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 showcased, and moreover potential optimizations are discussed, in order to ensure algorithm reliability and stability.

The findings of this paper aim to guide developers, researchers in choosing the algorithm that suits their sorting problem the best.

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.
>
> After they are all washed and dried, I have to put them back in the closet, in the order I am going to wear them (*ascending order of numbers*).
>
> How can I do this 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.

In short, the sorting problem in computer science consists of organizing a collection of data, called *keys*, into a specific order (*in our case, in ascending order of numbers*).

1.1 The importance of sorting

Although at first, sorting might seem like a rare problem one might encounter, wether that be in their everyday life or in software development, there is a reason why it is considered the most fundamental problem in the study of algorithms:

- A Sorting is essential in data analysis, as it helps data scientists/analysts organize, find, visualize and understand data in order to make better, reliable decisions
- Many algorithms use sorting as a key subroutine, or require a sorted data structure as a prerequisite[1].

  Referring back to the T-shirt problem, suppose one has to search for the T-shirt number x, $1 \leq x \leq 7$. Assuming the T-shirts are **sorted** in increasing order, the Binary Search algorithm can be applied, with a worst-case scenario runtime of $O(\log n) \overset{n=7 \text{ (days)}}{=} O(\log 7) \approx O(3)$, which means the search would require at most 3 steps, depending on the value of x.
- Since there is no "superior" sorting algorithm that can be used and relied on for any and every sorting problem, knowledge of various sorting algorithms and their respective strengths and weaknesses is critical.

  The fastest sorting algorithm for a specific situation depends on many factors, such as the input data type (*key type*), how the data is stored (the *record*), prior knowledge about the data to be sorted[1], and many more.

- The vast catalog of sorting algorithms that exist today employ a rich set of techniques. In fact, teaching various sorting algorithms also teaches various algorithm design techniques, along with algorithmic thinking.

  This paper will discuss such design techniques in each case.

1.2 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.

  If they are sorted, celebrate.

  If not, repeat until they become sorted.

  This sorting algorithm is called **BogoSort**, and has an average-case complexity of $\Theta(n \times n!)$[2], 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.*

- Another solution is laying out each T-shirt, and swapping each T-shirt with its neighbouring one, repeating the process $n = 7$ times, until the T-shirts are laid out in ascending order.

  This sorting algorithm is called **Bubble Sort**, and it will be explored in further detail later in this paper. It has a worst-case time complexity of $O(n^2)$, which in the case of our example evaluates to $O(n^2) = O(7^2) = O(49)$. Which means one would have to repeat the process at most 49 times.

As expected, these approaches do not ensure the shortest execution time.

Transitioning to formal definitions. Moving from a simple analogy to more complex, formal definitions and processes, we shall explore more time-efficient solutions to our sorting problem, as well as experimenting with larger, more complex types of inputs to be sorted.

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. Wherever deemed necessary, and to the best of my abilities, I have referenced and thus acknowledged the sources where I have derived ideas or extracts. Furthermore, I state that this paper has not been submitted anywhere else for publication.

The experiment process is done completely by myself, thus the experiment results are all original and not fabricated. Moreover, the plots, 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 short theoretical analysis will be conducted, proving its corectness, termination as well as its execution time complexity.

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

Throughout this paper, solutions will be categorized based off of the techniques they use, and for each such category, all of its solutions will be compared under various test cases (*input data types*), to then declare the most performant solution in that category.

Finally, all "winner" solutions of their respective category will be compared, in order to declare a "final winner", i.e. solution which generally performs well in any test case.

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[1].

3.1 Elements of a sorting problem

A sorting problem may consist of[1]:

- 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

[1] conditions that the input data of the problem must satisfy.

  - 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.

However, 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.

Now that the formalities have been defined and explained, we shall move on to categorizing sorting algorithms and analyzing them one-by-one.

[2] required properties of the problem's output.

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

  Few sorting algorithms are naturally stable, although most of the time any unstable sorting algorithm can be modified in order to become stable[3].
- In-place or Out-of-place sorting
  - An algorithm sorts in-place if only a constant number of elements of the input array get stored outside the array[1]. In practice, no copies of the array values are made outside of it, and the sorting is performed directly on the original array in memory.
  - 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 feature recursive behavior are based on the *divide and conquer* algorithm design technique.

    Usually, recursion is done to divide the problem into subproblems, thus minimizing execution time.
  - Algorithms which do not feature recursion are called *iterative*. Such algorithms sort at once (within one function call) by using iterative statements or loops.

- Adaptive or Non-Adaptive
  - A sorting algorithm is called adaptive if in the case of an input array with existing order, it sorts faster.

    In real-life cases, most sequences to be sorted already have somewhat of an order, therefore such a feature is very appealing for a sorting algorithm.
  - If an algorithm does not take advantage of an input array with an existing order, it is called non-adaptive.

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

```
1: function BUBBLE-SORT(arr[1, ..., n])
2:     for i = 1 to n do
3:         for j = 1 to n − i − 1 do
4:             if arr[j] > arr[j + 1] then
5:                 SWAP(arr[j], arr[j + 1])
6:             end if
7:         end for
8:     end for
9: end function
```

Corectness and complexity analysis

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].

Implementation

Algorithm 2 Insertion Sort

```
1: function INSERTION-SORT(arr[1, ..., n])
2:     for i = 2 to n do
3:         aux = arr[i]
4:         j = i − 1
5:         while j ≥ 1 and arr[j] > aux do
6:             arr[j + 1] = arr[j]
7:             j = j − 1
8:         end while
9:         arr[j + 1] = aux
10:    end for
11: end function
```

Corectness and complexity analysis

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 $arr[i, \ldots, n]$.

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

```
1:  function SELECTION-SORT(arr[1, ..., n])
2:      for i = 1 to n − 1 do
3:          min_index = i
4:          for j = i + 1 to n do
5:              if arr[j] < arr[min_index] then
6:                  min_index = j
7:              end if
8:          end for
9:          if arr[min_index] < arr[i] then
10:             swap(arr[i], arr[min_index])
11:         end if
12:     end for
13: end function
```

Corectness and complexity analysis

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.

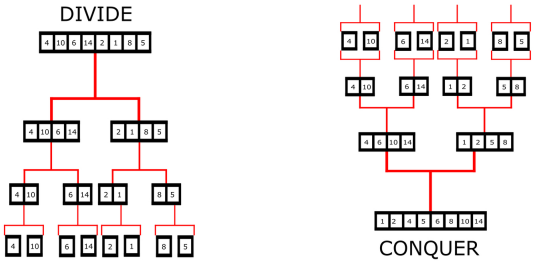

Figure 1: Merge sort in action.

Implementation

Algorithm 4 Merge Sort

```
1:  function MERGE-SORT(arr[1, ..., n])
2:      if n ≤ 1 then
3:          return arr
4:      end if
5:      mid = ⌊n/2⌋
6:      left = arr[1, ..., mid]
7:      right = arr[mid + 1, ..., n]
8:      left = MERGE-SORT(left)
9:      right = MERGE-SORT(right)
10:     MERGE(left, right, arr)
11:     return arr[1, ..., n]
12: end function
```

```
13: procedure MERGE(arr[1,...,n], left[1,...,l_size], right[1,...,r_size])
14:     l_counter = 0
15:     r_counter = 0
16:     i = 0

17:     while l_counter < l_size and r_counter < r_size do
18:         if left[l_counter] < right[r_counter] then
19:             arr[i] = left[l_counter]
20:             i = i + 1
21:             l_counter = l_counter + 1

22:         else
23:             arr[i] = right[r_counter]
24:             i = i + 1
25:             r_counter = r_counter + 1
26:         end if
27:     end while

28:     // 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.

29:     while l_counter < l_size do
30:         arr[i] = left[l_counter]
31:         i = i + 1
32:         l_counter = l_counter + 1
33:     end while

34:     while r_counter < r_size do
35:         arr[i] = right[r_counter]
36:         i = i + 1
37:         r_counter = r_counter + 1
38:     end while

39: end procedure
```

Corectness and complexity analysis

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, as, beside its efficiency in terms of time, it does not require any additionaly memory space.

Similar to Merge Sort, Quicksort uses the divide and conquer paradigm, but instead performs the division step using a **pivot** element, which splits the array in two.

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.
Corectness and complexity analysis
Implementation

Algorithm 5 QuickSort

```
1:  function QUICKSORT(arr[], start, end)
2:      if start ≥ 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] ≤ 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
```

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, other than size, will vary in type (*integer, character*), element size (*the size of each particular element, in memory*) and sorting configuration (*random, sorted, flat list*[3]).

The aim of this experiment is to showcase the strengths and vulnerabilities of each sorting algorithm. Additionally, we wish to determine what other factors, besides input array size, affect the execution time.

how much the input array, *other than its size*, affects execution time.

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`.

Test Cases

In each case, the array can have multiple sorting configurations (random order, already sorted order, flat list).

1. Sort an array of 2-byte unsigned integers, i.e. values $\in \left[0, 2^{8 \cdot 2}\right] = \left[0, 2^{16}\right]$
2. Sort an array of 4-byte unsigned integers, i.e. values $\in \left[0, 2^{8 \cdot 4}\right] = \left[0, 2^{32}\right]$
3. Sort an array of (unsigned) characters, i.e. ASCII[4] values $\in [0, 256]$.

2 In-place comparison-based sorting algorithms, compared

3 Not-in-place comparison-based sorting algorithms, compared

4 The "superior" sorting algorithm

[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)

[4] American standard bla bla bla, see table here for the representation of each character

Part IV
Conclusions and Ending

References

[1] 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.

[2] 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.

[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.