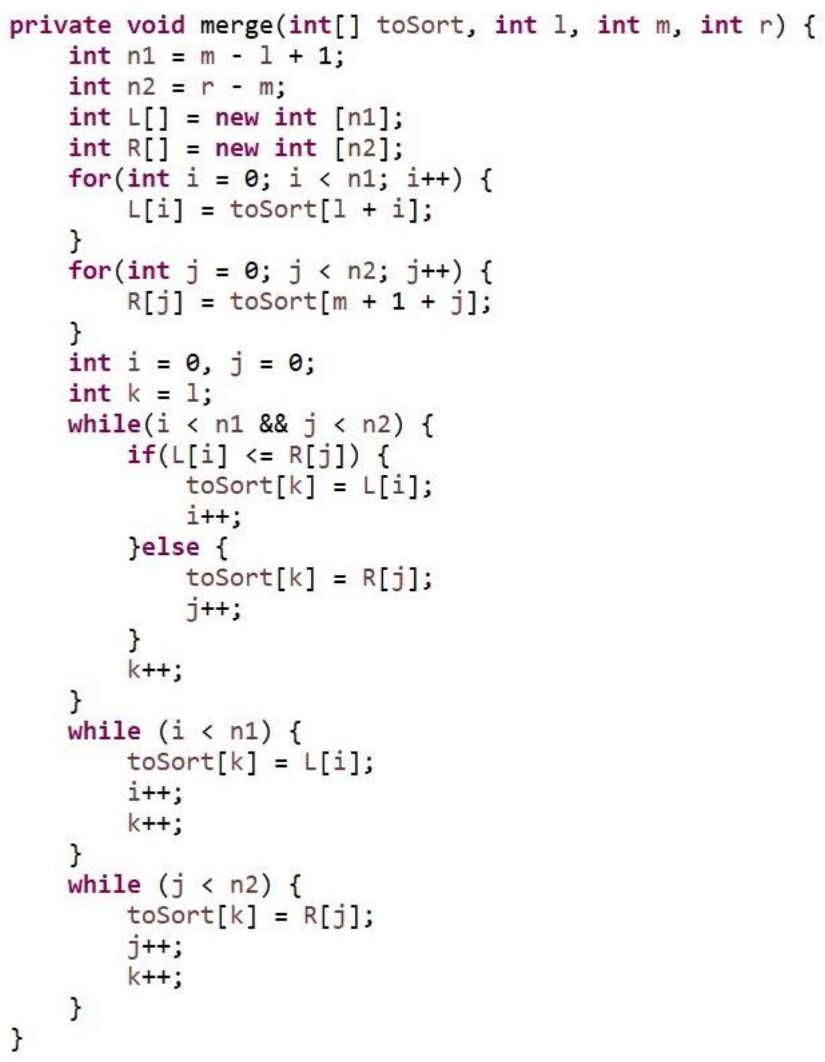
```
public void sort(int toSort[], int l, int r) {
    if(l < r) {
        int m = (l + r)/2;

        sort(toSort, l, m);
        sort(toSort, m + 1, r);

        merge(toSort, l, m, r);
    }
}
```

- Method parameters:
    - “toSort”: Array to sort.
    - “l”: Index of the leftmost side of the array to sort.
    - “r”: Index of the rightmost side of the array to sort.
- The conditional case acts as a base case for the recursion.
    - The recursion stops once 1-element array is achieved.
- Variable “m” stores the middle index of the given array.
- Recursive call:
    - Calls itself with the first half of the given array as parameter values.
    - Calls itself with the second half of the given array as parameter values.
- Base case:
    - Base case is reached once 1-element array is called in the recursion:
        - (toSort[], l, r) → (l < r) ➔ (toSort[], l, m) → !(0 < 0)
        - ({7, 2}, 0, 1) → (0 < 1) ➔ ({7, 2}, 0, 0) → !(0 < 0)
        - The algorithm continues its path with the second half:
            - (toSort[], m+1, r) → !(l < l)
            - ({7, 2}, 1, 1) → !(1 < 1)

[14] Code source: (“Merge Sort - GeeksforGeeks,” n.d.).

- Having reached the base case, continues its path calling “merge(toSort, l, m, r)”
  - Merge({7, 2}, 0, 0, 1)
  - Merge is called with the two 1-element arrays.

Merge phase:

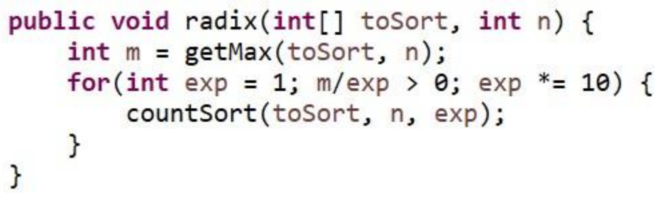


```
private void merge(int[] toSort, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[] = new int [n1];
    int R[] = new int [n2];
    for(int i = 0; i < n1; i++) {
        L[i] = toSort[l + i];
    }
    for(int j = 0; j < n2; j++) {
        R[j] = toSort[m + 1 + j];
    }
    int i = 0, j = 0;
    int k = l;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) {
            toSort[k] = L[i];
            i++;
        }else {
            toSort[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        toSort[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        toSort[k] = R[j];
        j++;
        k++;
    }
}
```

- Method parameters given by the “Divide Phase”.
- Variable “n1” stores the length of the first subarray to merge.
- Variable “n2” stores the length of the second subarray to merge.
- Variables “L[]” and “R[]” are new arrays with the length of the above variables.
- Two “For” loops to populate the subarrays with the values from the array given as method parameter.
- Variables “i” and “j” will be used to iterate over the two subarrays.
- Variable “k” will be used iterate over the array to sort in the positions where the values need to be sorted (from “l” t “r”).
- The “While” loop will select the first not positioned element of both arrays, until one of the arrays is fully positioned.
- The conditional inside the “While” loop will compare the elements selected by the “While” loop and store the correct one in the next position of the original array, to increase the position marker of the subarray where the elements have been positioned and the position of the original array.
- The next 2 “While” loops will position any element left in one of the subarrays.

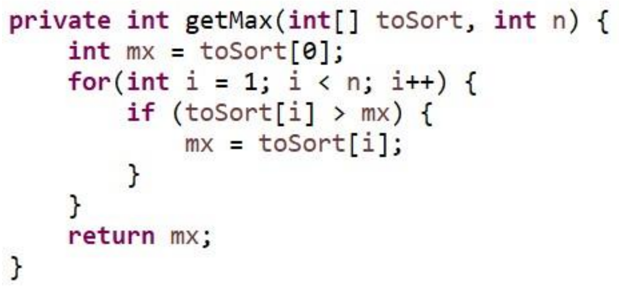
Radix sort Implementation

The Java implementation[15] used for Radix sort this project is formed by three different methods. The first one manages the iterations over the digits, the second one selects the maximum value of the given array, and the third one implements the subroutine –counting sort in our study– with the pertinent modifications.



```
public void radix(int[] toSort, int n) {
    int m = getMax(toSort, n);
    for(int exp = 1; m/exp > 0; exp *= 10) {
        countSort(toSort, n, exp);
    }
}
```

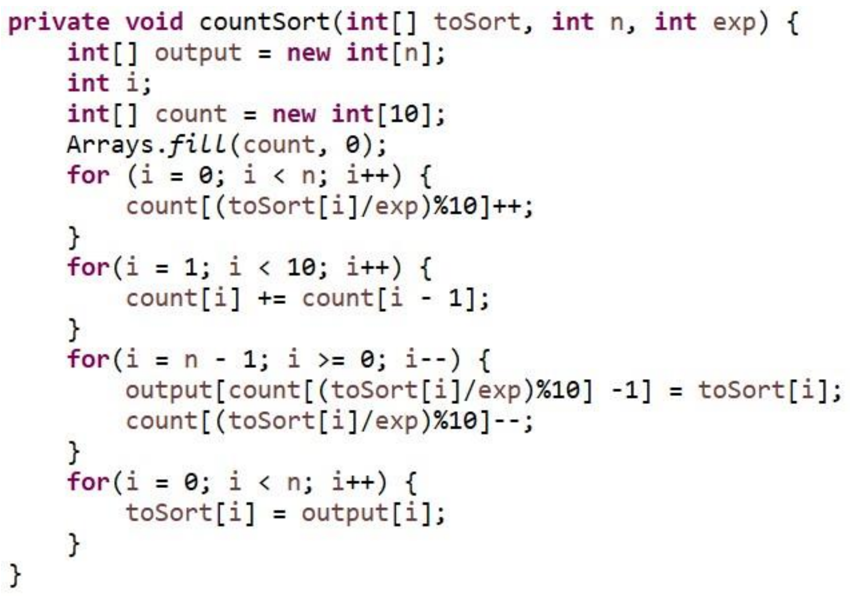
- The method parameters: Array to sort, length of the array.
- The Variable "m" to store the maximum digit.
- The "For" loop iterates over the significant digits, starting from the least significant.
  - The variable "exp" increases by multiplication of itself for 10.
  - Once the division of the maximum value by 10^i is less than 0, the iteration stops, as it has already reached its last significant digit.
    - E.g.: 845/1 = 845 → 845/10=84 → 845/100=8 → 845-1000=0
- The code inside the "For" loop call the sorting method with the array, its length and the digit to analyse in this iteration.



```
private int getMax(int[] toSort, int n) {
    int mx = toSort[0];
    for(int i = 1; i < n; i++) {
        if (toSort[i] > mx) {
            mx = toSort[i];
        }
    }
    return mx;
}
```

- The Method takes as parameters: array to sort, length of the array.
- The variable "mx" stored the first element of the array as hypothetical maximum value.
- The "For" loop iterates over the array.
- The conditional compares the elements to the one stored in "mx".
- If the element is bigger than the one stored in "mx", reassigns "mx" to it.
- Return the maximum value of the array.

[15] Source code: ("Radix Sort - GeeksforGeeks," n.d.).



```java
private void countSort(int[] toSort, int n, int exp) {
    int[] output = new int[n];
    int i;
    int[] count = new int[10];
    Arrays.fill(count, 0);
    for (i = 0; i < n; i++) {
        count[(toSort[i]/exp)%10]++;
    }
    for(i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
    for(i = n - 1; i >= 0; i--) {
        output[count[(toSort[i]/exp)%10] -1] = toSort[i];
        count[(toSort[i]/exp)%10]--;
    }
    for(i = 0; i < n; i++) {
        toSort[i] = output[i];
    }
}
```

This method implements Counting sort as subroutine of Radix sort, with some modifications.

- The method parameters: array to sort, length of the array, significant digit to analyse.
- The variable “output” initialises an array with the same length as the original array.
- The Variable “i” is going to be used for iteration purposes.
- The variable “count” initialises an array with radix as length (0-9 → 10 in our implementation).
- The method “Arrays.fill” populates the array with value “0”.
- The “For” loop iterates over the length of the arrays.
    - The code inside of the “For” loop adds “1” to the value of the count array where its index is the same as the value of the original array.
        - Counts the presence of the values in the original array and stores the count in the “count” array.
    - The operations “/exp” and “%10” isolate the digit to analyse:
        - “variable/exp”: the division of the value by the 10^i(exp) eliminates the non-relevant digits to the right of the value.
        - “value%10”: the modulus of the value by 10 eliminates the non-relevant digits to the left.
- The “For” loop iterates over the “count” array adding each element to its predecessor.
    - This operation calculates partially its position in the sorted array.
- The “For” loop iterates over the arrays (in reverse, to maintain relative position of duplicates).
    - Places the last element of the original array in the “output” array in:
        - Isolates the value of the original array where the array is iterating.
        - Takes the resultant value as index in “count” array.
            - 1 less as existence is 1 more than index (starts at 0).
        - Uses the resultant value of the “count” array as index for “output” array.
    - Subtraction of “1” in the “count” array in that position as 1 element has been correctly positioned.
- The last “For” loop iterates over the arrays to reassign the original array with the ordered elements now stored in the “output” array

Summary of the Time and Space Complexity of the Sorting Algorithms Studied:

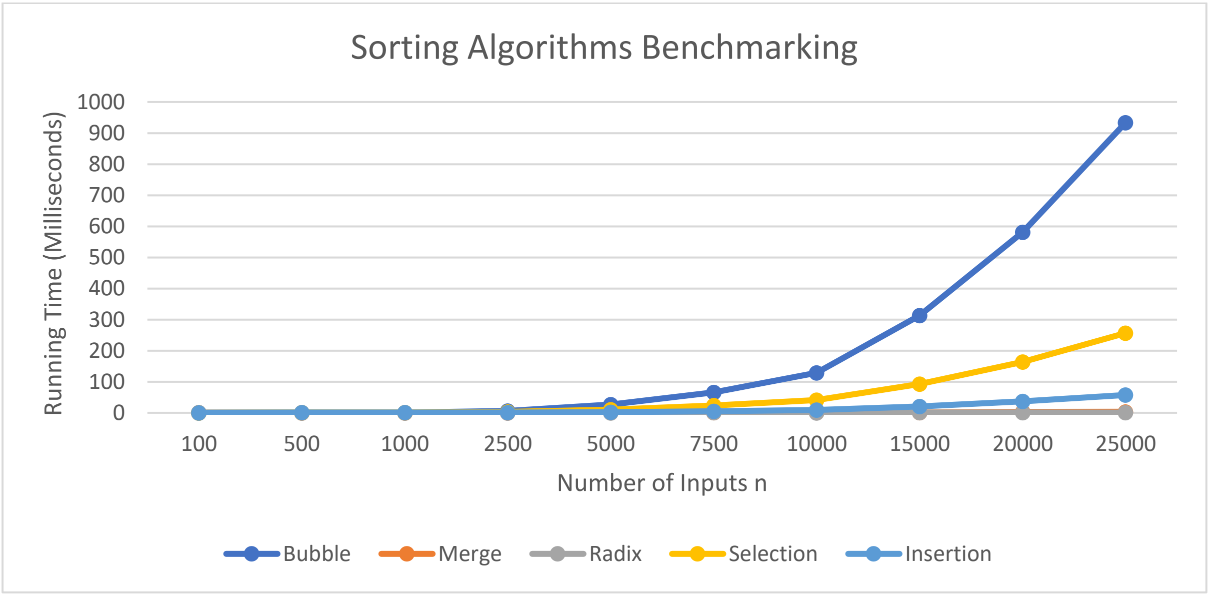
	Time Complexity			Space Complexity
	Best-case scenario (Ω)	Average-case scenario (Θ)	Worst-case scenario (O)	
Bubble sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Merge sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Radix sort	Ω(NK)	Θ(NK)	O(NK)	O(N+k)

Benchmarking Results

The results of the benchmarking suit are as follows. The value represented is the average time –measured in milliseconds– of the 10 randomly generated arrays of size "n".

Size	**100**	**500**	**1000**	**2500**	**5000**	**7500**	**10000**	**15000**	**20000**	**25000**
Bubble	0.384	0.938	1.044	5.931	26.62	65.669	129.075	313.016	581.546	933.261
Selection	0.1	1.047	0.451	2.666	10.457	23.338	41.3	92.52	164.137	256.461
Insertion	0.161	0.545	0.343	0.601	2.347	5.248	9.231	20.725	36.994	57.286
Merge	0.066	0.105	0.121	0.292	0.622	0.991	1.198	1.331	2.574	3.412
Radix	0.031	0.11	0.193	0.179	0.309	0.404	0.29	0.383	0.556	0.613

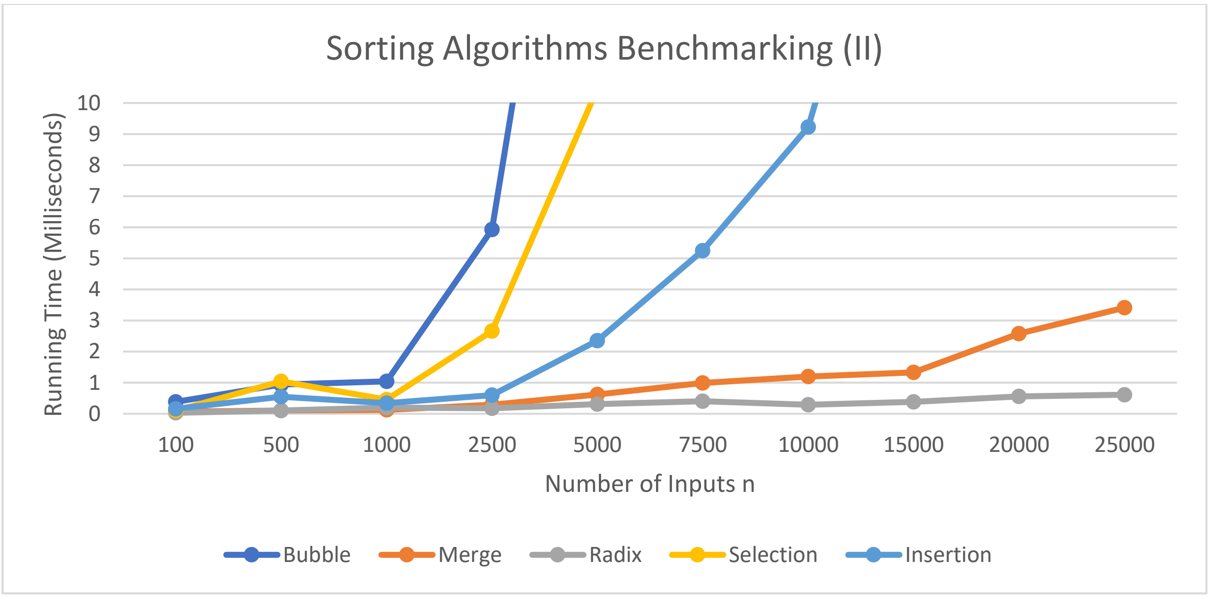
From these results, we observe that the progression of the results on Bubble sort as the size of "n" increases is surprisingly high, even more unexpected compared to other sorting algorithms with the same average time complexity (Selection and Insertion). Selection sort is approximately 364% faster than Bubble sort for an input of 25000 inputs, while Insertion sort is approximately 1636% faster for the same input size. Being the three of them part of the same family of "comparison-based" sorting algorithms, and sharing the same time complexity, such differences are definitely a surprise –which explains further the appreciation of Bubble sort as "the infamous" (T. Heineman et al., 2016, p. 57).



Another appreciation from the results is that the differences in performance between families were expected to be more uniform: there is an unexpected improvement in performance on the "efficient comparison-based" (Merge sort) compared to the best result on "comparison-based" (Insertion) in relation to the differences between the "efficient comparison-based" and the "non-comparison" (Radix sort); Merge sort is approximately 1900% faster than Insertion sort for 25000 inputs, but Radix sort is just 500% faster than Merge sort for the same input size.

The difference in performance between Bubble sort and Radix sort is unexpectedly high, being Radix sort approximately 155500% faster than Bubble sort. The performance of Radix sort in this comparison is only surpassed for the beauty and elegance of its working concept, coupled with astonishing fast times.

Merge and Radix sort have such high performance in relation to the rest of algorithms in our test than they overlap each other in the chart above. The following chart represents a closer view of their performance:



On the chart above we can appreciate how the difference in performance between Merge and Radix sort gets more pronounced on higher number of inputs.

Bibliography:

Astrachan, O. (2003). Bubble sort. *ACM SIGCSE Bulletin*, *35*(1), 1. https://doi.org/10.1145/792548.611918

B. Koffman, E., & A.T. Wolfgang, P. (2005). *Objects, Abstraction, Data Structures and Design Uing Java*. John Wiley & Sons, Inc.

Merge Sort - GeeksforGeeks. (n.d.). Retrieved May 2, 2020, from https://www.geeksforgeeks.org/merge-sort/

Radix Sort - GeeksforGeeks. (n.d.). Retrieved May 2, 2020, from https://www.geeksforgeeks.org/radix-sort/

T. Goodrich, M., & Tamassia, R. (2006). *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc.

T. Heineman, G., Pollice, G., & Selkow, S. (2016). *Algorithms in a Nutshell* (Second).

What does “Space Complexity” mean? - GeeksforGeeks. (n.d.). Retrieved April 28, 2020, from https://www.geeksforgeeks.org/g-fact-86/