The Race of Sorting Algorithms

Raul Moiș
Department of Computer Science
West University,
Timișoara, Romania
Email: `raul.mois02@e-uvt.ro`

March 2022

Abstract

One of the most thoroughly studied and experimented upon domains in all of Computer Science is that of sorting algorithms, alongside the ever present desire of creating the optimal method of sorting for all problems and data sets.
Through the study of the sorting algorithms mentioned in this paper, our main goal is that of obtaining a better understanding of the way these sorting algorithms work and the situations they are best used in, all the while presenting our results and offering an analysis of the time requirements each sort presents on different data sets through an experimental approach.

Contents

1 Introduction

Throughout the years, sorting algorithms have continued to be at the center of interest of many researchers in the field, meaning that many improvements and optimisations have already been discovered and documented in great detail. While a theoretical "expected" efficiency can be estimated mathematically for each individual algorithm, these are just that, estimations, which may or may not reflect the real world performance each of these sorts provide.

Thus, the objective, and by extension the motivation behind this paper: to take all the well known theoretical aspects of these algorithms and observe the effects they have when used in real world scenarios, seeing whether or not the expected results generated by mathematical estimations translate into practice.

In order to preserve time, this paper will not present or go into details about the actual algorithms or their structure ([1] and [2] contain great showcases of those while also giving complete explanations of the theory behind them), but will mainly focus on the expectations made from their theory and the real world results.

2 The sorts and the theory

The sorting algorithms that we will take a look at in this paper are seven of the most essential and well known sorts: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Heap Sort, Quick Sort and Counting Sort.

Below will be a short overview of the expectations we can make before testing the algorithms in real-world use, based on the theory behind each individual sort. (All the algorithms and their implementations are taken from either [1] or [2] as marked after each of their names).

2.1 Insertion Sort

In the case of insertion sort, the best, worst and average cases all perform:

$$\sum_{i=0}^{n-1} i = 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

comparisons, see [1] , resulting to a time complexity of $O(n^2)$.

Insertion Sort [1] is a very simple and straight forward algorithm with an undesirable O(n^2) time complexity. It does have O(n) time complexity on sorted arrays and only O(n) total number of swaps, but we expect its performance to be pretty poor exactly because of that quadratic time complexity.

2.2 Bubble Sort

In all three cases: best,worst and average case, bubble sort performs:

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{1}{2}\sum_{i=0}^{n-2}(n-1) = \frac{n(n-1)}{2}$$

comparisons, see [1] , resulting to a time complexity of $O(n^2)$.

Bubble Sort [1] is another member of the quadratic time algorithms. It certainty is simple to understand and does still retain the same $O(n)$ best case that Insertion sort has, but the number of swaps it has to perform add up to a quadratic amount. This means that, as the number of elements needed to be sorted increases, we expect Bubble Sort's performance to see a noticeable decline.

2.3 Selection sort

In all three cases: best,worst and average case, selection sort performs:

$$\sum_{i=0}^{n-2}(n-1-i) = \frac{1}{2}\sum_{i=0}^{n-2}(n-1) = \frac{n(n-1)}{2}$$

comparisons, see [1] , resulting to a time complexity of $O(n^2)$.

Selection Sort [1] is the last of the quadratic time algorithms we will take a look at. Like the other two, it still takes O(n^2) on sorted arrays, however, in contrast with Bubble Sort, it has a reduced number of element swaps ($O(n)$ swaps). As with the other two, performance is not expected to be great because of the time complexity.

2.4 Merge sort

Merge sort has a total of:

$$M(n) = 2^i M(\frac{n}{2^i}) + 2i\,n = nM(1) + 2nlg(n) = 2n\,lg(n)$$

number of movements, see [1] for the complete explanation, resulting to a time complexity of $O(n\,log(n))$.

Merge Sort [1] is the first of the algorithms from the list able to sort in $n\,log(n)$ time, being a straight-forward representation of a Divide and Conquer algorithm. Since it utilizes temporary arrays, the space necessary is larger, at $O(n)$, and it also has the downside of doing the whole sorting process even on sorted or very short data sets. Thus, performance will be good but will probably fall behind the other $n\,log(n)$ algorithms, especially on the particular cases mentioned before.

2.5 Heap sort

Heap sort has a time complexity (see [2]) of:

$$O(n) + (n-1)\,O(lg(n)) = O(n) + O(n\,lg(n)) = O(n\,lg(n))$$

Heap sort [2] is a $n\,log(n)$ class sort that uses a heap data structure in order to achieve its goal, allowing it, thus, to sort in place. We expect it to have good performance, however, the use of the heaps might cause a slower run-time on smaller sets of data.

2.6 Quick sort

Quick sort has an average time complexity of:

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1}(T(k) + \Theta(n)) = O(n\,log(n))$$

(see [2] for elaborate explanation). However, a notable thing about it is that it has a worst case time complexity of $O(n^2)$.

Quick Sort seems to have a massive disadvantage in its worst-case scenario compared to Merge Sort. However, that case only occurs when the list is sorted and the left most element is chosen as the pivot.

As such, we expect it to likely be on par with the other $n\,log(n)$ class algorithms we are testing, only having difficulties in those particular scenarios. It also does not require any extra space or use of a data structure to work, so it might even be faster than the other two in some cases.

2.7 Counting sort

Counting sort has a time complexity of: $O(n+k)$, where k represents the range of elements needed to be sorted (see [2]).

Counting Sort's [2] time efficiency gives it an advantage performance-wise compared to all other sorts discussed, especially as the data size increases. However, in smaller data sets, where n has yet to scale up in value, we expect it to get outpaced by the other sorts since counting sort's performance is not reliant solely on n, but on the size of k as well (hence $O(n+k)$ time complexity).

3 Testing procedures

This is the part where we take an empirical approach to analysing the aforementioned algorithms. After making some basic assumptions and expectations about the performance and situations each algorithm will perform best/worst in, it is time to actually apply them on real world examples.

The implementation of the algorithms and measuring of the performance provided by the sorts was done using the programming language C++. The experiment was done on a computer with the following specifications:
CPU: AMD Ryzen 5 2600 Hexacore processor at 3400 MHz, 16gb of DDR4 RAM at 3000mghz, OS: Windows 10 x64 bit.

In particular, we took full advantage of Visual Studio 2022's Chrono library (see [3]) in order to create an isolated timer of the sorting algorithms. By utilising a "Timer" class, we can create a basic timer that measures the elapsed time from the creation of the class until it's destruction.

3.1 Creating the timer

```
Timer()
  {
    //start the timer upon creation
    m_StartTimer = std::chrono::high_resolution_clock::now();
  }

  ~Timer()
  {
      //stop the timer upon deletion
      Stop();
  }
```

This is the surface-level part of the class. When the object Timer is created it makes a new timer "StartTimer", while its deletion calls a function "Stop()" in order to compute the time elapsed. "StartTimer" simply memorizes the exact time the "Timer" object was created *aka.* when the section of code we want to time begins.

This section of code and the following one are also available to view on GitHub (see [4])

3.2 Measuring the time elapsed

```
void Stop()
  {
  //endpoint of timer set on object deletion
  auto endTimepoint = std::chrono::high_resolution_clock::now();

  //getting the start time of the timer
  auto start = std::chrono::time_point_cast
  <std::chrono::seconds>(m_StartTimer).time_since_epoch().count();

  //getting the end time of the timer
  auto end = std::chrono::time_point_cast
  <std::chrono::seconds>(endTimepoint).time_since_epoch().count();

  //calculating the duration and adding it to the time_avg
  float duration = end - start;
  time_avg += duration;
  }
```

This is the heart of the timer: the "Stop()" function. In short, upon the destruction of the object "Timer", the function creates a new timer point when the object was deleted and gets the timer point when the object was created (shown in the previous segment of code). Then, it simply computes the time elapsed by getting the time difference between the end and the start, thus obtaining the run-time of the algorithm timed.
In this way, we can encapsulate a particular portion of code within the timer. We do this by creating the object right before a section of code we want to time and deleting it right after.

In order to automate the testing process, we simply looped the algorithm while switching the files containing data that needed to be sorted. Anything that was not part of the sorting itself, such as opening/closing the files, was not part of the timer, thus having no influence on the final time results obtained. Each data set was also sorted 100 different times in order to iron out any potential precision errors that might occur.

4 Case studies and actual testing

After analyzing and setting certain expectations on the way the sorts will perform, it's time to actually sort some data and compare their performance.

The testing data consists of varied amount of integers between 1 and 100.000 and, for the sake of time, the largest data size is 1 million digits. We also generated and sorted 1000 different variations of each data size (from 100 to 1 million).

4.1 The Results

*All run-times were measured in milliseconds (unless mentioned otherwise)
1k representing 1000 integers*

Data Size	Insertion Sort	Bubble Sort	Selection Sort	Merge Sort	Heap Sort	Quick Sort	Counting Sort
100	0.0023	0.012	0.0076	0.0044	0.0016	0.0031	0.111
500	0.039	0.245	0.129	0.027	0.082	0.0188	0.112
1k	0.137	0.922	0.473	0.061	0.166	0.039	0.113
5k	3.16	24.255	10.698	0.347	0.863	0.235	0.127
10k	14.38	109.7	42.803	0.74	1.785	0.51	0.147
25k	87.73	783.3	259.836	2.002	4.724	1.403	0.196
50k	344.41	3214.9	1019.42	4.297	9.52	2.969	0.289
75k	774.07	7687.4	2274.12	6.706	14.85	4.524	0.386
100k	1319.27	13.6 sec	4.1 sec	9.04	20.59	6.205	0.488
250k	7.35 sec	88.6 sec	25.1 sec	25.08	50.54	16.37	1.186
500k	30.4 sec	358.7 sec	72.5 sec	52.76	104.36	32.64	3.731
1mil	128.26 sec	1484 sec	293.8 sec	114.34	212.74	67.31	11.02

Figure 1: Table of run-times on randomized input

Data Size	Insertion Sort	Bubble Sort	Selection Sort	Merge Sort	Heap Sort	Quick Sort	Counting Sort
100	0.0023	0.012	0.0076	0.0044	0.0016	0.0031	0.111
1k	0.137	0.922	0.473	0.061	0.166	0.039	0.113
10k	14.38	109.7	42.803	0.74	1.785	0.51	0.147
100k	1319.27	13.6 sec	4.1 sec	9.04	20.59	6.205	0.488
1mil	128.26 sec	1484 sec	293.8 sec	114.34	212.74	67.31	11.02

Figure 2: 10^k Data Sizes for easier readability

For a better visual representation, these results are also represented in a graph style below.

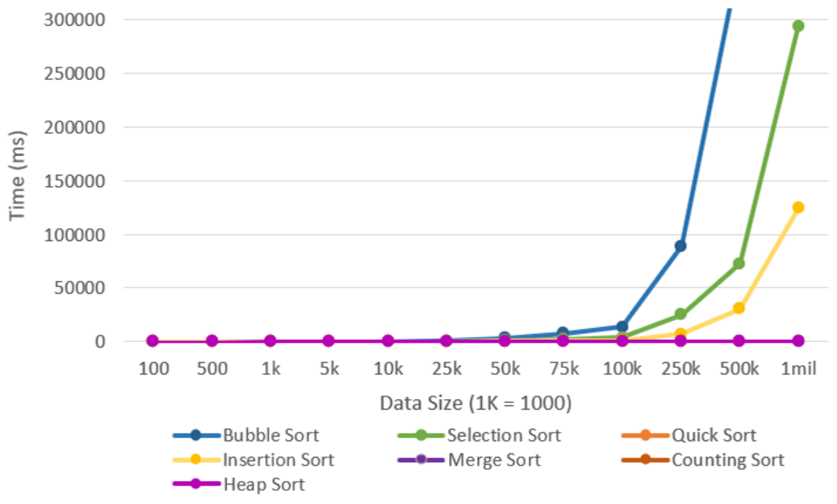


Figure 3: Graph representation of the run-times - zoomed out

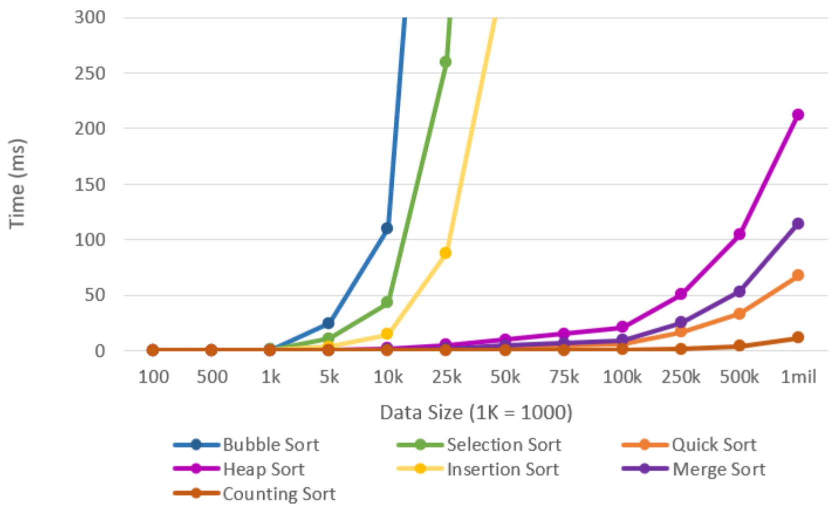


Figure 4: Zoomed in version of the graph

4.2 Interpreting the results

As expected, Bubble Sort was by far the slowest of the bunch for every array over 1.000 elements, the n^2 time complexity and higher number of swapping procedures compared to the two other sorts in the same time complexity class made it far worse as arrays got larger (it was an absolute time sink for anything over 50k).

	100	1k	10k	100k	1mil
Bubble Sort	0.012 ms	0.922 ms	109.7 ms	13.6 sec	1484 sec

Selection sort was definitively helped by its lower number of swaps, being quick compared to other sorts for anything 1k and under, but we can clearly see the slow-down on anything over that.

	100	1k	10k	100k	1mil
Selection Sort	0.007 ms	0.473 ms	42.8 ms	4.1 sec	293.8 sec

Insertion sort, as expected, pulled ahead of the other two. Being helped by the low number of swaps and its advantage when the array is sorted, it was almost the fastest on arrays of 100 integers and still competitively fast for anything 1k and under. However, anything above 1k and the exponential growth of time really shows it's impact, but that was by no means surprising.

	100	1k	10k	100k	1mil
Insertion Sort	0.0023 ms	0.137 ms	14.38 ms	13.19 sec	128.26 sec

Merge sort and Heap sort were a little confusing at first, since they both have the same time efficiency. The extra time that merge spends merging should have been offset by the time heap sort needs to build the heap itself, so we were a little surprised by the results. We tried multiple tests and even thought of rebuilding the code, but after some research we realized that, because of the way that merge sort interacts with the cache (see [5]), it was quite normal. Since the CPU used for testing has a 16 MB L3 cache, it was quite logical that the merge sort performed faster. The downside of memory still remains though and should not be ignored, since it still needs that second array.

	100	1k	10k	100k	1mil
Merge Sort	0.0044 ms	0.061 ms	0.74 ms	9.04 ms	114.3 ms
Heap Sort	0.0016 ms	0.166 ms	1.785 ms	20.59 ms	212.7 ms

Counting sort, to no one's surprise, is really fast on large inputs, being ten times as fast as heap sort for the 1 million integer array. However, it is quite slow on those lower inputs, and that is because of the proprieties the data set we chose had: integers between 0 and 100.000. Thus, on lower inputs, the size of the array we have to sort is way less than the size of k (showing the caveat of that O(k+n) time complexity). Thus, it is the slowest in the 100-size test and quite slow up until the 5k mark.

	100	1k	10k	100k	1mil
Counting Sort	0.007 ms	0.473 ms	42.8 ms	4.1 sec	293.8 sec

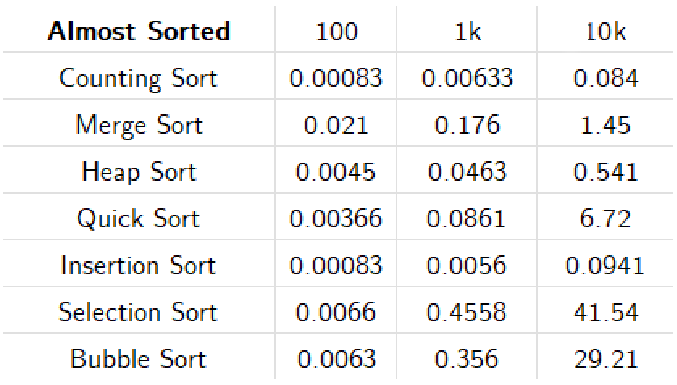
"How come it is so fast?" is what we initially thought about quick sort once we saw that 1 million array test. But it makes sense. The worst case only happens on almost or completely sorted arrays, but, on the randomly generated integer lists we tested it does not seem to have affected the performance of the sort in the grand scheme of things. It is still slower than counting sort, but it sorts in place, which can be a necessity when working with limited memory. It seems that it really is quick in practice, truly deserving of its name.

	100	1k	10k	100k	1mil
Quick Sort	0.007 ms	0.473 ms	42.8 ms	4.1 sec	293.8 sec

4.3 Almost sorted arrays

The worst case of quick sort happening on almost or fully sorted arrays gave us the idea to also test the cases where the arrays are actually almost sorted. Thus, we tested 3 extra scenarios: almost sorted arrays of random integers, sorted arrays with only the first/first few elements not sorted and arrays with only the last/last few elements not sorted, in order to see how the algorithms would perform.

The data sets used for testing this are yet again integers, this time arranged in the three particular cases stated above, with values up to 10.000. The sizes of the sets are 100, 1k and 10k integers respectively, each file being re-sorted 200 times for accuracy. The run-times were measured in milliseconds.



Almost Sorted	100	1k	10k
Counting Sort	0.00083	0.00633	0.084
Merge Sort	0.021	0.176	1.45
Heap Sort	0.0045	0.0463	0.541
Quick Sort	0.00366	0.0861	6.72
Insertion Sort	0.00083	0.0056	0.0941
Selection Sort	0.0066	0.4558	41.54
Bubble Sort	0.0063	0.356	29.21

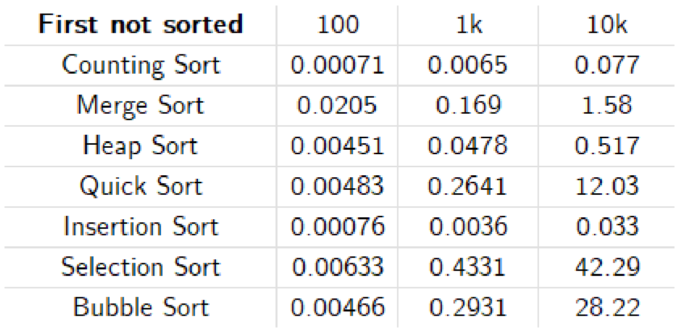
Figure 5: Run-times on random, almost sorted arrays

On almost sorted arrays, selection sort had a really rough time. It's quadratic time complexity combined with the fact that, compared to the other two n^2 class algorithms, it still has $O(n^2)$ time complexity on sorted arrays, really puts it behind when it comes to any sort of almost sorted sequence.

Quick sort also had quite a slow run, and this shows its real weakness: almost sorted arrays. As we have speculated, it was quick in practice, but where there are a lot of arrays where the chance of the first/last few elements being out of order, it falls behind the other $n\ log(n)$ algorithms.

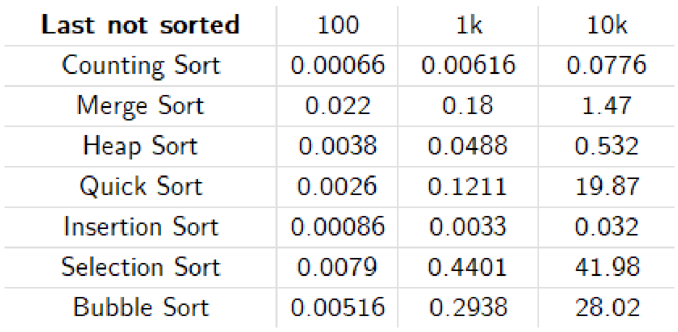
Also notable is how fast insertion sort actually is. Since the arrays are almost sorted, the reduced number of swaps and that $O(n)$ best case time complexity make insertion sort an extremely competitive sort on all three particular cases we tested in these sections.

4.4 First/Last elements not sorted



First not sorted	100	1k	10k
Counting Sort	0.00071	0.0065	0.077
Merge Sort	0.0205	0.169	1.58
Heap Sort	0.00451	0.0478	0.517
Quick Sort	0.00483	0.2641	12.03
Insertion Sort	0.00076	0.0036	0.033
Selection Sort	0.00633	0.4331	42.29
Bubble Sort	0.00466	0.2931	28.22

Figure 6: Run-times on lists with the first not sorted



Last not sorted	100	1k	10k
Counting Sort	0.00066	0.00616	0.0776
Merge Sort	0.022	0.18	1.47
Heap Sort	0.0038	0.0488	0.532
Quick Sort	0.0026	0.1211	19.87
Insertion Sort	0.00086	0.0033	0.032
Selection Sort	0.0079	0.4401	41.98
Bubble Sort	0.00516	0.2938	28.02

Figure 7: Run-times on lists with the last elements not sorted

We decided to group up the first/last not sorted cases together since they both showcase particularly well the downfall of the classic quick sort algorithm. When only the first/last few elements are not in the right place, quick sort really struggles, letting that quadratic complexity present in its worst case show its impact on performance.

The effect of choosing the pivot poorly is especially evident. In the "last not sorted" table, since the quick sort algorithm that we used for testing chose the pivot as the last element of the array, the overall run-time was the most affected, as can be seen from the difference in performance present between the two cases. This, however, can be solved by simply choosing the pivot wisely.

We can also see how selection sort continues to fall behind compared to all the others. But, again, we expected that precisely because it still maintains a $O(n^2)$ time complexity on sorted arrays, unlike the other $O(n^2)$ class algorithms we tested.

These two cases are also great for illustrating just how good insertion sort is on almost sorted arrays. When only a few elements from the front/back of the array were not in the right places, insertion sort completely destroyed all other sorts when it came to running time. An excellent performance by a seemingly not so great sort with $O(n^2)$ time complexity.

5 Conclusions

Overall, this experimental approach alongside the need to further explore the theoretical side of these sorts really helped us understand the applications of sorting algorithms better. Not only did we implement them, which came with its own fair share problems from time to time, but we also got a taste of how these algorithms operate with really large sets of data, really putting into perspective how much time complexity affects run-time when dealing with such large sets of data.

We also managed to delve a bit into how computer architecture can influence sorting, since even two seemingly identical sorts efficiency-wise such as Heap and Merge sort can still have such a gap when it comes to speed.

And, while the results of this "race" were by no means unexpected, there definitively were certain moments and "upsets" in the race that made us curious and fueled the desire to research more into how and why they can happen.

For future projects, our main focus will be on paying attention to details while testing. As stated before, there were a fair share of problems along the way until we reached this final state of the paper, especially in the experimental part. However, now that we took a objective look at our work, most of those were caused by simply not being attentive enough while creating test data or implementing the algorithms themselves when testing.

As a more personal note, after all the testing done, I do not believe there is a "best" sorting method. I do, however, think that this comparison has helped me make better decisions when applying sorts and it will surely make me constantly ask: "Wait, is this really the best sort for this situation?".

References

[1] Adam Drozdek *Data Structure And Algorithms In C++, Third Edition*, Cengage Learning (2005)

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - *Introduction to Algorithms, Third Edition*, The MIT Press (2009)

[3] Microsoft - Chrono Library, Visual Studio 2022
https://docs.microsoft.com/en-us/cpp/standard-library/chrono

[4] GitHub (https://github.com/m-raul5/race-of-sorts)

[5] Anthony LaMarca, Richard E. Ladner - The Influence of Caches on the Performance of Sorting, University of Washington, 1997