The algorithm showdown: A **theoretical** and **experimental** comparison of popular sorting algorithms

Raul-Andrei Ariton
Department of Computer Science,
West University of Timișoara
raul.ariton05@e-uvt.ro

Abstract

Perhaps one of the most fundamental problems in Computer Science, algorithms that solve the problem of sorting are critical to anyone starting out in the area of Algorithms and Data Structures. They are a showcase of clever algorithmic thinking, as well as an application of common algorithm design techniques.

This research paper compares popular sorting algorithms such as Bubble Sort, Selection Sort, QuickSort, and more, using a theoretical as well as a practical, experimental approach.

Generally, this paper aims to emphasize that there is no "superior" sorting algorithm and that each algorithm has its ideal use case, its strengths and weaknesses. Such weaknesses are discussed and showcased.

The findings of this paper aim to guide developers, researchers in understanding the strengths and weaknesses of elementary sorting algorithms, as well as to showcase a few use cases for them.

Contents

Part I
Introduction

1 An everyday sorting problem

Consider the following problem:

> I have a T-shirt for every day of the week (7 *T-shirts*). I decide to wash them all together at the end of the week.
>
> How can I put them back in the closet, in the order I am going to wear them (*ascending order of numbers*), in the **shortest time possible**, and using the **least amount of space** possible (*separating groups of T-shirts as little as possible*)?
>
> **Note.** *For analogy purposes, assume each T-shirt is laid facing-down, and its number can only be seen if picked up. Only one T-shirt can be picked up at a time.*

The above is an over-simplified example of a **sorting problem** that an average person might encounter in everyday life.

1.1 Possible solutions

- One solution is the so-called "brute-force" approach:
  Throw the pile of $n = 7$ T-shirts in the air, and lay them out, eventually checking if they are sorted.

  This sorting algorithm is called **BogoSort**, and has an average-case complexity of $\Theta(n \times n!)$[1], which in the case of our example evaluates to $\Theta(n \times n!) = \Theta(7 \times 7!) = \Theta(7 \times 5040) = \Theta(35280)$. Which means, one would have to repeat the process on average 35280 times. *Not to mention the fact that this process could repeat forever.*

Declaration of originality

I, Raul-Andrei Ariton, under the West University of Timișoara hereby present my research paper, titled *The algorithm showdown: A theoretical and experimental comparison of popular sorting algorithms*, and I declare that it is a product of my own research and investigation.

The experiment process is done completely by myself, thus the experiment results are all original and not fabricated. Moreover, the graphs, that visually represent my experiment results are all created using my code and mine only.

2 Reading Instructions

For the rest of the paper, the formal definition of the sorting problem will be presented, as well as various solutions of this problem (sorting algorithms).

For each solution, a general (*pseudocode*) implementation of the solution will be presented.

Afterwards, all solutions will be compared under various test cases (*input data types*), in search for a general, well-performing sorting algorithm.

3 The problem of sorting in Computer Science

Consider the following problem:

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
Output: A permutation (reordering) $\langle a_1^{'}, a_2^{'}, \ldots, a_n^{'} \rangle$ of the inputs such that

$$a_1^{'} \preceq a_2^{'} \preceq \cdots \preceq a_n^{'}$$

where $\preceq$ is an order relation (predicate).

The above describes the sorting problem in Computer Science[2].

3.1 Elements of a sorting problem

A sorting problem may consist of[2]:

- The **record**, the collection of data to be sorted.

  In the theoretical analyses of algorithms, the record is considered to be an array. In the implementations and experiments, however, Python lists will be used, which are essentially dynamic arrays, *which grow along with the insertion of elements.*

- The **key**, the value to be sorted, or in other words, the value that determines the reordering of the record.

  In the theoretical analyses of algorithms, the keys are considered to be numerical values (numbers). In the experiments, however, various types of keys will be used.

- The **satellite data**, which are values carried around with the key, but which do not determine the reordering of the record. In practice, when reordering keys, a sorting algorithm must permute the satellite data along with its respective key.

  The algorithm implementations in this paper are not implemented for such cases. Entries in the record consist purely of keys (the values that will be permuted)

3.2 Solutions to the sorting problem

The sorting problem in Computer Science is solved using sorting algorithms. Generally, an algorithm must be:

- Correct, which means:

  - its input satisfies the problem's preconditions[1][3]
  - it terminates after a finite number of steps
  - its output satisfies the problem's postconditions[2][3]

- Efficient, i.e. it needs to utilize a *reasonable* amount of resources (*time, space in memory*).

A note on space complexity. This paper focuses mainly on comparing algorithms based on their execution time rather than the space they use in memory.

This is rather a personal choice, as machine memory as a resouce is getting more and more affordable as a result of many impressive advances in the area of Computer Architecture.

Additionally, time is a much more important resource to account for when analyzing and implementing algorithms.

Wherever possible, mentions of the algorithm's space complexity will be made.

Part II
The Algorithms

Classification of sorting algorithms

Below follow some terminologies that not only allow better organization of this paper, but also serve as a guide to researchers as well as learners to differentiate between the different types of sorting algorithms. Special acknowledgments to [4] for the extensive coverage on this topic.

A sorting algorithm can be categorized based on the following criteria:

- Comparison sorting or Non-Comparison sorting
  - Comparison-based algorithms sort using a comparison operator between two elements in the array, which is defined using a function (*either built-in or user-defined*). The best known time complexity of a comparison-based sorting algorithm is $O(n \log n)$.
  - Algorithms which are not based on comparison of elements sort based on assumptions about the input data, thus can only be applied to input data which satisfy specific conditions. In a very ideal case, a non-comparison sorting algorithm can achieve a complexity of $O(n)$.
- Stability In the case where two values in the array are equal, their relative order, i.e. the order in which they appear in the original array order, must be preserved to consider a sorting algorithm stable.

  If this relative order is not preserved, the algorithm is unstable, i.e. not stable.
- In-place or Out-of-place sorting

[1]conditions that the input data of the problem must satisfy.
[2]required properties of the problem's output.

  - An algorithm sorts in-place if only a constant number of elements of the input array get stored outside the array[2].
  - An algorithm sorts out-of-place if it makes copies (also called *buffers*) of the array, or parts of it, and performs the sorting process on those copies, and not directly on the original array in memory.

- Recursive or Iterative
  - A function is called *recursive* if it calls itself finitely or infinitely many times.
  - Algorithms which do not feature recursion are called *iterative*. Such algorithms sort at once (within one function call) by using iterative statements or loops.

1 Bubble Sort

Main idea. Starting with the first element of the array up until the last one, adjacent elements are compared and swapped if they are out of order.

Implementation

Algorithm 1 Bubble Sort

```
function BUBBLE-SORT($arr[1, \dots, n]$)
    for $i = 1$ to $n$ do
        for $j = 1$ to $n - i - 1$ do
            if $arr[j] > arr[j + 1]$ then
                SWAP($arr[j], arr[j + 1]$)
            end if
        end for
    end for
end function
```

Complexity and classification. Bubble Sort is a stable, iterative, in-place sorting algorithm. In the worst-case scenario, its time complexity is $O(n^2)$, as well as a complexity of $\Theta(n^2)$ in the average-case. It has a worst-case space complexity of $O(1)$.

2 Insertion Sort

Insertion sort is similar to the way one would sort playing cards. It performs well on small, mostly-sorted input arrays.

Main idea. The algorithm sorts the first two elements in the input array, then the third element in relation to first two, and so on until all the array elements are sorted[5].

Complexity and classification. Insertion Sort is a stable, iterative, in-place sorting algorithm. In the worst-case scenario, its time complexity is $O(n^2)$, as well as a complexity of $\Theta(n^2)$ in the average-case. It has a worst-case space complexity of $O(1)$.

Implementation

Algorithm 2 Insertion Sort

```
function INSERTION-SORT($arr[1, \dots, n]$)
    for $i = 2$ to $n$ do
        $aux = arr[i]$
        $j = i - 1$
        while $j \geq 1$ and $arr[j] > aux$ do
            $arr[j + 1] = arr[j]$
            $j = j - 1$
        end while
        $arr[j + 1] = aux$
    end for
end function
```

3 Selection Sort

Main idea. Starting with the first element in the array, for each element at position i, search for the minimum value in the sub-array $arr[i, \dots, n]$.

If that value is less than the current element $arr[i]$, swap them.

Repeat the process $n - 1$ times, because the element $arr[n]$ will be naturally sorted.

Implementation

Algorithm 3 Selection Sort

```
function SELECTION-SORT($arr[1, \dots, n]$)
    for $i = 1$ to $n - 1$ do
        $min\_index = i$

        for $j = i + 1$ to $n$ do
            if $arr[j] < arr[min\_index]$ then
                $min\_index = j$
            end if
        end for

        if $arr[min\_index] < arr[i]$ then
            $swap(arr[i], arr[min\_index])$
        end if
    end for
end function
```

Complexity and classification. Selection Sort is an iterative, in-place sorting algorithm. It is not stable. In the worst-case scenario, its time complexity is $O(n^2)$, as well as a complexity of $\Theta(n^2)$ in the average-case. It has a worst-case space complexity of $O(1)$.

4 Merge Sort

Merge Sort is often used as an introduction to the “divide and conquer” technique/paradigm. It makes use of this approach to solve the sorting problem, by dividing the problem (the array to be sorted) into smaller subarrays, then conquering (sorting and merging) subarrays until the final array is sorted. It is known to be inefficient in terms of memory space used.

Main idea. The input array is split into two bisects, which then are split using a recursive call of the MERGE-SORT function, and finally merged and sorted using the MERGE procedure. Figure 1 features an illustrated application of Merge Sort.

Implementation

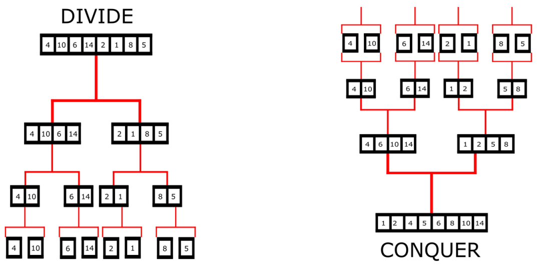

Figure 1: Merge sort in action.

Algorithm 4 Merge Sort

```
function MERGE-SORT($arr[1,\dots,n]$)

    if $n \leq 1$ then
        return $arr$
    end if

    $mid = \left\lfloor \frac{n}{2} \right\rfloor$
    $left = arr[1,\dots,mid]$
    $right = arr[mid+1,\dots,n]$

    $left =$ MERGE-SORT($left$)
    $right =$ MERGE-SORT($right$)
    MERGE($left, right, arr$)

    return $arr[1,\dots,n]$
end function
```

Complexity and classification. Merge Sort is a stable, recursive, out-of-place sorting algorithm. In the worst-case scenario, its time complexity is $O(n \log n)$, as well as a complexity of $\Theta(n \log n)$ in the average-case. It has a space complexity of $O(n)$.

```
procedure MERGE($arr[1,\dots,n], left[1,\dots,l\_size], right[1,\dots,r\_size]$)
    $l\_counter = 0$
    $r\_counter = 0$
    $i = 0$

    while $l\_counter < l\_size$ and $r\_counter < r\_size$ do
        if $left[l\_counter] < right[r\_counter]$ then
            $arr[i] = left[l\_counter]$
            $i = i + 1$
            $l\_counter = l\_counter + 1$

        else
            $arr[i] = right[r\_counter]$
            $i = i + 1$
            $r\_counter = r\_counter + 1$
        end if
    end while

    // In some cases, elements from either the left or right subarrays will be
       left unplaced in the final array. We ensure they are all placed using
       while loops.

    while $l\_counter < l\_size$ do
        $arr[i] = left[l\_counter]$
        $i = i + 1$
        $l\_counter = l\_counter + 1$
    end while

    while $r\_counter < r\_size$ do
        $arr[i] = right[r\_counter]$
        $i = i + 1$
        $r\_counter = r\_counter + 1$
    end while

end procedure
```

5 Quick Sort

Quicksort, hence its name, is known as a fast, general purpose sorting algorithm[5]. It is widely used in real life applications.

Main idea. First, a pivot element is chosen. It can be any element in the array, but in our implementation, we select it as the last element.

Then using a subroutine PARTITION, the pivot element is moved to the position it will have in the final sorted array, i.e. it gets sorted. All elements before it have smaller values and all elements after it have values greater or equal to it.

Thus, the array gets divided into *two* subarrays (assuming $n \geq 4$), which can be sorted independently, without a need for swapping elements between subarrays.

Now, the process repeats recursively. In each sub-array the pivot element is chosen, then moved to its final sorted position, and once again the sub-array gets divided.

Implementation

Algorithm 5 QuickSort

```
1: function QUICKSORT($arr[], start, end$)

2:     if $start \geq end$ then
3:         return
4:     end if

5:     $pivot$ = PARTITION($arr, start, end$)
6:     QUICKSORT($arr, start, pivot - 1$)
7:     QUICKSORT($arr, pivot + 1, end$)

8: end function
```

```
9: function PARTITION($arr[], start, end$)
10:     $pivot = arr[end]$
11:     $i = start - 1$
12:     for $j = start$ to $end - 1$ do
13:         if $arr[j] \leq pivot$ then
14:             $i = i + 1$
15:             SWAP($arr[i], arr[j]$)
16:         end if
17:     end for
18:     SWAP($arr[i + 1], arr[end]$)
19:     return $i + 1$
20: end function
```

Complexity and classification. QuickSort is a stable, recursive, out-of-place sorting algorithm. In the worst-case scenario, its time complexity is $O(n^2)$, as well as a complexity of $\Theta(n \log n)$ in the average-case. In the worst-case, it has a space complexity of $O(\log n)$.

Part III
Case Study

1 Methodology

In this part of the paper, each sorting algorithm will be put to the test, by being given various of input arrays to sort. The input arrays will vary in sorting configuration (*random, sorted, flat list*[3]).

The aim of this experiment is to showcase the strengths and vulnerabilities of each sorting algorithm.

Technical details. For the conduction of these tests, including measurements and data visualization, I used the Python programming language.

All the code, including the algorithm implementations, is publicly available on `https://github.com/raul-it0o0/the-sorting-showdown`.

2 Results

As expected, the graphs for Bubble Sort and Selection Sort resemble a $f(n) = n^2$ graph, where n is the input array size.

[3]We define flat list as a list (i.e. an array) with few unique values that appear repeatedly in the list (e.g. a list with many ones, twos, and/or threes)

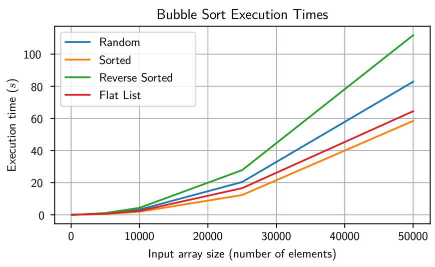

Figure 2: Experiment results on the Bubble Sort algorithm.

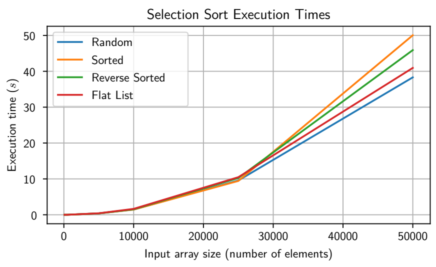

Figure 3: Experiment results on the Selection Sort algorithm.

A small observation would be that Selection Sort sorts a bit faster; In some cases (e.g. in the case of a *reverse sorted* array with 50000 elements), it sorts in approximately *half* the time.

Another observation is that Bubble Sort sorts an unsorted (i.e. *random*) array faster, as opposed to sorting an already sorted array.

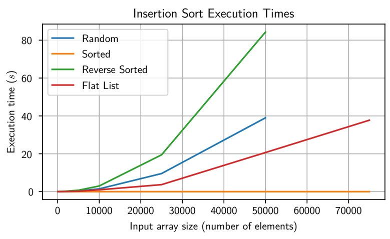

Figure 4: Experiment results on the Insertion Sort algorithm.

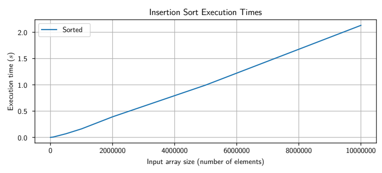

Figure 5: Results of Insertion Sort sorting a pre-sorted array

Insertion Sort comes as a surprise. It sorts a *reverse sorted* array the worst, as one would expect.

However, given an *already sorted* array, it sorts in **linear** time, as seen in figure 5. The same behavior can be observed with an *almost sorted* array. Skiena in [3] explains that these behaviours are a result of the condition in the inner **while** loop. For better context, see algorithm 2. In *almost sorted* or *sorted* arrays, the algorithm exits out of this while loop quite quickly, since there already exists a sorted order.

This unique ability aside, Insertion Sort is not often used for real-life sorting scenarios as it does not sort as well in other array configurations.

However, when one knows with confidence that the input array is at least *nearly sorted*, Insertion Sort might be a good option, considering its relatively easy implementation as well.

Conveniently, Merge Sort does well with all sorting configurations, with a graph resembling $f(n) = O(n \log n)$.

However, one observation is that Merge Sort sorts *slightly* slower with small input array sizes, compared to other algorithms (Bubble, Selection, Insertion). This

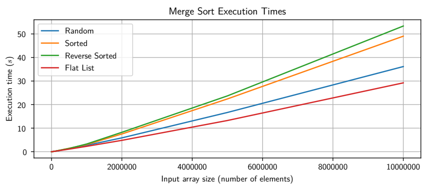

Figure 6: Experiment results on the Merge Sort algorithm.

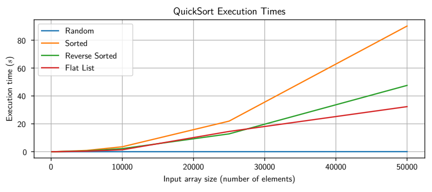

Figure 7: Experiment results on the QuickSort algorithm.

is due to its *divide and conquer* approach, which as a result causes a lot of *dividing* and *merging*. In small datasets these tasks outweigh the actual sorting process, which causes this increase in execution time.

Overall, Merge Sort should be used when a dataset is known to be large. For smaller datasets, it is better to use simpler sorting algorithms.

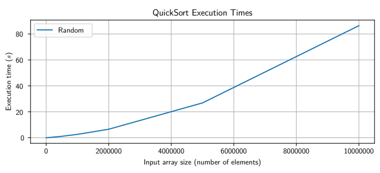

Figure 8: Results of QuickSort on a randomly sorted array

As it turns out, QuickSort is not as quick as some claim it to be. In special cases, where the array is not guaranteed to be *randomly ordered*, its execution times resemble those in the case of Insertion Sort.

However, the most alarming observation is the spike in execution time in the case of *already sorted* arrays.

As QuickSort depends heavily on the PARTITION process for dividing and sorting, in such cases of *sorted* arrays, the partitioning becomes **unbalanced**, in turn causing recursion calls relative to the input array size. The large finitude of recursion calls causes excessive memory use.

Generally, QuickSort is reliable for arrays whose properties are not known. In the cases where there exists some knowledge on how the array is pre-ordered, one should reconsider implementation and **measure**, as we have observed earlier.

3 Related Work

The search for sorting algorithms is an open problem in Computer Science; That is, the search for algorithms suited for a particular situation rather than the search for one “superior, above-all” sorting algorithm (*which does not exist*).

Many other works exist on comparison of sorting algorithms. Some present optimizations on the algorithms presented in this paper, while others take into heavy consideration the space complexity of each algorithm.

- **Karunanithi, Ashok kumar. “A Survey, Discussion and Comparison of Sorting Algorithms.” (2014).**

  Karunanithi presents many sorting algorithms in his work, as well as an elementary explanation to how each of them work.

- **Estivill-Castro, Vladmir, and Derick Wood. "A survey of adaptive sorting algorithms." ACM Computing Surveys (CSUR) 24.4 (1992): 441-476.**

Estivill et al. conduct a much more in-depth experiment on sorting algorithm performance, and offer their own approaches for even faster sorting methods.

Part IV
Conclusions and Future Work

The *superior* sorting algorithm

In this paper I have attempted to compare elementary sorting algorithms in terms of execution time. I used arrays of integers increasing in size to showcase how execution time is proportional to input array size.

No sorting algorithm is perfect, as we have observed throughout our experimentation. Each has its strengths and weaknesses, depending on the type of sorting problem it is tasked to solve.

1 Future Work

This paper has only covered the basics of the world of sorting. Here are a few open problems which I aim to address in future work:

- What other factors affect sorting?

  Does the data type (*integer, character, string*) of the key affect the sorting process? If so, how?

- Improvements to the presented algorithms

  Many improvements have been proposed for all the algorithms presented in this paper, many which are designed to perform exceptionally well in very specific conditions.

References

[1] H. Gruber, M. Holzer, and O. Ruepp, "Sorting the slow way: An analysis of perversely awful randomized sorting algorithms," vol. 4475, Jun. 2007, pp. 183–197, ISBN: 978-3-540-72913-6. DOI: 10.1007/978-3-540-72914-3_17.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, third edition*, en. MIT Press, 2009, p. 1314, ISBN: 9780262033848.

[3] S. S. Skiena, *The Algorithm Design Manual*, en. Springer, 2010, ISBN: 9781849967204.

[4] R. Ali, "Sorting and classification of sorting algorithms," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 14, pp. 5920–5924, 2021.

[5] A. Bharadwaj and S. Mishra, "Comparison of sorting algorithms based on input sequences," *International Journal of Computer Applications*, vol. 78, no. 14, 2013.