A THEORETICAL AND EXPERIMENTAL COMPARISON OF SORTING ALGORITHMS

Maximilian Todea
Department of Computer Science,
West University of Timișoara
email: maximilian.todea03@e-uvt.ro

Abstract

In this paper I will showcase a theoretical and practical comparison of some of the most common sorting algorithms. We will be splitting the algorithms into three categories: exponential algorithms - bubble, insertion and selection sort; logarithmic algorithms - quick, merge and heap sort; and non comparison-based algorithms - counting, radix and bucket sort. Through various practical tests and theoretical analysis, we will highlight each of the categories strengths and weaknesses and we will draw a final conclusion on the entire mass of algorithms: where, when and how each should be implemented in practice.

1 Introduction

We have a fairly simple problem: You are a beginner programmer and you have just finished learning nine of the most basic sorting algorithms - bubble, insertion, selection, quick, merge, heap, count, radix and bucket - you understand that they have various time and space complexities, but you are left wondering: what is the best case use of these algorithms? Better said: which is the fastest, most memory efficient sorting algorithm? That is what we will be outlining in this paper.

This paper proposes a simple solution for these questions, that is: write a Python program that times each of these nine sorting algorithms performance and compares their various properties to reach a conclusion. By the end of this paper, you should have a clear idea of when and where these algorithms can be implemented.

You may be inclined to believe that you can run your own tests throughout your programs by simply implementing whatever sorting algorithm has the fastest time complexity and adding a time function to see how long it takes to sort compared to your previously implemented algorithm. The problems with this approach would be that you would never truly get understand what leads to the algorithm to run so fast, or there may be specifics about the way each one works. Simply said, for that type of "experimental" workflow to work, you require previous knowledge of the innards of the algorithms that you want to test.

For example, you may use bubble sort to sort very large arrays - from your basic knowledge of these algorithms, you may be inclined to simply use merge sort, knowing that it has a better time complexity, ignoring the nuances of the interactions of merge sort with the arrays you are sorting. The easiest way to overcome this is to have a mental image of how these algorithms function and what they excel at - that is what this paper will do.

For an easier understanding of the analysis, we will be working with a big, running example throughout this paper: a student-teacher database for an university. There are upwards of 10.000 students saved in the database (including students that have finished their studies) and around 300 teachers split into multiple departments.

This paper will contribute a thorough examination of the most basic sorting algorithms, mapping out their use cases in a practical and easy to understand manner. It is assumed that you already know how all the data types and sorting algorithms work, and what other ADS nomenclature utilised in the paper means. The paper will run through a theoretical analysis of all the sorting algorithms, then those theoretical results will be put together with practical results to form a conclusive use case for all of the algorithms.

This paper is an original work, all sources have been properly acknowledged and cited. Only the program utilized in the experimental part of the paper was done with two colleagues.

(Github: `github.com/AlexandraStulianec/Sorting_Algorithms`)

2 Formal Description of Problem and Solution

To see and understand the results of this paper, we must first settle what an algorithm is, furthermore what a sorting algorithm is and then understand what their purpose is in the field of computer science.

"Algorithm.[1]": a procedure for solving a mathematical problem (as of finding the greatest common divisor) in a finite number of steps that frequently involves repetition of an operation.

There are a large variety of algorithms, the term is very broad and encompasses many areas of computer science. An algorithm can be used to do a specific mathematical calculus, to convert text from lowercase to uppercase, much more - it can also be used to sort elements of a given group - a sorting algorithm.

Let us say we have a grouping of some elements that have an order - an array - be they numbers, words, or other types of data; assuming that these elements have a "sorted" state - a state where a specific order is applied to them - we can create a sorting algorithm to go through this array and place the elements in their sorted state. For our case, the sorted state is the state where all the elements are placed in an increasing order.

Now that we know what a sorting algorithm does, why do we have an entire category of such algorithms? Isn't just one sorting algorithm enough for any and all arrays presented? The answer is no, different sorting algorithms parse the data that they are given in different ways: let us define some of these properties and give them names.

Time complexity is the amount of operations that it takes a sorting algorithm to sort an array of elements.

! This is one of the main talking points of this paper, as we want to see which algorithms are the fastest and assess if there are any drawbacks to using them. This subject is a lot more nuanced, as speed alone does not make an algorithm entirely usable - we need to take into account the other properties of the sorting algorithm to better understand its strengths.

Space complexity refers to the memory required to run the sorting algorithm.

! Some sorting algorithms, due to the very big amounts of memory they require to process the information, are not suitable for a lot of cases, even if they have a great time complexity.

Stability in sorting algorithms determines whether these algorithms maintain the same order after sorting elements with the same key.

Comparison-based sorts directly compare the elements one to another to create a final sorted version of the array, meanwhile non-comparison based sorts use other means to place these elements into their sorted position.

Recursive algorithms work by utilising themselves within themselves, each time calling itself with a less significant parameter value.

Adaptability refers to a sorting algorithms ability to skip certain sections of the data.

! Simply put, if it does not need to parse certain parts of the array, it will skip it. This usually leads to certain edge cases, like almost sorted arrays, being processed faster than a random array

These properties are what make each algorithm have its own use case - furthermore, going through these experiments we will see that some of these properties are interlinked, which will allow us to generalize the situations that they can be used in, they can be found in [**.**]

Lastly, the sorting algorithms interact in different ways with the types of lists that you have, here are the ones implemented in the program used to run tests for this paper:

Random An array with elements placed in random order.

Flat A random array with a small sample size of repeating elements.

Reverse sorted A reverse sorted array.

Sorted A sorted array.

Almost sorted A sorted array with two elements swapped out of place.

! We are not going to be covering anything but random arrays in this paper.

Now that we have defined the terms that we will be working with throughout the paper, let us talk about what our problem is and how we will be finding its solution. Our problem is that we do not know what the properties and speeds of the nine aforementioned algorithms are. To solve this, we will be doing a theoretical analysis of the algorithms (analyzing their code to draw some conclusions about their properties) and then we will be doing a practical experiment to see in more detail how they all act.

3 Model and Implementation of Problem and Solution

For the practical part, we will be using Python to create a program that generates random lists that will be sorted through each of the nine sorting algorithms, storing the sort times in a CSV *(comma separated values)* file. Let us take it step by step:

1. Implement all of the sorting algorithms as functions in our program.
2. Create a loop that repeats for the amount of arrays that you want to sort.
3. For each loop, generate a random array to be sorted (based on the array type you want to test).
4. Run a copy of the array through all of the aforementioned functions, making sure to save the time before and after the function. (our program utilizes the `time` library from Python, specifically the function `time.perf_counter_ns`)
5. For each function within that loop, take the difference of the after and before times and add them to a list.
6. After going through all functions, put the list in the output `.csv` file.

7. Clear all variables and lists and repeat this process until the loop stops.

Simply put, our program works as a "stopwatch" for every sorting algorithm, putting the results neatly in a file for us to examine. From there, we can simply take the results and create an average that will allow us to gather a general idea of how each one of the sorting algorithms works.

You can utilize the program yourself and run your own tests, simply hit run, enter the the filename followed by `.csv`, the amount of lists you want to sort and list length, and after a period of processing, you will have a file appear in the project folder which can be opened in any spreadsheet application *(eg. Excel, Google Spreadsheets, etc.)*. From there, you can simply average out the results on each column to create a final "average time complexity" for that specific list length. It is advised that the smaller the list lengths are, the more lists you should sort, for more neutral results.

Another thing to note is the complexity of the entire Python program: the loop runs n times *(n being the amount of lists you want to sort)*, each time doing the work of all nine algorithms. This can be very slow for long arrays - so make sure that you turn off any of the sorting algorithms that you are not interested in testing. Additionally, you can go in the program and change the list type from the default random option to sorted, reverse sorted, flat or almost sorted.

Keep in mind that running these experiments may give vast results - some computers are faster than others. What you need to keep an eye on are the differences in speeds between the different sorting times, meaning how much faster one is compared to another. For example, insertion sort should, by the findings of this paper, be faster than bubble sort by a big margin.

4 Case Studies and Experiment

4.1 Theoretical Analysis

First, to understand the practical results of our experiment, let us look at Table 1. We see that the table is split in three: these are the main sections that we will be dividing the work of the paper into, that is exponential algorithms - algorithms with a time complexity of n^2; logarithmic algorithms - those with a time complexity of $n \log_2 n$; and the non comparison-based algorithms - those that have a time complexity close to n.

Sort	Bubble	Insert	Sel	Quick	Merge	Heap	Count	Radix	Bucket
Time	n^2	n^2	n^2	$n \log_2 n$	$n \log_2 n$	$n \log_2 n$	$n + k$	$d(n + k)$	$n + k$
Space	1	1	1	$\log_2 n$	n	1	k	$n + k$	$n + k$
Stable	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes
Comp.	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Recursive	No	No	No	Yes	Yes	No	No	No	No
Adaptive	Yes	Yes	No	Yes	No	No	Yes	Yes	Yes

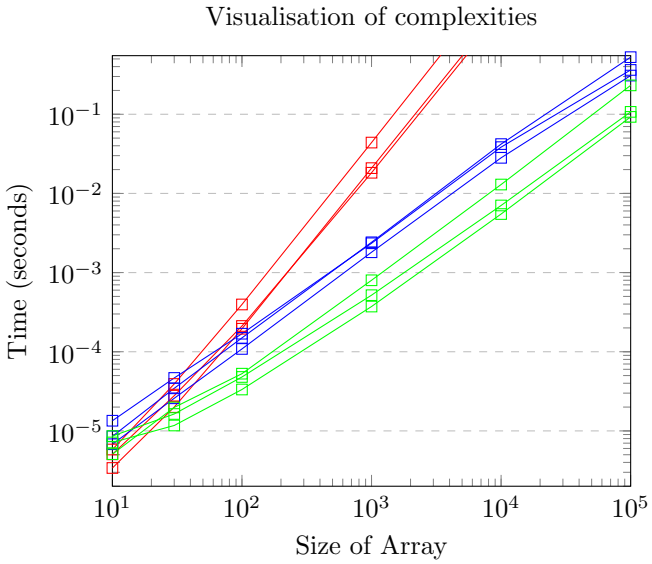
Table 1: Attributes of each sorting algorithm.

Let us put our running example into practice: we notice that, theoretically, the non comparison-based sorting algorithms have almost linear time complexities, meanwhile the exponential algorithms have the best space complexity; the logarithmic ones just sit in the middle, not the best

at one or the other. Given our basic understanding of the properties, we could just use heapsort and not have to worry about running any more tests on our program: heapsort will just sort everything relatively fast. We will see that these results are a bit more nuanced in practice, and more algorithms can be employed to make our student-teacher database faster.

Now, let us get to the meat of this paper: the practical experiments. We understand and have a theoretical reference for how these sorting algorithms should act - that is, how much memory, time and what properties they have, the only thing left is to create some practical sheets of results that we can draw our own conclusions from: starting with the random arrays.

Looking at Figure 4.1 we can see the results of the tests put together side by side. This gives us a very direct point of reference on just how well each of the algorithms performed compared to others on different list lengths.



n^2	$n \log_2 n$	non comp.
red	blue	green

The first thing you will notice is that the exponential algorithms all suddenly spike up at the 100 elements marker, while the logarithmic and non comparison-based ones maintain a steady pace. This can be seen closer towards the end, at the 100k mark, where the exponential algorithms take minutes to give a sorted result, while the logarithmic algorithms stay close to the 1 second mark *(we will see this in more detail in the following subsections).*

This does not mean that exponential algorithms have no use, as we can see at the very bottom, between 10-100 elements, the exponential algorithms actually do a fair job at sorting the arrays, some even outperforming the logarithmic algorithms. This, alongside the small memory consumption that they have makes them great for smaller lists of under 100 elements.

What we can deduce from this is: although we are splitting the sorting algorithms into three categories, the logarithmic and non comparison-based ones seem to have similar results for completely random arrays - already disproving the idea that non comparison-based algorithms are supposed to have close to linear time complexity. On the other hand, we have managed to highlight specifically what exponential algorithms seem to be best at: small lists and small memory consumption. We shall take a deeper look at this in Subsection 4.2.

4.2 Exponential Algorithms' Results

Let us start by noticing the similarities between the exponential algorithms (see Table 2; they all have a relatively similar growth, given their very small workload, the difference in speed between them is close to non-observable in practice, feeling almost instant for the general use cases, this means that any one of them is good for human interaction - sorting a list of elements on a site, sorting items in an inventory, anything that is being observed by a human being and does not have a higher purpose in a program.

Rand	Bubble	Insert	Select
10	$5.82\mu s$	$5.05\mu s$	$3.4\mu s$
30	$39.1\mu s$	$27.8\mu s$	$19.9\mu s$
100	$395\mu s$	$211\mu s$	$195\mu s$
1k	0.04s	0.01s	0.02s
10k	5.15s	1.92s	2.22s
100k	8m 40s	3m 23s	4m 23s

Table 2: Sorting time for exponential algorithms.

To further cement the similarities, all of these are comparison based, and they all have a space complexity of 1, but that is where their similarities stop. If we had to choose a best algorithm purely based on speed, selection sort, as seen in Table 2, making it the best sort if you do not have a clear point of reference of what your array will look like outside of its size. The problem with selection sort is that it is not adaptive, stable and it doesn't scale in time too well.

If you have arrays of unknown lengths that may exceed the 100 elements limit by a larger margin, you may want to consider utilising insertion sort, which has a worse time than selection sort by a small margin, but has the benefits of being adaptive, stable and it scales upwards the best of all three. Bubble sort seems to be the worst, and as far as our tests go nothing about it makes it preferable in its current state.

Within our running example; we may want to implement selection sort for small lists, for example for viewing the teachers from a specific department. We know that we will never really have over 100 teachers in a single department, making it the fastest and most effective choice. If we happen to have to sort more than 100 at a time, then we can simply choose insertion sort, which scales up well and has the benefit of being stable.

Given the nature of exponential sorting algorithms, they are usually best used on devices that have a limited amount of memory to allocate to a sort. Now, if you are working within a bigger-purpose system where the sorting time should be as short as possible and you do not

have a hard limit on the resources available, Subsection 4.3 will see how logarithmic algorithms mitigate the worst-case scenarios that exponential algorithms have to deal with.

4.3 Logarithmic Algorithms' Results

Let us now take a look at the results of the logarithmic algorithms, as seen in Table 3. We see that for small arrays, none of these algorithms do a much better job at sorting small lists than the exponential algorithms, so we will rather look at their ability to sort bigger lists.

Rand	Quick	Merge	Heap
10	$6.78\mu s$	$13.4\mu s$	$8.54\mu s$
30	$25.8\mu s$	$46.6\mu s$	$34.5\mu s$
100	$108\mu s$	$168\mu s$	$149\mu s$
1k	$1807\mu s$	$2346\mu s$	$2412\mu s$
10k	0.28s	0.03s	0.04s
100k	0.3s	0.36s	0.52s

Table 3: Sorting time for logarithmic algorithms.

Let us start by looking at their speed: we easily see that quick sort has the best results. It is adaptive, recursive, and it scales well for small lists, making it a well-rounded sorting algorithm. Its main drawbacks are its somewhat large space complexity, its lack of stability and its high memory usage.

If we are looking for a logarithmic sorting algorithm with a very small space complexity, we can look at heap sort. Scaling it upwards, it is the worst-scaling of the logarithmic algorithms, but its small memory consumption makes it great when you are working with limited space. The problems with heapsort are its lack of stability and adaptability.

Here comes merge sort, the middle child of the two. Its time complexity is slightly worse than quick sort, and so is its space complexity, but it has the power of being stable and it retains the adaptive nature of quicksort - so you are only really sacrificing some time and space to keep your stability.

For our running example, if we want to sort all of the students from a department, we can implement a quicksort - this will very easily sort all of the students in less than a second - as fast as it is, it may not be necessary, seeing how insertion sort has a speed of less than a second anyway - insertion sort would still be good even if we had to sort 10k elements.

Given, if we truly were looking for speed at no cost for memory, we could simply utilize heapsort and have an overall faster sorting speed, but heapsort is more tedious and messy to implement.

4.4 Non Comparison-Based Algorithms' Results

Now comes the final part, the non comparison-based sorting algorithms. As we saw in the theoretical analysis we did in Subsection 4.1 they have the property of being "almost linear," almost

meaning that it highly depends on the type of list that you give it. Let us first take a look at Table 4.

Rand	Count	Radix	Bucket
10	$7.14\mu s$	$5.08\mu s$	$8.6\mu s$
30	$11.7\mu s$	$19.8\mu s$	$16.3\mu s$
100	$33.3\mu s$	$52.8\mu s$	$47.9\mu s$
1k	$372\mu s$	$799\mu s$	$518\mu s$
10k	$5491\mu s$	0.01s	$7044\mu s$
100k	0.09s	0.23s	0.1s

Table 4: Sorting time for non comparison-based algorithms.

As we see, count sort happens to be the fastest of all the algorithms we have encountered until now, its sorting time is an entire order of magnitude faster than the rest - this makes it easily the best one of them all - its time complexity is relatively bad, that is true, but for its performance is it warranted. The problem comes from the lists that you want to sort: given the nature of its space and time complexities larger ranges of numbers will only make it take up more space and time. That combined with its non comparison nature makes it not be as versatile as the logarithmic and exponential algorithms.

This happens to be the case for all of these algorithms, meaning that, while they are highly efficient, they are also highly specific and harder to use on other types of data.

Within our running example, counting sort would be perfect to compute a "leader board" of all the students from a department (or from the whole faculty), by simply taking their average grades and sorting them.

4.5 Conclusion

As our experiments have shown, there is no one-size-fits-all sorting algorithm, we really have to take into consideration what types of arrays we are going to be working with, the systems we will be applying the sorts within and the nature of the problem and minimum performance we want to achieve. Let us quickly go over the best picks from each category, to truly cement the results:

! Exponential algorithms have small space complexities

Selection sort for arrays up to 100 elements

Insertion sort for arrays scaling up, extreme adaptability

! Logarithmic algorithms have great time complexity

Quicksort is the fastest

Heapsort has space complexity of one

Mergesort is stable

! Non comparison-based algorithms have the best time complexity, but are specific

Counting sort has extreme speeds, especially on more flat lists

5 Related Work

Cormen, Thomas H., et al. **Introduction to algorithms.** MIT press, 2022.

Knuth, Donald E. **"The analysis of algorithms."** Actes du Congres International des Mathématiciens (Nice, 1970). Vol. 3. 1970.

Roopa, K., and J. Reshma. **"A comparative study of sorting and searching algorithms."** International Research Journal of Engineering and Technology (IRJET) 5.1 (2018): 1412-1416.

Khan, Mohsin, Samina Shaheen, and Furqan Qureshi. **"Comparative analysis of five sorting algorithms on the basis of best case, average case, and worst case."** Int. J. Inf. Technol. Electr. Eng. 3.1 (2014): 1-10.

6 Conclusions and Future Work

The problem that we have solved is not new, but we have offered a simple and effective approach to find a solution. The advantages of the solution is that it can be recreated, although the results will vary slightly based on the specs of the computer utilized to run the tests. This type of problem solving can be applied to other domains, anything that is time oriented: testing real-time graphics and shaders, testing audio effects real-time latency, and much more. This paper is specific to the field of computer science.

Further exploration of the topic can be done, like testing the space complexity of the algorithms, creating more efficient versions of the algorithms we have tested, testing parallel implementation of the algorithms, and much more.

References

[1]"Algorithm." Merriam-Webster.com Dictionary, Merriam-Webster.