The Relation between Input Complexity, Size and Time Cost in Quick Sort and Insertion Sort

Flavius-Adrian Holerga
West University of Timisoara
Email: *flavius.holerga99@e-uvt.ro*

2019
April

Abstract

We use a mathematical property of permutations to observe and visualize the variations of efficiency and time cost in Quick Sort compared to other non-recursive sorting methods based on the input complexity and size. We then display and discuss the obtained results in the scope of providing verifiable data to sustain the generality of Quick Sort and compare it to other commonly used algorithms.

1 Introduction

This paper discusses the relation between the input complexity, size and efficiency in Quick Sort and other sorting methods, with emphasis on Insertion Sort as the representative algorithm for non-recursive sorting procedures and aims to create a visual representation to sustain the property of generality of the selected sorting method.

The scope of this paper is to illustrate the advantages and disadvantages of using Quick Sort as a general purpose sorting algorithm and visualize the threshold at which, depending on the number of permutations executed on the input vector, Quick Sorted becomes slower than other non-recursive methods. The importance of this problem is raised by the common misconceptions regarding Quick Sort and provides new data to illustrate and depict the special cases in which Quick Sort is a better or worse alternative than other procedures.

To depict the results, in the following paper, we will propose and detail the outcome of a couple of experiments that generate verifiable data to

sustain the generality of Quick Sort and the cases in which other simpler methods are recommended to be used.

Before going into details regarding the previously mentioned property of Quick Sort, we will first take a look at the other commonly used sorting algorithms, observe their efficiency on randomly generated inputs and then introduce the notion of input complexity to further analyze the special cases that may contradict our initial hypothesis. We will also provide some visual schematics to help visualize the discussed methods from an algorithmic point of view.

2 General purpose

The paper aims to visualize the best general purpose sorting algorithm in comparison with other methods, but to do that, we must first provide a formal representation of what it means for an algorithm to be better than another.

From an idealistic point of view, the best sorting algorithm should be able to efficiently sort a list of objects of any kind, since the type of the elements can present itself in different forms (e.g., strings, integers). Even more, it is essential that the algorithm should be stable and as fast as possible when dealing with inputs of both small and large sizes.

In general, non-comparison based algorithms are fundamentally faster than comparison based ones, but lack behind in terms of memory usage, which essentially makes them unfit for general purposes (e.g., sorting big chunks of non-numeric data).

The most common sorting algorithms that fit our general purpose condition, which we will detail in the following pages, are Insertion Sort and Quick Sort, both being comparison based algorithms, but which work in two fundamentally different ways. Even more, the mentioned algorithms present a viable approach to the general sorting problem, though they raise different problems like pivot tweaking in the case of Quick Sort, or data selection in the case of Insertion Sort. Likewise, they are also essentially representatives of their respective class: recursive and non-recursive sorting algorithms.

Other options considered in the following experiments were less effective algorithms such as Bubble Sort and Selection Sort, which are briefly discussed at the end of the paper. To further emphasis the point constructed,

Merge Sort has also been observed as an argument to support the viability of recursive algorithms in sorting.

Furthermore, to understand the difference between the various sorting algorithms, we should also discern the differences between the many areas sorting can be used in and the implications that these given areas raise, therefore a formal description of the properties of sorting is fundamental.

2.1 Properties of sorting algorithms

For the purposes of this paper, we will classify the sorting algorithms by:

- Memory usage (and use of other computer resources)
- Computational complexity (worst, average and best case) in regards to the size of the list
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability (depending on if the elements with equal keys maintain their initial relative order in the list after the execution)
- Adaptability (whether or not the complexity of the shuffling process corresponding to the input affects the running time)

Depending on the task, a sorting algorithm may favor a trade-off in memory for extra stability, or may favor repeating a sequence of avoidable steps to keep the memory usage at a minimum. Situations can vary on a wide spectrum, which is the reason why there exists a large range of algorithms specialized in specific tasks.

Because of this diversity in scope, not much focus has been placed on deciding which algorithm is best suited to keep up with the ever changing need of computation in the sorting field.

2.2 The problem of adaptability

The focus of this paper will mainly involve the problem of *adaptability*. Specifically, Quick Sort presents this conundrum in which depending on the order of the elements from the given list, the efficiency of the algorithm can vary significantly, while on the other hand, other simpler procedures present much more consistent results (e.g., Insertion Sort).

It is important to note that the worst case behaviour in Quick Sort is rather rare, which we will illustrate in the experiments later in this paper.

In order to observe the differences in time cost in relation with the input complexity and input size for the two chosen algorithms, we must first define the notion of **array complexity** as a system to use when conducting experiments.

Method 1 In order to understand the relation between the running time growth function and the relation between the elements from the given array, take the number of inversions in a permutation as a model to differentiate the complexity of an unsorted array.

Let $v(i)$ be the number of inversions for a permutation a_k. Take x_k the corresponding array to the given permutation.

$$a_0 = \begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 1 & 2 & 3 & \cdots & n-1 & n \end{pmatrix}, v(0) = 0;$$

$$a_1 = \begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 1 & 2 & 3 & \cdots & n & n-1 \end{pmatrix}, v(1) = 1$$

$$\vdots$$

$$a_n = \begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ n & n-1 & n-2 & \cdots & 2 & 1 \end{pmatrix}, v(n) = \frac{n(n-1)}{2}$$

To obtain the next array in the sequence, swap the elements in the corresponding arrays:

$$x[n-k] \leftrightarrow x[n-k-1], k \in [1, n].$$

In the experiments which depend on array complexity, we used an algorithm that follows this system to generate inputs for the tests, therefore simulating worst, best and average case behaviour to illustrate the property of generality in Quick Sort with the most important special cases taken into consideration, such that a sorted array would be associated with a permutation with $v = 0$ inversions, and a reversed sequence would have $v = \frac{n(n-1)}{2}$ inversions.

Method 2 Another way to differentiate the complexity of an array x_k is by computing the absolute sum of the differences between the elements and their indexes, such that:

$$d = \sum_{i=1}^{\infty} |x[i] - i|.$$

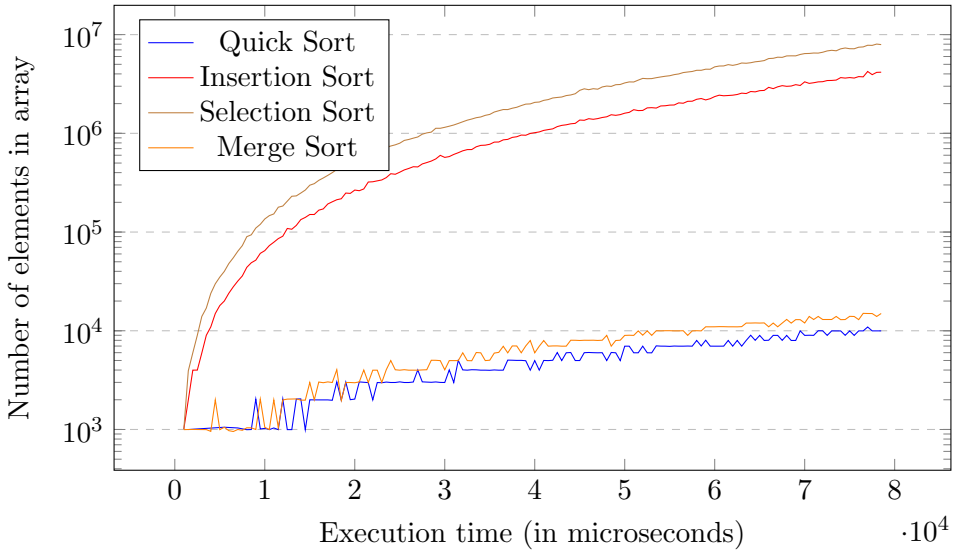
Using this method, similarly to *Method 1*, one of the arrays corresponding to a minimal difficulty value (d) will be the sorted array and the one corresponding to a maximum value will be its reversed sequence.

For the purpose of this paper, we will exclusively use the first method to depict array complexity, but we consider extending the analysis in the future, to also consider this method.

2.3 Average case experiment

This first experiment is going to put emphasis on how the most popular algorithms, and especially the previously discussed ones, perform on a increasing set of natural numbers. For this experiment, we will generate arrays of average complexity $v = \frac{n(n-1)}{4}$, where n is the number of elements in the given array.

Trivial comparison between sorting methods on randomly generated inputs



From the listed algorithms, as expected, Insertion Sort was the fastest non-recursive algorithm and thus will serve as the representative of the non-recursive sorting methods class in the following tests.

Furthermore, it should be noticed that as expected in our initial hypothesis, Quick Sort is the fastest algorithm depicted in our tests on a general case.

This experiment justifies our choice in selecting Insertion Sort and Quick Sort as the focus of this paper.

3 Analysis of complexity

To get an overview of why the results from the first experiment were to be expected, we take a look at the complexity analysis for Insertion Sort and Quick Sort.

3.1 Insertion Sort

Insertion Sort is a simple non-recursive algorithm that sorts a list of objects in place one by one, by taking each element and inserting it in the correct place in the sorted array.

The algorithm itself is much less efficient on large inputs than other more complex algorithms such as Quick Sort or Merge Sort, but gains ground on simplicity and stability, being one of the most reliable and fast algorithms for small chunks of data.

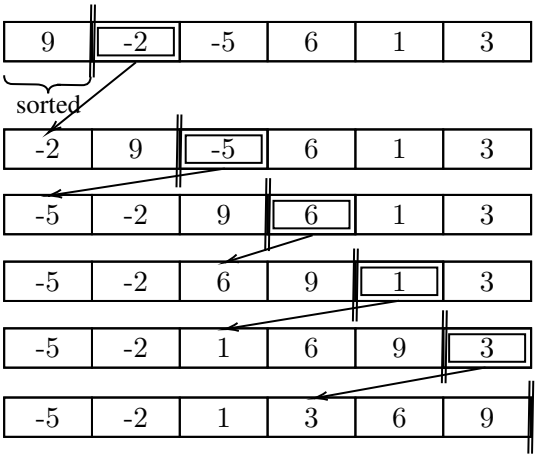


Figure 1: Example of the insertion procedure

In the figure above, we provided an example of how the algorithm works by taking one element and placing it in the right position by shifting the other elements in the sorted subarray to make space for the new object.

3.2 Viability of Insertion Sort

From a viability point of view, the Insertion Sort method is a stable and efficient way of sorting an array of objects of any type in a predictable length of time. Even more, because of its simplicity, Insertion Sort is one of the more accessible and commonly used sorting algorithms.

Worst case: Input in reversed order. The worst case from a running time point of view is when the array complexity (defined previously in the first method) is of a maximum value ($v = \frac{n(n-1)}{2}$):

$$T(n) = \sum_{i=2}^{n} \Theta(i) = \Theta(n^2).$$

Average case: Medium levels of array difficulty:

$$T(n) = \sum_{i=2}^{n} \Theta(i/2) = \Theta(n^2).$$

Unfortunately, even though compared to other sorting algorithms it shines in time efficiency on small inputs, it still is ineffective on inputs of big sizes (as illustrated in our first experiment).

3.3 Quick Sort

Quick Sort is an efficient recursive sorting procedure that is highly adaptable. Published[5] originally in 1961 by British computer scientist Tony Hoare, it is one of the most commonly used algorithms, which when well optimized for a given task, can become twice or even three times as fast as other recursive sorting methods such as Heap Sort and Merge Sort.

Quicksort is a divide and conquer algorithm. It works by dividing an array into two smaller sub-arrays such that every element in the first sub-array is smaller or equal to a chosen pivot element, and the other elements, by consequence, greater than it.

Below, we provided another schematic on the same example discussed when we briefly covered insertion sort, to put emphasis on the different

approach this method takes (compared to other non-recursive procedures such as Insertion Sort).

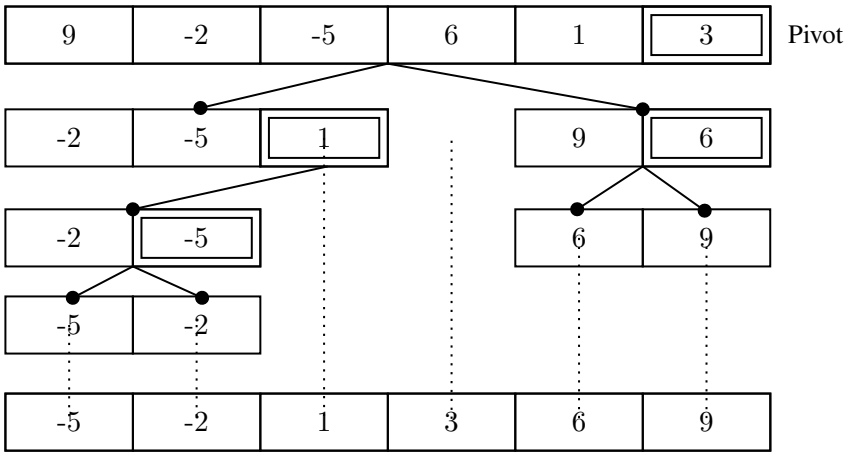


Figure 2: Example of the Quick Sort procedure

3.4 Viability of Quick Sort

The issue of viability is an essential matter to be discussed when talking about Quick Sort, as its the aim of this paper to provide evidence to sustain the property of generality, which is directly related to this matter.

Worst case: Inputs of array complexity $v = 0$ or $v = \frac{n(n-1)}{2}$. The reason for this can be observed by computing the "Worst-case recursion tree":

$$T(n) = T(0) + T(n-1) + \Theta(n).$$

Because of the partitioning step Quick Sort executes, if the number of inversions is minimum or maximum, one side of the partition will always have no elements, thus:

$$T(n) = \Theta(1) + T(n-1) + \Theta(n),$$

$$T(n) = T(n-1) + \Theta(n),$$

$$T(n) = \theta(n^2).$$

We will take this into consideration when illustrating the relation between array complexity and input size in the following experiments, to see how much worse, if at all, this behaviour compares to the behaviour of non-recursive sorting algorithms.

Best case: The partition splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n).$$

Average case: The partition splits the array almost evenly. In practice, this is the most common case.

As long as the split does not follow the worst case, we get:

$$T(n) = T\left(\frac{c_1}{100}n\right) + T\left(\frac{100 - c_1}{100}n\right) + \Theta(n) = \Theta(n \lg n).$$

This property holds as long as $c1 \neq 100$ and $c1 \neq 0$, therefore making worst case behaviour a rather rare case.

Now for the purpose of this paper, we should provide a visual representation on how bad this behaviour affects the running time, if at all, and analyze it in comparison with our proposed representative of non-recursive sorting methods: Insertion Sort as well as illustrate a commonly picked solution in optimizing Quick Sort.

3.5 Probabilistic efficiency

The fact that the average case of Quick Sort has the same complexity as the best case, raises the problem on how to optimize the algorithm in such a way to minimize the number of worst cases.
The solution proved to be partitioning around a **random** pivot. Using this method, the running time becomes independent of the input order and no assumption needs to be or can be made about the splitting distribution and thus no specific input elicits the worst-case behavior.
Let $k = 0, 1, \ldots, n-1$ be the index of a randomly selected pivot.

$$X_k = \begin{cases} 1, \text{ if the partition procedure generated a } k : n-k-1 \text{ split} \\ 0, \text{ otherwise} \end{cases}$$

Since all splits are equally likely,

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n), \text{ if } 0 : n-1 \text{ split} \\ T(1) + T(n-2) + \Theta(n), \text{ if } 1 : n-2 \text{ split} \\ \vdots \\ T(n-1) + T(0) + \Theta(n), \text{ if } n-1 : 0 \text{ split} \end{cases}$$

$$\Rightarrow T(n) = \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))$$

Since it is probabilistically improbable for $X_k = 1$, the average case for Quick Sort becomes $T(n) = \Theta(n \lg n)$. A more complex proof can be argued, but for the scope of this paper we will mainly focus on practical and visual proofs and the impact of the array complexity over the algorithm's efficiency. More details on this issue are as well depicted in [5].

4 Implementation and results

Now that we have established a goal and we have a system through which we can visualize the differences in performance depending on a stable criterion, we used a program to generate inputs of various array complexities and then observed how the many algorithms discussed perform on the various cases.

4.1 Technical details

The following experiments were conducted by calling the different sorting procedures on arrays of a variety of sizes and measuring the running time of these functions. The programming language used was C++ and as a development environment we used: Code::Blocks 17.12 IDE with the commonly used GNU GCC Compiler.
Other specifications: Intel(R) Core(TM) i7-6700HQ CPU @ 2600GHz processor, Windows 10, 64 bit Operating System.

Regarding implementation details, the experiments consist of 2 main parts:

- Setting the environment and generating the input vectors,
- Sorting copies of the vectors created in the first part and storing the execution times in a file, for it to be processed later.

The results are depicted in microseconds and mostly visualized on a logarithmic vertical axis as the differences in the procedure's time cost growth functions are considerable.

4.2 Visual representation of generated data

Using the method described previously when the problem of adaptability was discussed, we generated a number of arrays with increasing array complexity values. We then measured the time it takes for the algorithms to finish executing and repeated the process for different input sizes.

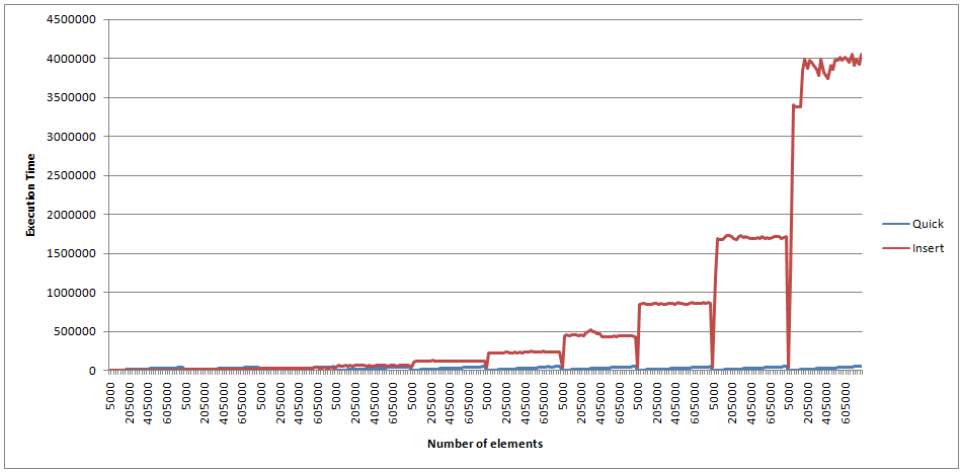


Figure 3: Overview of the whole experiment in regards to array complexity

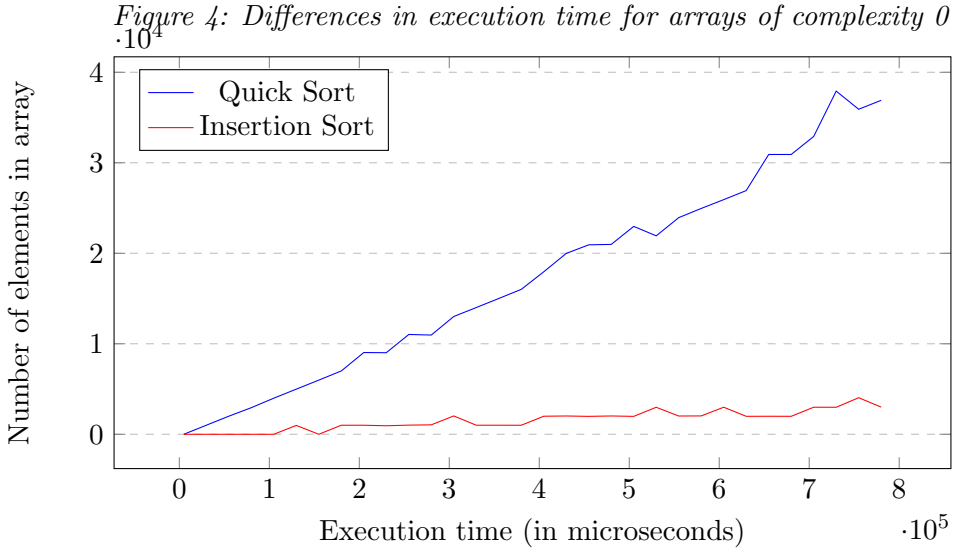
In *Figure 3* are depicted the cycles generated of increasing n elements on different levels of array complexity for the selected sorting algorithms (where $n \in [500, 10^6]$). This graph serves the purpose of presenting a general idea on how much array complexity affects the performance of an algorithm.

We use *cycle* to refer to the interval of time where the number of elements in the tested array increases on a constant number of inversions in the permutations corresponding to the given array. During the test, every time we reached a million elements, the array complexity value increased and the tests were reiterated for different values of n.

To give a more simple interpretation of the results, the cycles reflect the relation between array complexity and input size, such that for the same amount of elements in the array x_k, the running time is visibly influenced based on the number of inversions of the corresponding permutation.

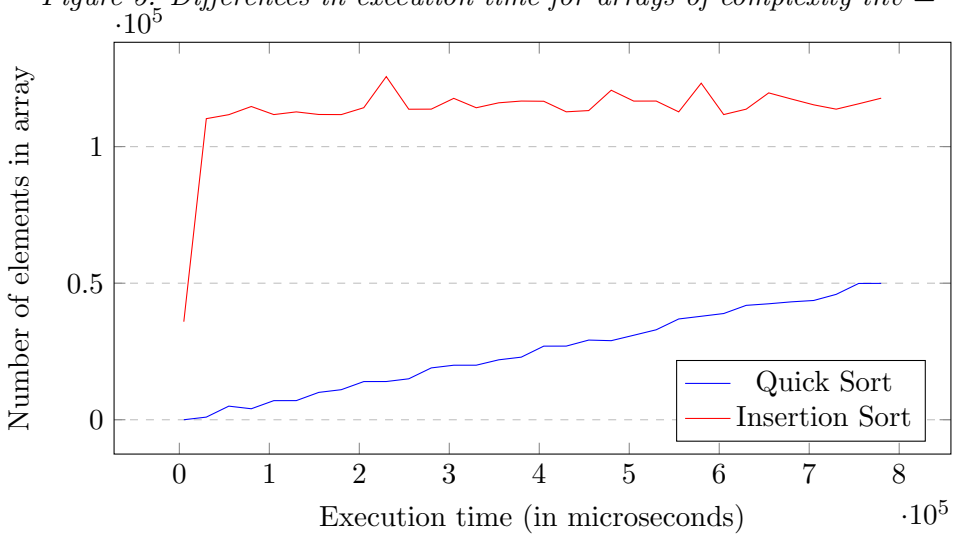
When the level of difficulty is 0 (the array is sorted before calling the sorting procedures), Quick Sort with centered pivot lacked behind Insertion Sort in terms of speed for very small input sizes. A more detailed graph containing these results can be seen in *Figure 4*.

Figure 4: Differences in execution time for arrays of complexity 0



Once the complexity level is increased, Quick Sort rapidly takes precedent over Insertion Sort. For example, in *Figure 3*, during the 5th cycle, this becomes evident. A close up of the 5th cycle with complexity $inv = \frac{n(n-1)}{2^{11}}$ can be analyzed in *Figure 5* below.

Figure 5: Differences in execution time for arrays of complexity $inv = \frac{n(n-1)}{2^{11}}$



It seems that the more we distance ourselves from extreme points in terms of array complexity, the more viable Quick Sort becomes compared to other methods. Further more, in these experiments a right sided pivot version of Quick Sort was used, which only further points to the conclusion that Quick Sort can be considered as one of the, if not the fastest general purpose sorting algorithm.

Furthermore, this differences could be observed only on arrays of very small input sizes. Once the input size is big enough, we observed that Quick Sort, even in a Worst Case scenario is still able to outperform a non-recursive sorting algorithm such as Insertion Sort.

In a final test we included the other sorting methods on cycles of different input sizes and decreasing levels of array complexity.

During each cycle, it can clearly be observed the running time decrease as the array complexity decreases for each sorting algorithm. The most noticeable differences can be seen for the sorting methods which have the worst case behaviour radically different than the best case behaviour(e.g., Bubble Sort).

The results can be observed in *Figure 6* below. Because of the extreme difference in execution time, we completely removed Quick Sort and Merge Sort from this illustration as they are on a whole different level compared to the non-recursive procedure and thus we hope to emphasis the effect array complexity has on all sorting algorithms, not just the ones discussed.

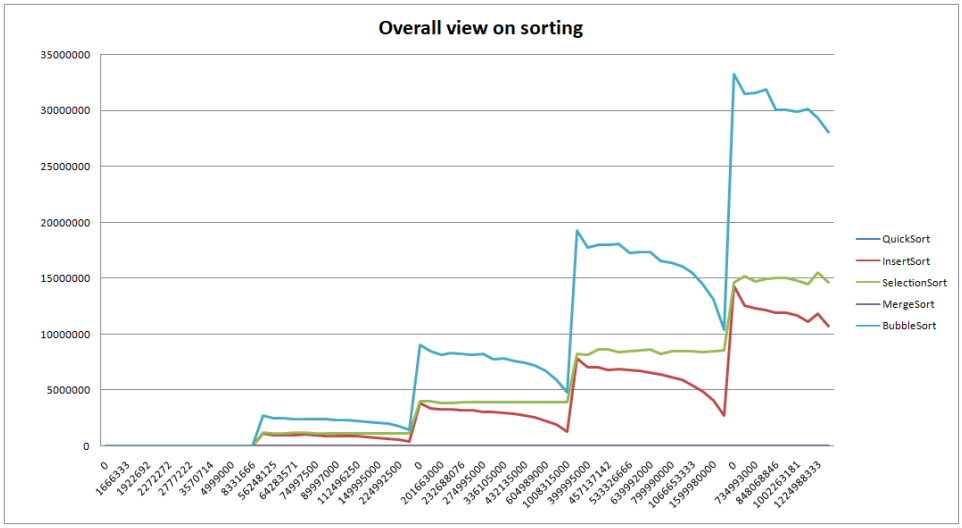


Figure 6: Overall view of sorting algorithms in regards to array complexity

5 Related work

As the topic of this paper is of great importance, all technical details were impossible to cover on such a vast topic. Furthermore, a lot of the computations needed to compute the visual representations showcased in this paper are trivial and such, were not covered explicitly in the contents of this article. Other pieces of work regarding the subject previously discussed that helped us obtain a clear picture on the matter were basic notions of algorithmics and data structures which are explained in more details in [2].

Besides trivial knowledge, in the literature, there exist various researches which have been conducted with the scope of evaluating performance or complexity between the various sorting algorithms. In [1], an evaluation of Median, Heap, and Quick Sort was conducted similarly to the experiments conducted in this paper, but in regards to CPU time and memory, such that most of the focus of the paper has been placed on the physical performance of the procedure, while this paper aims to visualize the particular cases for which the recursive sorting methods should be left out in the favor of non-recursive ones such as Insertion Sort. Similarly [3] covers also mentioned algorithms such as Merge Sort and again Bubble and Insertion Sort. Another piece of literature that covered Merge Sort in detail is [4], which introduces the algorithm as a divide-and-conquer method. Other research conducted put emphasis on Heap Sort as being the most efficient recursive sorting algorithm in an area more limited and Bubble Sort as the least effective non-recursive commonly used sorting procedure.

6 Conclusions and future work

In conclusion, we tried to create a simplified visual representation of time complexity in relation with input complexity and size and observed that for our purposes, Quick Sort is a viable option to consider when looking for a general purpose sorting algorithm. Furthermore, we observed that the worst case behavior of Quick Sort is elicited in very rare cases which can be minimized for general purpose by randomizing the pivot element.

We realize that even if Quick Sort has proved as a viable solution to our problem, other sorting methods with more specific applications do exist and have the potential to overshadow the methods depicted in this paper, but due to time constraints, we were unable to further explore these various cases. In the future, we also aim to consider the other method to

depict array complexity and analyze the consistencies or inconsistencies in the results obtained by comparing the discussed sorting algorithms through other lenses.

References

[1] Adesina, Opeyemi. (2013). A Comparative Study of Sorting Algorithms. African Journal of Computing. ICT. 6. 199-206. https://www.researchgate.net/publication/288825600_A_Comparative_Study_of_Sorting_Algorithms

[2] Charles Leiserson, and Erik Demaine. 6.046J Introduction to Algorithms (SMA 5503). Fall 2005. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu. License: Creative Commons BY-NC-SA.

[3] Pasetto D., Akhriev A., "A comparative study of parallel sort algorithms", IBM Dublin Research Laboratory, Dublin15, Ireland.

[4] Qin M., "Merge Sort Algorithm", Department of Computer Sciences, Florida Institute of Technology, Melbourne, FL 32901.

[5] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. Commun, ACM 4, 7 (July 1961). http://dx.doi.org/10.1145/366622.366644