Report Benchmarking Sorting Algorithms

Guilherme Galluzzi Couto Paes - G00364695

Module: Computational Thinking with Algorithms

Introduction on Sorting Algorithms

A sorting algorithm is a method that reorganizes a given number of items into a specific order, such as alphabetical, highest-to-lowest value or shortest-to-longest distance. It takes lists of items as input data, which then performs specific operations on those lists to provide ordered arrays as an output. The most frequently used orders are numerical order and lexicographical order.

When analysing a sorting algorithm, it is important to explain its worst-case, average-case, and best-case time complexity. For this reason, no algorithm is better for all occasions, so its fundamental to understand each algorithm strengths and weaknesses.

Sorting algorithms can be classified through a few characteristics, such as its time complexity, which in computer science is called as Big O notation, where it describes an algorithm running time or space requirements depending on the input size. Algorithms are also classified according to its stability, recursion, memory usage, or as is called, space complexity, as an example, selection sort has a space complexity of O(1), as iit doesn't need an extra allocation of memory in order to sort the provided list, but in comparison, merge sort, which is an efficient comparison based algorithm is different, and it has a space complexity of O(n), since merge sort divides the given array in two, creating a new list at each step, thus the sorting operation requires the allocation of a new space in memory.

Another difference when classifying algorithms is whether they are of comparison based or non comparison based, a comparison sort evaluates the data by comparing two elements with a comparison operator.

Comparison sorts uses comparison operations only to determine which of two elements should appear first in a sorted list. A sorting algorithm is called a simple comparison-based if the only way to gain information about the total order is by comparing a pair of elements at a time via the order $\leq$ (less than). Simple sorts are usually more efficient on smaller data, due to lower overhead, along this category we can think of Insertion Sort, Selection Sort and Bubble Sort.

Efficient sorts are similar as simple sorts, but they are more efficient, practical general sorting algorithms are almost always based on an algorithm with worst-case complexity of O(n log n), which the most common are heap sort, merge sort, and quicksort. Each has advantages and disadvantages, with the most significant being that the simple implementation of merge sort uses O(n) additional space, and simple implementation of quicksort has O(n2) worst-case complexity.

And with non comparison sorting algorithms, it uses the internal character of the values to be sorted. It can only be applied to some particular cases, and requires particular values. Therefore

the best case time complexity is O(n), since it's necessary to compare elements. Along this category of sorting algorithms we can point out Counting Sort, Bucket Sort and Radix Sort.

The table below illustrates all the 5 sorting algorithms used in this benchmarks along with its time and space complexity:

Sorting Algorithms Time Complexity Comparison				
Algorithm	**Time Complexity**			**Space Complexity**
	Best	**Average**	**Worst**	**Worst**
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log (n))	O(n log (n))	O(n log (n))	O(n)
Counting Sort	O(n + k)	O(n + k)	O(n + k)	O(k)

Sorting Algorithms

Insertion Sort

Insertion sort is a very popular sorting algorithm, and it works by inserting elements from an unsorted array into a sorted array, one item at a time. Insertion sort usually takes $O(n^2)$ time, which is considered too slow to be used on large data sets.

An advantage with insertion sort is the fact that it needs only a constant amount of memory space for its operation. It is more efficient than other comparison-based algorithms such as bubble sort or selection sort.

As previously mentioned, Insertion sort falls in the category of simple comparison based algorithm and even though insertion sort is efficient, if we provide an array already sorted, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

For a better understanding of how Insertion sort works, I have created this table diagram, which should help explaining how the algorithm runs:

Array	Comparison	Final Array
[**5, 7,** 2, 3]	Is 7 less than 5?	[**5, 7,** 2, 3]
[**5, 7, 2**, 3]	Is 2 less than 7? Is 2 less than 5?	[5, **2, 7**, 3] [**2, 5**, 7, 3]
[**2, 5, 7, 3**]	Is 3 less than 7? Is 3 less than 5? Is 3 less than 2?	[2, 5, **3, 7**] [2, **3, 5**, 7] [2, 3, 5, 7]

As demonstrated in the Introduction, the time and space complexity of an algorithm is a very important factor, and we can see below, the characteristics of Insertion sort:

Worst Case Time Complexity:	$O(n^2)$
Best Case Time Complexity:	O(n)
Average Time Complexity :	$O(n^2)$
Space Complexity:	O(1)

Selection Sort

Selection sort is usually considered the most simplest sorting algorithm. It works by finding the smallest element in an array and swapping it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and so forth on until the entire array is sorted. Since it repeatedly selects the next-smallest element and swaps it into the right place, its named “Selection sort”. In practice, selection sort generally performs worse than insertion sort.

Just like with Insertion sort, I have created a table diagram which explains how the algorithm works:

Array	Comparison	Final Array
[**5, 7, 2, 3**]	Find the minimum element in the array [0...3] and place it at the beginning of array	[5, 7, **2**, 3] [**2**, 7, **5**, 3]
[2**, 7, 5, 3**]	Find the minimum element in the array [1...3] and place it at the beginning of array	[2, 7, 5, **3**] [2, **3**, 5, **7**]
[2, 3, **5, 7**]	Find the minimum element in the array [2...3] and place it at the beginning of array	[2, 3, **5,** 7] [2, 3, **5**, 7]

And as well, we can't forget to mention it's time and space complexity:

Worst Case Time Complexity:	$O(n^2)$
Best Case Time Complexity:	$O(n^2)$
Average Time Complexity :	$O(n^2)$
Space Complexity:	O(1)

Bubble Sort

Bubble sort works by repeatedly stepping through lists that need to be sorted, it compares each pair of adjacent items and swaps them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted. The name comes from the fact that smaller elements bubble towards the top of the list.

The position of the elements in the array makes an important factor in determining performance. For example, large elements in the beginning are not a problem as they are easily swapped, Nevertheless, small elements towards the end are moved to the beginning slowly. For this reason, these elements are usually called rabbits and turtles.

The bubble sort algorithm may be optimized if placing larger elements in the final position. After every pass, all elements after the last swap are sorted and do not need to be checked again, thereby skipping the tracking of swapped variables.

It's considered to be a very simple algorithm to understand, but most of times impractical, although it can be practical in some cases, when the data is nearly sorted.

Here we can understand how Bubble sort algorithm works, by looking through this diagram below:

Array	Comparison	Final Array
[**5, 7,** 2, 3]	Is 5 > 7?	[5, 7, 2, 3]
[5, **7, 2**, 3]	Is 7 > 2?	[5, **2, 7**, 3]
[5, 2, **7, 3**]	Is 7 > 3?	[5, 2, **3, 7**]

[**5, 2**, 3, 7]	Is 5 > 2?	[**2, 5**, 3, 7]
[2, **5, 3**, 7]	Is 5 > 3?	[2, **3, 5**, 7]
[2, 3, **5, 7**]	Is 5 > 7?	[2, 3, 5, 7]

[**2, 3**, 5, 7]	Is 2 > 3?	[2, 3, 5, 7]
[2, **3, 5**, 7]	Is 3 > 5?	[2, 3, 5, 7]
[2, 3, **5, 7**]	Is 5 > 7?	[2, 3, 5, 7]

And here we can also compare it's time and space complexity with the other algorithms, note that the worst case is $O(n^2)$, which falls along the same line of Insertion and Selection sort:

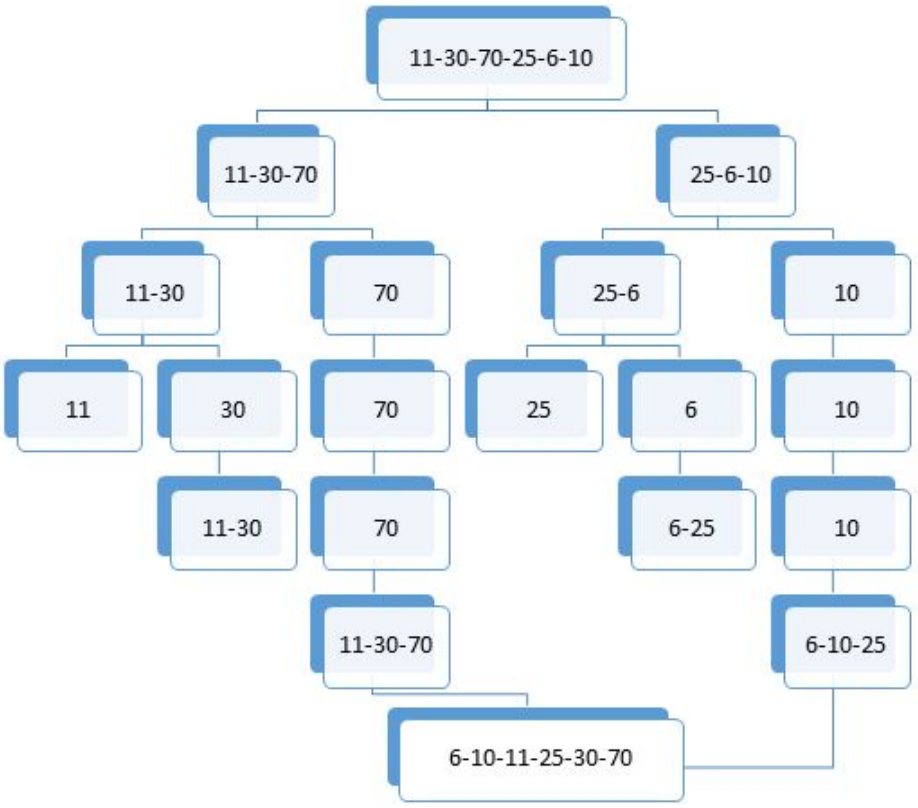
Worst Case Time Complexity:	$O(n^2)$
Best Case Time Complexity:	O(n)
Average Time Complexity :	$O(n^2)$
Space Complexity:	O(1)

Merge Sort

Merge sort was developed by John Von Neumann in 1945, but it's still widely used today, it's an algorithm that uses the divide and conquer approach, that means that if an array A contains 0 or 1 elements, then it's already sorted and nothing is done, if not, the algorithm divides A into two sub-arrays of equal number of elements and recursively sort the first and second halves separately. Finally, both sorted sub-arrays are merged into one sorted list.

The main idea behind merge sort is that, the short list takes less time to be sorted, therefore it's considered to be an efficient comparison based algorithm.

An example of how Merge Sort works:



Below we can see it's time and space complexity, and, since it's worst case is O(n log n), it makes it faster than the previous algorithms covered:

Worst Case Time Complexity:	O(n log (n))
Best Case Time Complexity:	O(n log (n))
Average Time Complexity :	O(n log (n))
Space Complexity:	O(n)

Counting Sort

Counting sort is an efficient algorithm for sorting a list of positive integers. Unlike other sorting algorithms, such as selection sort, counting sort is an integer sorting algorithm and is a non comparison based algorithm. While any comparison based sorting algorithm requires comparisons, counting sort has a running time of when the length of the input list is not much smaller than the largest key value in the list.

Counting sort was proposed by Harold Seward in 1954, and it's still very much used today, it allows to sort a collection of items in close to linear time, which is one advantage, although we can think of a disadvantage being that counting sort only works when the range of potential items in the input is known ahead of time.

Simply put, we can say that counting sort works by iterating through the list, counting the number of times each item shows in the array, and then using those counts to create a final sorted list.

Below there is a demonstration of how counting sort really works:

Consider the following array with 8 inputs in total (n), ranging from 0 to 9 (k):

7	1	2	5	2	8	9	5

A counting array is created with the index on top, and below the count of how many times each number is on the original array:

0	1	2	3	4	5	6	7	8	9
0	1	2	0	0	2	0	1	1	1

And now a final sorted array is generated according to the counted number of items:

1	2	2	5	5	7	8	9

And finally we have to understand its time and space complexity:

Worst Case Time Complexity:	O(n + k)
Best Case Time Complexity:	O(n + k)
Average Time Complexity :	O(n + k)
Space Complexity:	O(k)

Benchmarks

In this section it will be covered the results from the benchmark that was obtained from running the python file sorting.py. As previously discussed, the code is meant to run and mark the duration of the time that took to run, which will be translated to a table and later to a line graph, for better understanding and analysis of the results.

For comparison basis, it was taken 5 of the most common sorting algorithms, they being: Insertion sort, Selection sort, Bubble sort, Merge sort and Counting sort. Some of them being comparison based, simple or efficient, and also non comparison based.

First of all, in order to have a more close result as possible, the benchmarks were run 10 times for each input size n, and for each of the 5 sorting algorithms. Then the results of those 10 runs were averaged to get one final result for each sorting algorithm and its input size.

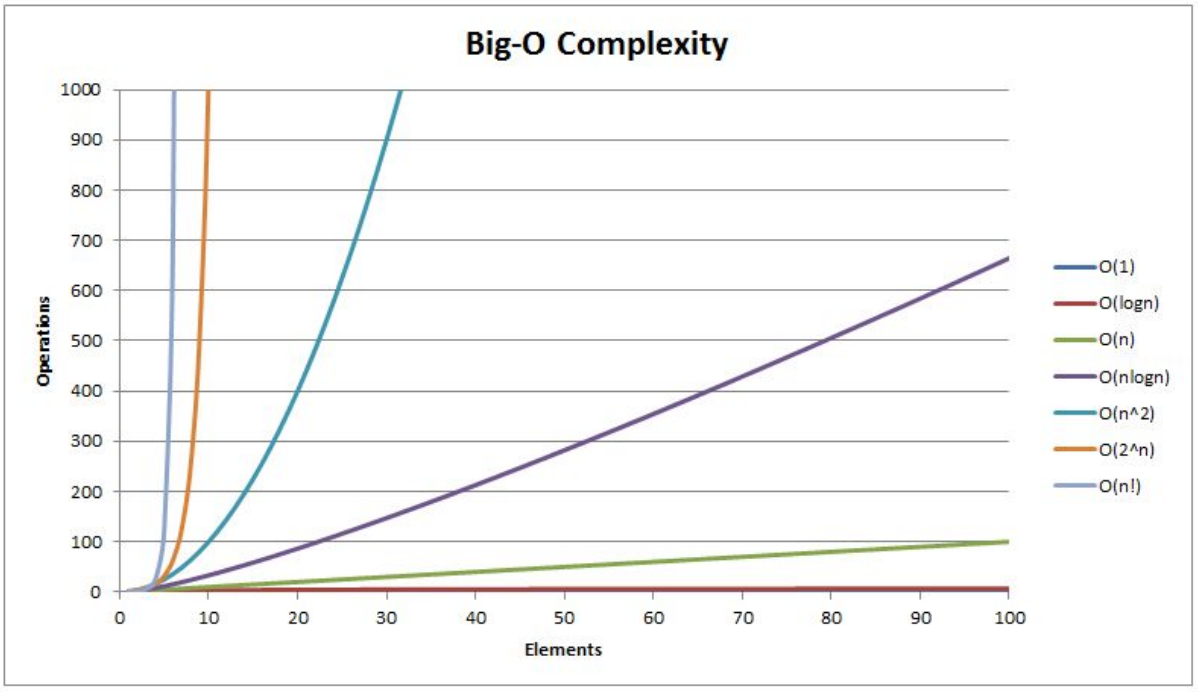
I have split each code for the sorting algorithms into different python files and made on final file for the benchmarks so it makes easier to read and understand the code, the benchmarks can be reproducible by running the file sorting.py, have in mind that the results will differ slightly from the table below.

The results below were obtained running the sorting.py code on a Dell XPS 15 laptop, with Intel i7-7700HQ @ 2.80GHz.

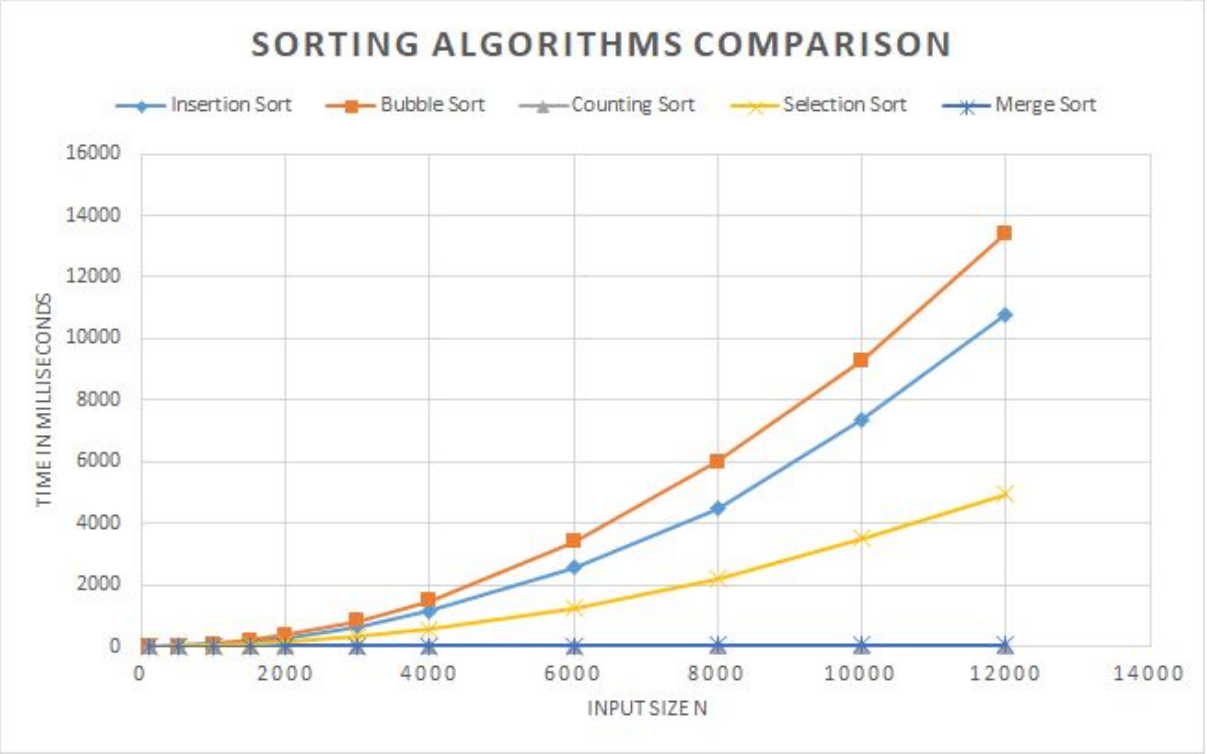
Size	Insertion Sort	Selection Sort	Bubble Sort	Merge Sort	Counting Sort
100	0	0	0	0.998	0
500	15.964	8.976	20.946	0.998	0.998
1000	76.767	33.942	92.752	3.989	0
1500	160.57	76.795	202.459	5.985	0

2000	293.216	135.637	374	8.976	1.005
3000	636.299	329.088	837.759	13.963	0
4000	1164.886	561.499	1476.055	18.95	0.997
6000	2575.152	1257.638	3411.881	28.922	0.997
8000	4488.966	2219.068	6002.988	39.893	1.995
10000	7389.246	3527.57	9278.162	52.859	1.995
12000	10784.171	4965.724	13409.152	64.861	1.994

As the table above demonstrates, we can clearly see that the fastest sorting algorithms are merge and counting sort, this happens because as previously stated, the worst case scenario for merge sort is O (n log n) and for counting sort is O (n+k), which is faster than insertion, selection and bubble sorting, since their worst case scenario is $O(n^2)$, we can understand it easier if looking at the line graph above, which demonstrates all the Big-O time complexity and it's running time:



Now, with the Big-O time complexity in mind, we can also compare our running results looking at at plotted line graph as below:



Looking at this comparison between all the 5 sorting algorithms along with the Big O time complexity graph, we can confirm that the results were as expected, when taking the Big O time complexity of the algorithms in consideration.

As demonstrated by the graph, we can see that the slowest of the 5 algorithms are insertion, selection and bubble sort, since they all are simple comparison based algorithms with a time complexity ranging between O(n) and $O(n^2)$, although between these 3, selection sort, has shown itself the fastest among the simple comparison based algorithms.

Therefore we can conclude that the benchmarks results were as expected and we can clearly see the difference in running time between all 5 different sorting algorithms, and also to have a clear understanding on each advantages and disadvantages of different types of algorithms and learning when to use a specific algorithm in a program so you can take up the most out of the application in regards to running time, stability and memory usage.

References

Brilliant.org, Simple Sorting Algorithms [Online]. Available from: https://brilliant.org/wiki/sorting-algorithms [viewed 20 April 2019].

Brilliant.org, Space Complexity [Online]. Available from: https://brilliant.org/wiki/space-complexity/ [viewed 25 April 2019].

Brilliant.org, Merge Sort [Online]. Available from: https://brilliant.org/wiki/merge/ [viewed 25 April 2019].

BigOCheatSheet, Big O Cheat Sheet [Online]. Available from: http://bigocheatsheet.com/ [viewed 25 April 2019].

Cloudboost, 2017. Simple Sorting Algorithms [Online]. Available from: https://blog.cloudboost.io/simple-sorting-algorithms-bd473e0ebd5 [viewed 02 April 2019].

GeeksforGeeks, Counting Sort [Online]. Available from: https://www.geeksforgeeks.org/counting-sort/ [viewed 11 April 2019].

GeeksforGeeks, Sorting Algorithms [Online]. Available from: https://www.geeksforgeeks.org/sorting-algorithms/ [viewed 15 April 2019].

GitHub, 2018, Sorting Algorithms [Online]. Available from: https://github.com/joeyajames/Python/blob/master/Sorting%20Algorithms/SortingAlgorithms [viewed 10 April 2019].

InterviewCake, Insertion Sort [Online]. Available from: https://www.interviewcake.com/concept/java/insertion-sort [viewed 15 April 2019].

InterviewCake, Insertion Sort [Online]. Available from: https://www.interviewcake.com/concept/java/insertion-sort [viewed 15 April 2019].

InterviewCake, Counting Sort [Online]. Available from: https://www.interviewcake.com/concept/python3/counting-sort [viewed 05 April 2019].

InteractivePython, Bubble Sort [Online]. Available from: http://interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html [viewed 05 April 2019].

InteractivePython, Selection Sort [Online]. Available from: https://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html [viewed 25 April 2019].

InteractivePython, Insertion Sort [Online]. Available from: http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html [viewed 25 April 2019].

P. Mannion (2019). Sorting Algorithms Lecture 3 - GMIT [viewed 11 April 2019].

StudyTonight, Insertion Sort [Online]. Available from: https://www.studytonight.com/data-structures/insertion-sorting [viewed 21 April 2019].

StudyTonight, Selection Sort [Online]. Available from: https://www.studytonight.com/data-structures/selection-sorting [viewed 21 April 2019].

TechoPedia, Insertion Sort [Online]. Available from: https://www.techopedia.com/definition/20039/insertion-sort [viewed 11 April 2019].

Whatis.com, 2018, Sorting Algorithms [Online]. Available from: https://whatis.techtarget.com/definition/sorting-algorithm [viewed 11 April 2019].