An analysis of Comb Sort

Felipe Vaiano Calderan [*]

Federal University of São Paulo (UNIFESP)

May 2022

Abstract

Comb Sort is an unstable sorting algorithm that betters Bubble Sort by first comparing elements that are far apart and by progressively reducing this gap until the compared elements are directly adjacent to one another. At this point, it behaves exactly like the Bubble Sort. The purpose of adding said gap is to eliminate small values at the end of the array (assuming an ascending sort), since they are accountable for the biggest slowdowns in a Bubble Sort. An analysis of the algorithm shows its average performance is significantly better than Bubble Sort's. This paper also exhibits many tests that display how Comb Sort stacks up against 6 other algorithms: Bubble Sort, Heap Sort, Insertion Sort, Merge Sort, Quick Sort and Selection Sort using different array sizes and starting distributions.

Contents

[*]fvcalderan@gmail.com

1 Introduction

Sorting arrays is one of the most recurrent tasks in computer science, since data is a core part of most pieces of software. This, of course, makes sorting algorithms an important topic to be researched, but they represent more than techniques that solve this specific job: they help computer scientists discuss how to discover, analyze and improve good algorithms, in general [5].

There are multiple versions of these algorithms, where some are easier to implement and have reduced source code. This is, more often than not, relevant when they are implemented in systems with very limited program memory, such as rudimentary embedded systems. One example of this kind of algorithm is the Bubble Sort, even though there are better ones in every conceivable way.

Another ordinary use of the Bubble Sort is in undergraduate courses, which is a controversial topic in itself, because the algorithm it's highly contraindicated by many scientists and educators, because they believe that early courses should adhere to established best practices [1].

The main problem with Bubble Sort is its time complexity and general purposelessness. It has an average time complexity of $O(n^2)$ and performs worse than some famous algorithms in the same class, like Insertion Sort and Selection Sort, as shown in figure 1. Moreover, Insertion Sort has the big advantage of sorting almost sorted arrays efficiently and Selection Sort can sort arrays with a linear number of movements, while Bubble Sort doesn't have any specialty attached.

Even with all the drawbacks, different ways to improve Bubble Sort were developed over the years. In 1968, Batcher [2] suggested a very intricate improvement that works by merging pairs of sorted subsequences, essentially being a merge exchange sort [5]. In 1980, Dobosiewicz [4] suggested a simpler alternative that essentially replicated what Shell Sort does to Insertion Sort default algorithm, but to Bubble Sort. Lacey [6] later rediscovered this same alternative in 1991 and called it the Comb Sort.

In this paper, I describe in details the idea behind the Comb Sort, then, I present an analysis of its complexity and show various comparisons with other sorting algorithms, some that have $O(n^2)$ average time complexity and others that have $O(nlogn)$ average time complexity. The comparisons are used to provide discussions and analyses of the CPU time usage, number of comparisons and number of movements of various algorithms.

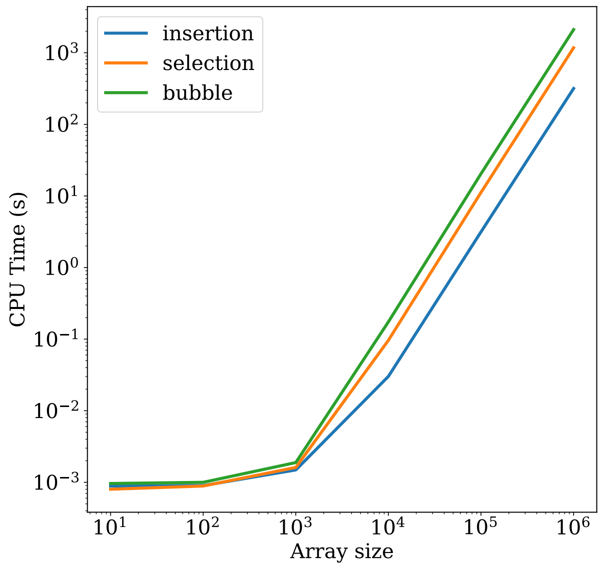

Figure 1: Average CPU time taken by $O(n^2)$ algorithms to sort Array Set 1 (see 4.2)

2 Comb Sort

Thinking about the Comb Sort as a variation of Shell Sort, its comparisons and movements are made in relation to elements not necessarily adjacent to one another, but rather, with an ever-reducing gap between them. This leads to the possibility of elements with equal value end up swapped (the one with smaller index, initially, can end up with a greater one) at the end of the sorting procedure, causing the algorithm to be unstable. The distance formed by the gap is known as the increment of the algorithm, and when the increment is 1, Comb Sort behaves precisely like Bubble Sort. A very high-level description of the Comb Sort algorithm is:

1. $n \leftarrow |array|, \; gap \leftarrow n, \; shrink > 1$
2. $gap \leftarrow 1$ if $gap = 1$, else $gap \leftarrow \lfloor gap/shrink \rfloor$
3. Compare each element at the position i with the one at the $(gap+i)$th position, $\forall \; i \in [0..n-gap]$. If the first is bigger, swap them.
4. While the array is not sorted, repeat from 2.

Having the algorithm described, let's examine its time and space complexities in the worst, average and best case scenarios.

The space complexity is trivial: Comb Sort, just like Bubble Sort, uses a fixed number of auxiliary variables to perform the movements, so it does not depend on the size or order of elements of the input array. In other words, the space complexity of the algorithm is $O(1)$, independently of the input array.

To analyze the time complexity, two different factors need to be taken into consideration: the function that quantifies the number of comparisons $C(n)$ and the number of movements $M(n)$. The overall time complexity is yielded by $T(n) = C(n) + M(n)$.

2.1 Worst Case Scenario

To obtain the worst case possible for the Comb Sort algorithm, the following circumstances are required:

- $shrink = |array|$
- Array is inversely ordered

The first item causes Comb Sort to immediately behave like Bubble Sort, because in the setup $gap = |array|$ and $shrink = |array|$, so already in the first iteration $gap \leftarrow \lfloor gap/shrink \rfloor = \lfloor n/n \rfloor = 1$.

The second item triggers the worst case for Bubble Sort (which is now also the worst case of Comb Sort). The time complexity for the worst case for the Bubble Sort is $O(n^2)$ [1]. This is both for number of comparisons and movements, therefore $T(n) = C(n) + M(n) = O(n^2) + O(n^2) = O(n^2)$.

Changing the value of *shrink* will not make the performance any worse, since $O(n^2)$ is the maximal value for time complexity $T(n)$ of the Bubble Sort algorithm. This means that if the complexity of the Comb Sort algorithm is T_s, where s is the *shrink* value, it's veracious that $T_{s'} \in O(T_n) = O(n^2),\ s' \neq n$.

2.2 Average Case Scenario

Let's consider the average case scenario for when $shrink = |array|$ and that the array is well shuffled. As already mentioned, this makes the Comb Sort act like Bubble Sort.

In this case, Comb Sort compares each element with every other one to check if any need to be raised to higher indices. This makes the number of comparisons quadratic, that is, $C(n) = O(n^2)$.

The number of movements is $M(n) = 3S(n)$, where $S(n)$ is the number of swaps. This is taking into consideration that to swap two indices A and B, an auxiliary variable C is a requisite, and 3 different movements are done: $C \leftarrow A$, $A \leftarrow B$, $B \leftarrow C$.

In a shuffled array, each element will be, in average, at a position that is $n/2$ indices away from its post-sort position. Hence, the average number of swaps is $S(n) = n \cdot n/2 = n^2/2$, so the average number of movements is $M(n) = 3n^2/2 \in O(n^2)$ and the time complexity is $T(n) = C(n) + M(n) = O(n^2)$.

If any value of *shrink* can be picked, the complexity can be trimmed to $\Omega(n^2/2^p)$, where p is the number of increments. The proof for this case is much more sophisticated and requires advanced mathematical concepts, therefore I encourage the reader to read the paper by Brejova [3] dedicated to this topic.

2.3 Best Case Scenario

The best case happens in a very similar fashion to the worst case, except the array should be ordered (not inversely ordered). Having Comb Sort behave just like Bubble Sort, if the array is already ordered, the best case scenario for the Bubble Sort algorithm happens. In this situation, the time complexity of Bubble Sort (and Comb Sort) is $O(n)$. More precisely, the algorithm makes n comparisons (a single scan through the array) and 0 movements (the elements already are in their ideal positions), therefore the complexity is $T(n) = C(n) + M(n) = O(n) + O(1) = O(n)$.

If any value for *shrink* can be picked, the best case of the Comb Sort can be generalized as $O(nlogn)$, that is, it will not get any better than when $shrink = |array|$, since this already leads to the best case possible. With this, it is possible to conclude that if the complexity of the best case for the Comb Sort algorithm is T_s, then $T_{s'} \in \Omega(T_n),\ s' \neq n$. The proof for the more generalized can also be found in [3].

In Subsection 5.2, I further cover the importance of the *shrink* value in the time complexity of the Comb Sort algorithm.

3 Implementation

Although the implementation of the Comb Sort itself is very small, the complete code for this project (including the functions to execute the tests) is too extensive to enter in details here. In this paper, I'll focus on the general structure of the project and the Comb Sort.

The whole implementation, except for the code to generate charts, is written in the C language, since the compiled program runs very fast and is simple enough to not required specialized libraries to be easily implemented.

3.1 Main Function

Starting with the data structures, the program uses primitive variable types (such as `void`, `int` and `float`), integer arrays to store the values before and after the sorting procedure and also an abstract data type (ADT) called `sort`.

The `sort` ADT is used to store the amount of comparisons and movements made by the sorting algorithms and is implemented as follows:

```c
typedef struct {
    uint64_t cmp; /* # of comparisons */
    uint64_t mov; /* # of movements   */
} sort;
```

All the sorting algorithms are of type `sort`, so the main function can collect the data to print to the screen. The pipeline of the `main` function is the following:

1. Load array to be sorted from saved binary file into a C array
2. Sort the C array using any of the 7 implemented sorting methods
3. Print number of comparisons and movements to the screen

Notice that the `main` function does not print any time, since the CPU time is externally calculated using Unix `time` program. This is important, considering that `time` can separate between total, user, and system time. For this paper, only the user time is relevant

3.2 Sorting Algorithms

Although Comb Sort is the centerpiece of this paper, in Subsection 5.1 it's compared to 6 other algorithms, so all 7 algorithms were implemented using the same coding style. The other algorithms are: Insertion Sort, Selection Sort, Bubble Sort, Heap Sort, Merge Sort and Quick Sort. The first half of these are $O(n^2)$ algorithms and the last half are $O(nlogn)$ algorithms.

Let's take a look at the Comb Sort implementation. The other algorithms follow a very similar pattern and can be found inside the project's repository provided.

```c
 1 sort comb_sort(int * A, int n)
 2 {
 3   int sorted = 0, gap = n, i, sm, aux;
 4   sort s = { .cmp = 0, .mov = 0};
 5 
 6   while (!sorted) {
 7     gap = (int)floor(gap / 1.3);
 8     if (gap <= 1) {
 9       gap = 1;
10       sorted = 1;
11     }
12 
13     for (i = 0; i < n - gap; i++) {
14       sm = gap + i;
15       if (/* cmp */ s.cmp ++, A[i] > A[sm]){
16         aux = A[sm];   /* mov */
17         A[sm] = A[i];  /* mov */
18         A[i] = aux;    /* mov */ s.mov += 3;
19         sorted = 0;
20       }
21     }
22   }
23 
24   return s;
25 }
```

The interesting bits of this code are lines 7 to 11, which compute the gap size for the current iteration and lines 15 to 20, that swaps the elements being compared if the first is greater than the second (this is, of course, where the sorting happens).

It's possible to see the application of the `sort` ADT in line 15: there's a comparison between keys, so the program increases `s.cmp` by one. The use of `sort` is also visible in line 18, where 3 new key movements are summed to `s.mov`.

3.3 Array Generation Tool

In Subsection 3.1 I mentioned the program loads a binary file into a C array. This file needs to be generated in the first place, so the Array Generation Tool can be used.

There is not much to say about this tool, apart from the fact that it can generate arrays of any size (up to the signed integer's maximum size), with a specific seed and specifics mode and type. Subsection 4.2 briefly describes all the available modes and types.

After the program has generated the entire array inside RAM, it saves the contents to a binary file that can be loaded by the main program. There is also a simple Array Loader Tool that loads and prints the array to the screen for debugging purposes.

3.4 Comb Sort Shrink Tool

Some of the tests that I ran were to check how different values for the *shrink* variable affects the performance of the algorithm, so I created a simplified version of the program exclusively for this purpose. It runs the Comb Sort algorithm and prints the number of comparisons, movements and the current *shrink* value to the screen.

4 Computational Environment

4.1 Computer Specifications

The environment used to compile the program and run the tests has the following specifications:

Hardware:

- **CPU:** 1 Core of AMD EPYC 7551
- **RAM:** 1GiB DIMM RAM

Software:

- **OS:** Ubuntu 20.04.4 LTS
- **KERNEL:** Linux 5.13.0-1018-oracle
- **GCC:** 9.4.0

4.2 Array Set 1

The arrays generated have the following characteristics:

- 10 seeds for the random number generator
- 4 modes: ordered, inversely ordered, almost ordered and random
- 2 set types: only unique elements and unrestricted
- 6 sizes: 10, 100, 1000, 10000, 100000 and 1000000 elements

where almost ordered arrays have 1% of their elements (when possible) at the wrong places.

This set contains the combination of all the above characteristics, adding up to $10 \cdot 4 \cdot 2 \cdot 6 = 480$ different arrays.

4.3 Array Set 2

In this set, all the 100000 arrays have 10000 randomly generated elements, with repetition allowed.

5 Tests

5.1 CPU Time, Number of Comparisons, and Number of Movements

Here, I compare the Comb Sort algorithm to six other sorting algorithms: Insertion Sort, Selection Sort, Bubble Sort, Heap Sort, Merge Sort and Quick Sort. For these tests, the Array Set 1 (See subsection 4.2) was utilized.

For all the tests in this section, a *shrink* value of 1.3 is being used for the Comb Sort algorithm. In later analyses, I will show how this value affects the performance of the algorithm. The value 1.3 is suggested by Lacey [6] as the best value possible, according to their tests.

It is also worth mentioning that the Merge Sort algorithm is not in-place, as it uses auxiliary arrays to sort the data. This lets the algorithm stay in the $O(nlogn)$ time complexity category, but increases its space complexity to $O(n)$, while every other algorithm implemented has a space complexity of $O(1)$.

This paper exhibits the experimental results comparing the algorithms in 3 different collections:

- Almost ordered arrays
- Inversely ordered arrays
- Randomly ordered arrays

where the arrays are composed of unique elements. After these 3 collections are presented, I also show the average results including all the arrays from the Array Set 1.

There is not a substantial difference between the performance of the algorithms when ordering a fully ordered or an almost ordered array, and the same applies to the variants with repeating elements. That is why these variants are not worth analyzing here.

5.1.1 Almost ordered

Almost ordered arrays are the best case for many of the implemented algorithms, losing only to fully ordered arrays. There are exceptions, like the Quick Sort implementation, which does poorly in both of these arrays variations.

Figure 2 shows that the average CPU time taken by the Comb Sort algorithm to sort the arrays is comparable to Heap Sort. This same pattern emerges on the number of Comparisons, depicted in Figure S1. When looking at the number of movements made by Comb Sort and Heap Sort, the first algorithm performs a considerably lower number of movements when sorting smaller arrays, but when the size reaches 10^4 elements, there is not much difference between them.

Merge Sort has some trouble sorting the smaller arrays efficiently, but it's probably due to population bias, since only 10 different seeds were tested for this specific array setup.

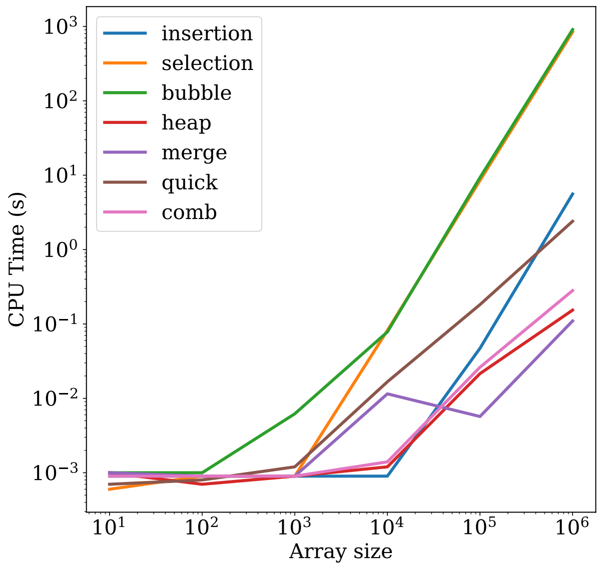

Figure 2: Average CPU time taken by each one of the implemented algorithms to sort almost ordered arrays of the Array Set 1

Comb Sort beats all the $O(n^2)$ algorithm in this category, but due to the increased code complexity (including floating point division), it may not be advisable for rudimentary embedded systems when sorting smaller arrays, since Insertion Sort fares very well up to 10^4 arrays.

If the number of movements is still the most important criterion, i.e., sorting arrays where each element is very big in terms of memory consumption, Selection Sort is still the best among the tested ones by a very big margin, as shown in Figure S2.

5.1.2 Inversely Ordered

Inversely ordered arrays is the worst case scenario for many of the implemented algorithms, including the Comb Sort. Figure 3 reveals that Comb Sort and Heap Sort, once again, have very close performance in

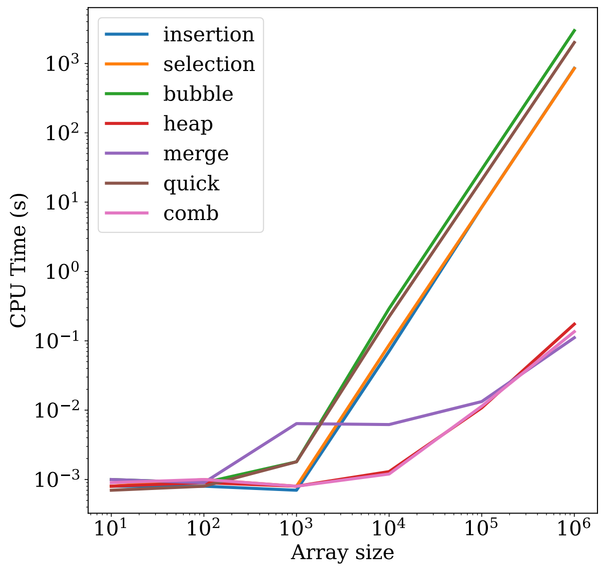

Figure 3: Average CPU time taken by each one of the implemented algorithms to sort inversely ordered arrays of the Array Set 1

terms of CPU time. Merge Sort has some difficulties again, for the same reason as before.

In terms of number of comparisons, Comb Sort does more of them than Heap Sort, but in terms of movements, it's the complete opposite, as observed in Figures S3 and S4. Needless to say, Selection Sort still makes fewer movements than any of the implemented algorithms.

Quick Sort has a very hard time dealing with inversely ordered arrays, so Comb Sort would be a very good alternative to it in cases where this kind of situation can show up, but so would be any of the other efficient algorithms, like Heap Sort and Merge Sort.

5.1.3 Randomly Ordered

Except for systems where the array progressively (and slowly) grows with new elements, while being ordered at each step, randomly ordered arrays are the most common situation to come along, so these results are arguably the most relevant up to now.

Observing Figure 4, it's possible to see a clear detachment between the $O(n^2)$ and $O(nlogn)$ average case algorithms. This is especially interesting considering that the average case for the Comb Sort algorithm is $O(n^2)$ (or $\Omega(n^2/2^p)$, regarding a more careful analysis) and yet, it acts much more like an $O(nlogn)$ algorithm.

Although it seems to do worse sorting smaller arrays, Comb Sort had the same problem as the Merge Sort had in past tests. Also, this category of arrays is favorable for the Quick Sort algorithm, contrary to the other two categories shown.

In terms of number of comparisons, all the $O(nlogn)$ algorithms (and Comb Sort) stay very close to one another (Figure S5), and the same can be said for the number of movements (Figure S2). Again, the Selection Sort algorithm is the one that makes the fewest amount of movements.

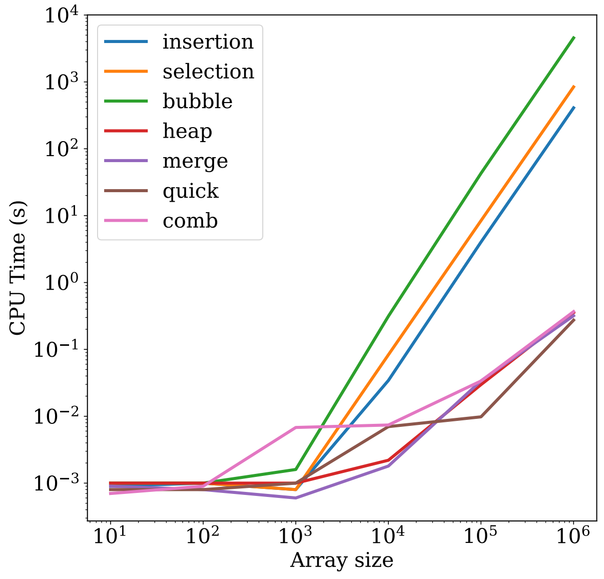

Figure 4: Average CPU time taken by each one of the implemented algorithms to sort randomly ordered arrays of the Array Set 1

5.1.4 All arrays

When considering all the arrays available in the set, the resulting data is very similar, except for the Quick Sort algorithm, which ends up having a much worse overall performance due to the addition of ordered and inversely ordered arrays. In CPU time, Comb Sort and Heap Sort are again very close (and so is Merge Sort), as it's possible to see in Figure 5.

The verbatim same can be said about the number of comparisons (Figure S7). Comb Sort is slightly more successful in respect to the number of movements made than the $O(nlogn)$ algorithm, losing only to Selection Sort.

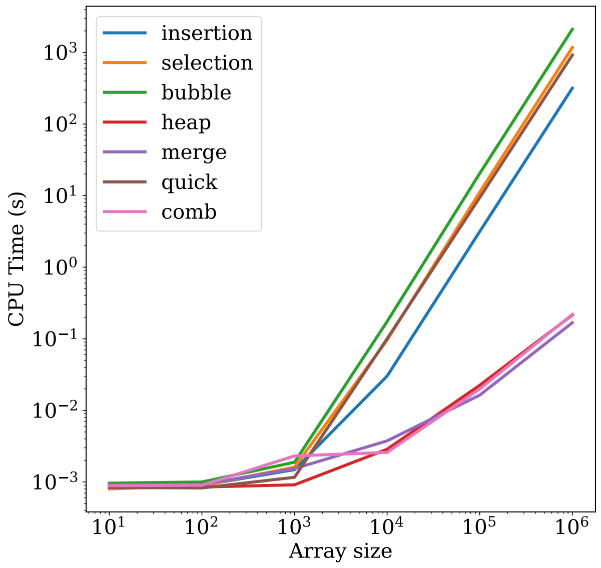

Figure 5: Average CPU time taken by each one of the implemented algorithms to sort the Array Set 1

5.2 Shrink constant effect

Throughout all previous tests, the value for the *shrink* constant was kept as 1.3. As already mentioned, this value is the one proposed by Lacey [6]. They tested different *shrink* values for sorting 200,000 random arrays with 1000 to 1040 elements each and empirically found 1.3 as the optimal value, since it reduced the amount of comparisons made by the Comb Sort algorithm.

In their paper, they also stated that the number of comparisons is just about equivalent to the sorting time, so that is why the number of comparisons was used to select the best value for the *shrink* constant.

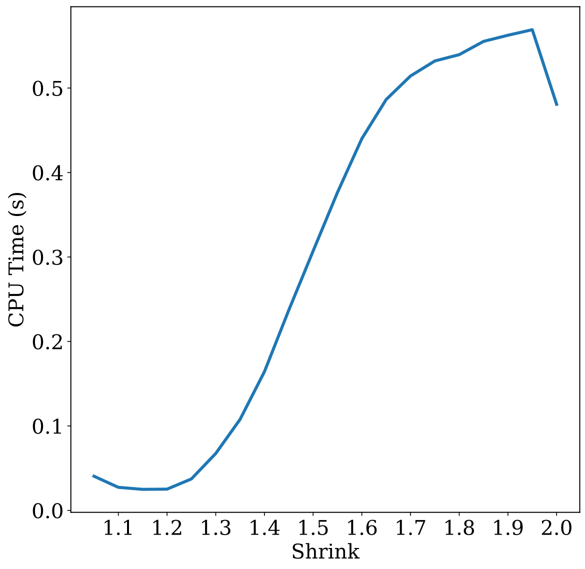

Figure 6: Average CPU time taken by Comb Sort, using different values for *shrink*, to sort the Array Set 2

I tested assorted values (from 1.05 to 2.00 in steps of 0.05) for *shrink* running the Comb Sort algorithm for 10,000 random arrays with 10,000 elements each and detected that, in fact, 1.3 is the best value in terms of number of comparisons, but it does not directly correlate to the CPU time taken. Using the CPU time as the measure, instead of number of comparisons, the chosen value for *shrink* would be 1.2 (or 1.25 if the number of movements was the criterion).

Table 1 reports these measurements up to $shrink = 1.5$, since after that, the value explodes.

Figure 6 shows the average CPU time plot, Figure S9 illustrates the average number of comparisons, and Figure S10, the average number of movements, all of them up to $shrink = 2.0$.

Shrink	CPU time	# comps	# moves
1.05	0.040487	2849504	465103
1.10	0.027313	1615526	261087
1.15	0.024935	1244211	240178
1.20	0.025182	1031265	229554
1.25	0.037266	977341	228487
1.30	0.067407	950057	239405
1.35	0.107795	1099693	261955
1.40	0.164257	3407378	389446
1.45	0.237102	9742929	763954
1.50	0.307125	16378272	1220444

Table 1: Average CPU time taken, average number of comparisons and average number of movements made by Comb Sort, using different values for *shrink*, to sort the Array Set 2

6 Conclusion

Comb Sort takes the foundation of the Bubble Sort algorithm and combines with the improvements Shell Sort makes to Insertion Sort, bringing forth a more efficient algorithm that competes closely to the more known $O(nlogn)$ sorting algorithms.

Although the algorithm is rather simple to implement, it is challenging to analyze deeply, due to the nature of how the elements are compared. The *gap* value makes it problematic to find a consistent relation between the iterations and the closeness of sorting completion, since it is reduced by a constant known as *shrink*, which is, typically, a value greater than 1 and smaller than 2.

Using the data collected from the experimental analyses, the Comb Sort algorithm performs very closely to the Heap Sort. This makes Comb Sort a questionable choice, since Heap Sort has $O(nlogn)$ complexity for the best, average and worst case scenarios, while Comb can perform as poorly as the Bubble Sort algorithm ($O(n^2)$) at times.

Comb Sort also performs similarly to the Merge Sort and Quick Sort algorithms. It is a better option than Merge Sort when space complexity is a concern, if stability is dispensable (since Merge Sort is stable, Comb is not). It is a better pick than Quick Sort in the worst case scenarios, but not in the average ones.

The value for the *shrink* constant is complicated to analyze. The literature indicates that 1.3 might be a good choice, since it is the one that leads to the fewest amount of comparisons. Even though this is corroborated in this paper, the problem arises when the CPU time is taken into account. There is not a perfect match between them, since in my experimental results, the lowest average CPU time taken to sort is acquired by choosing $shrink = 1.2$.

In general, the Comb Sort algorithm does the job of sorting relatively fine, but it does not supply anything new to the table that more well-researched algorithms (with more specialized variants), like the Heap Sort, do. This makes its use-case limited to theoretical studies, as there are better alternatives.

References

[1] Owen Astrachan. Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, 35(1):1–5, 2003.

[2] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[3] Bronislava Brejová. Analyzing variants of shellsort. *Information Processing Letters*, 79(5):223–227, 2001.

[4] Wlodzimierz Dobosiewicz. An efficient variation of bubble sort. *Information Processing Letters*, 11(1):5–6, 1980.

[5] Donald E Knuth. Sorting and searching. 1973.

[6] Stephen Lacey and Richard Box. A fast, easy sort. *Byte*, 16(4):315–ff, 1991.

Supplementary Information

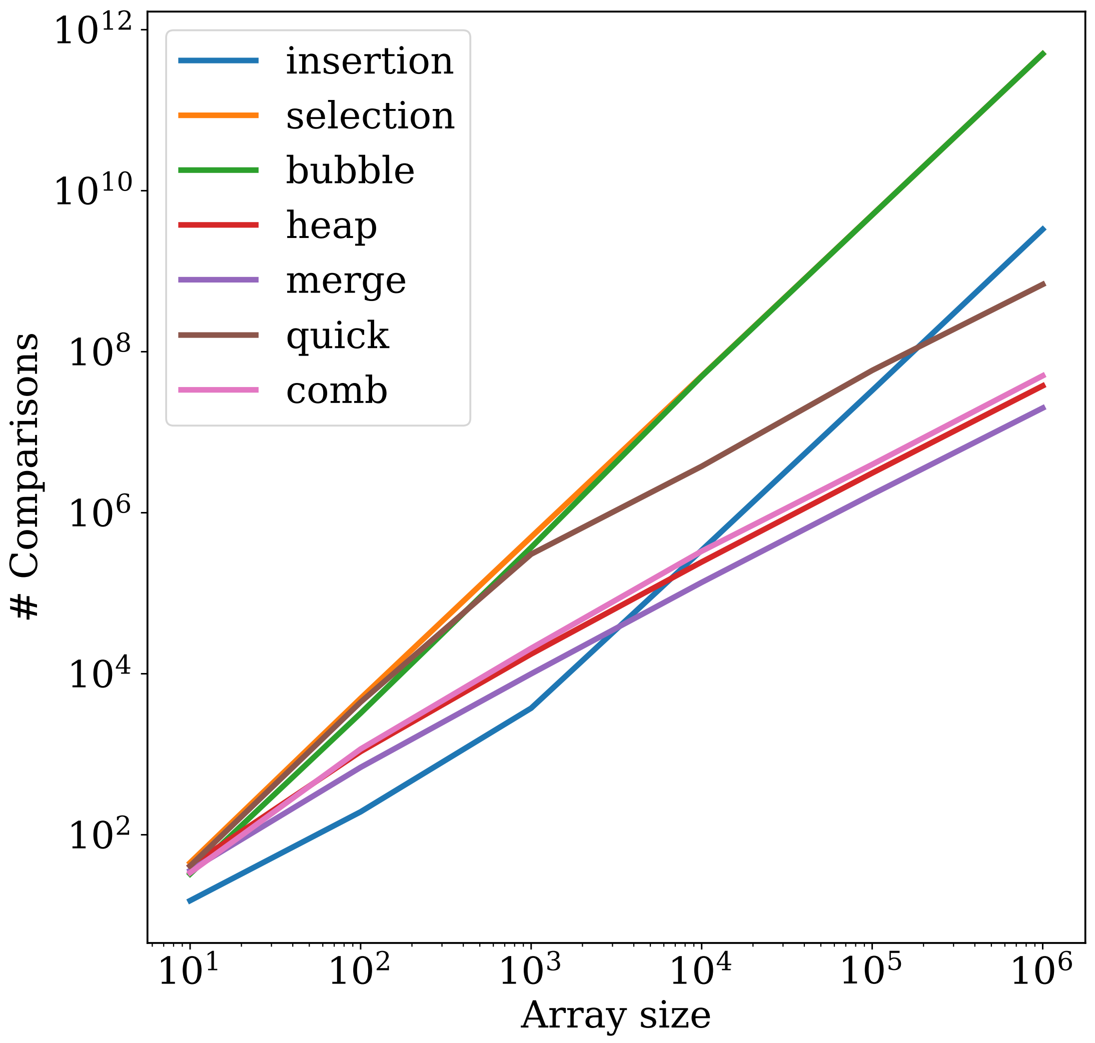

Figure S1: Average number of comparisons made by each one of the implemented algorithms to sort almost ordered arrays of the Array Set 1

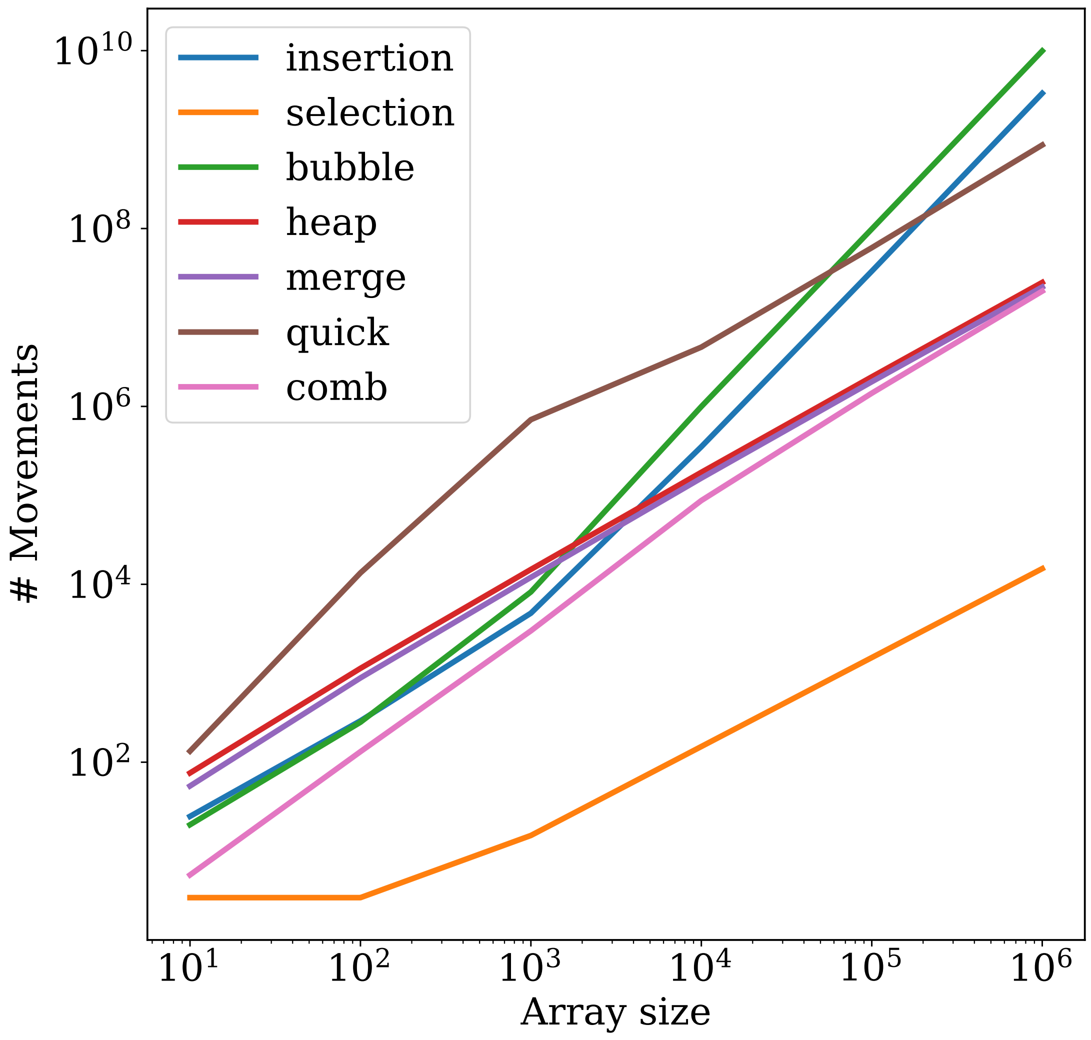

Figure S2: Average number of movements made by each one of the implemented algorithms to sort almost ordered arrays of the Array Set 1

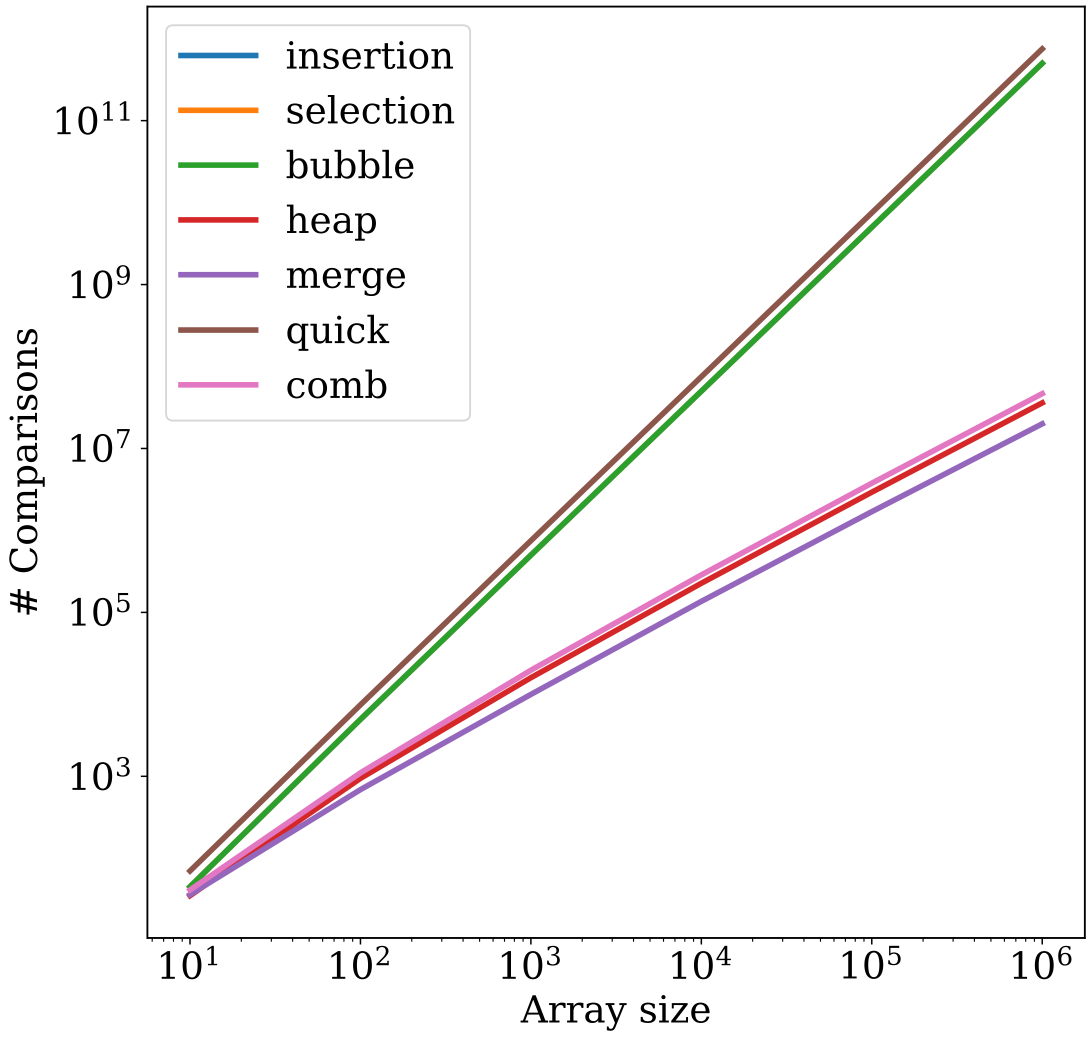

Figure S3: Average number of comparisons made by each one of the implemented algorithms to sort inversely ordered arrays of the Array Set 1

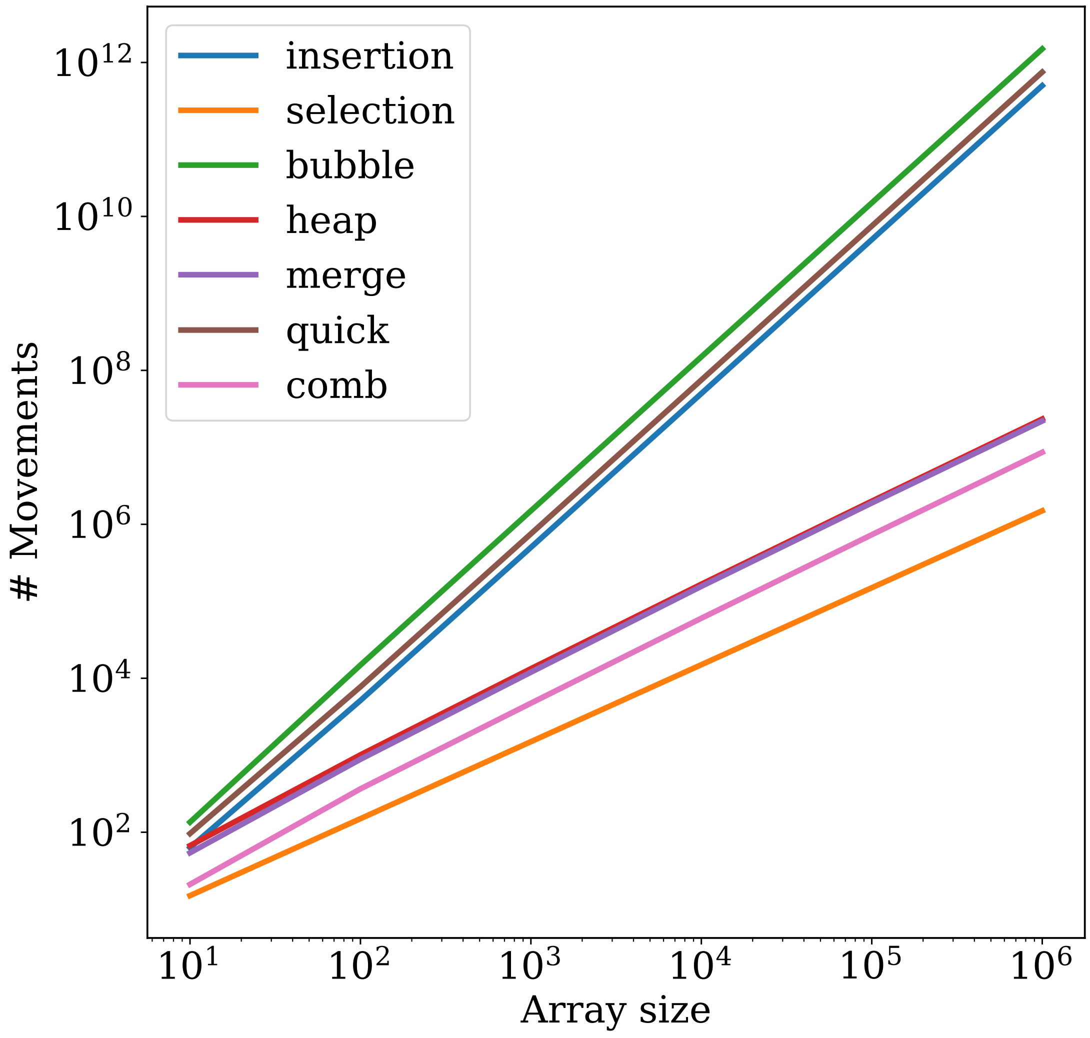

Figure S4: Average number of movements made by each one of the implemented algorithms to sort inversely ordered arrays of the Array Set 1

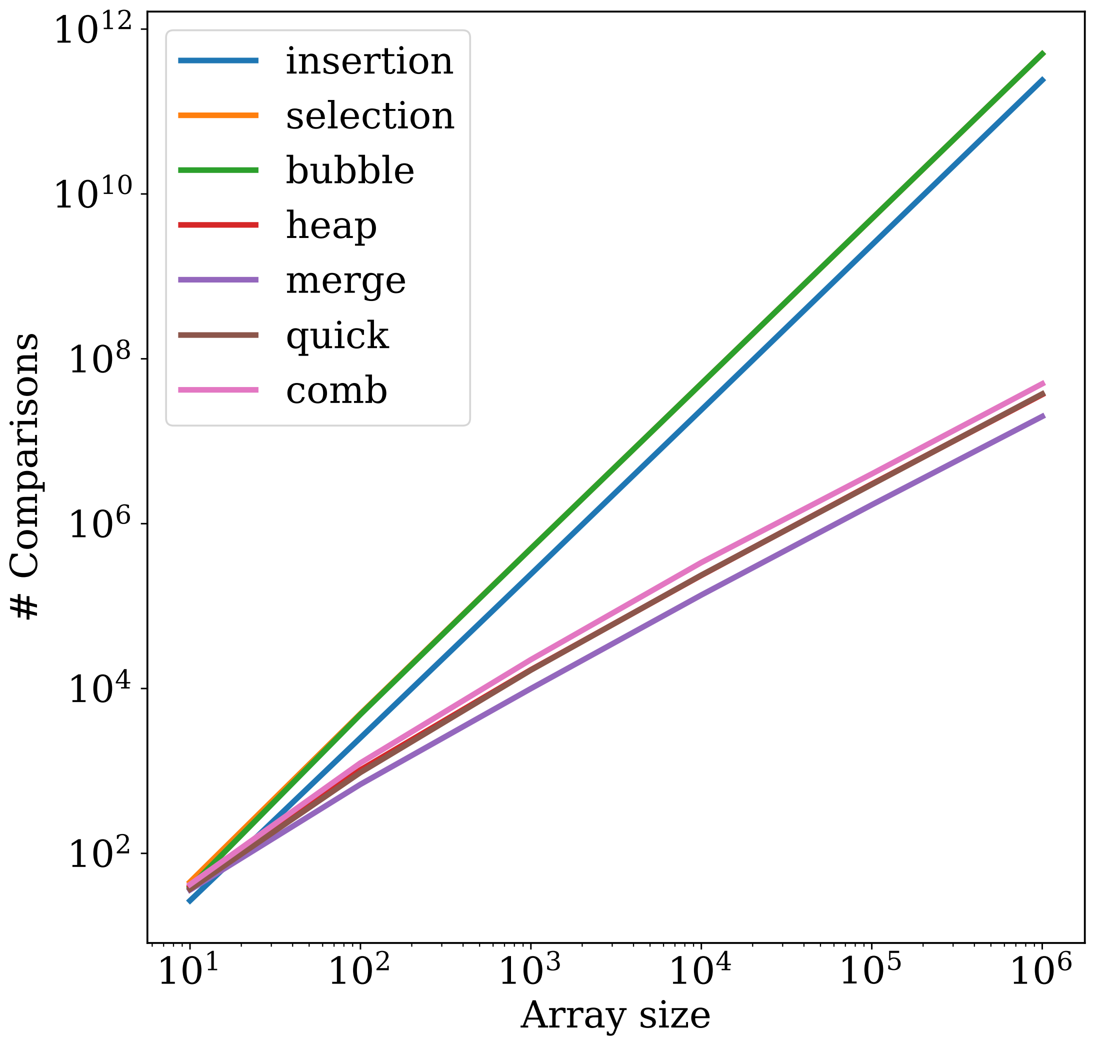

Figure S5: Average number of comparisons made by each one of the implemented algorithms to sort randomly ordered arrays of the Array Set 1

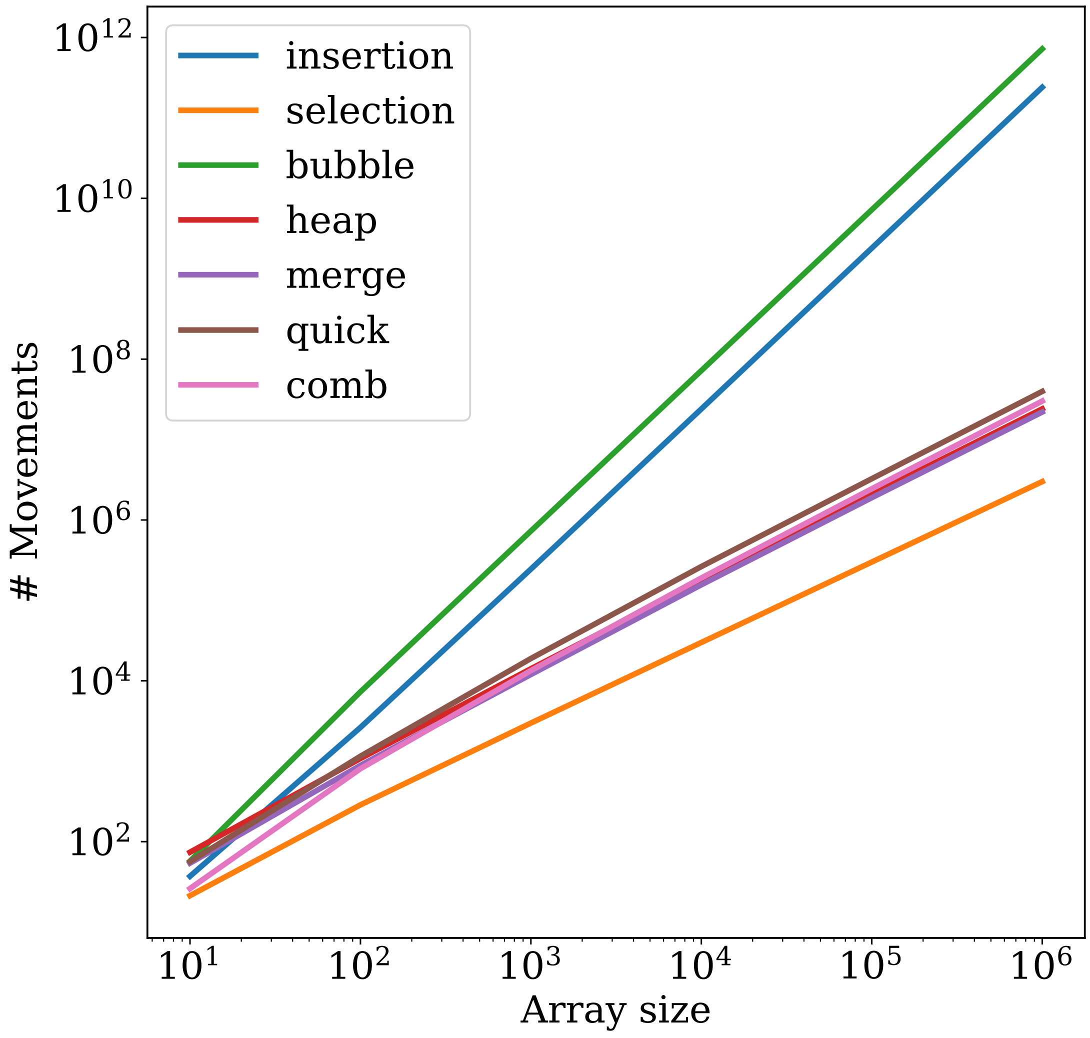

Figure S6: Average number of movements made by each one of the implemented algorithms to sort randomly ordered arrays of the Array Set 1

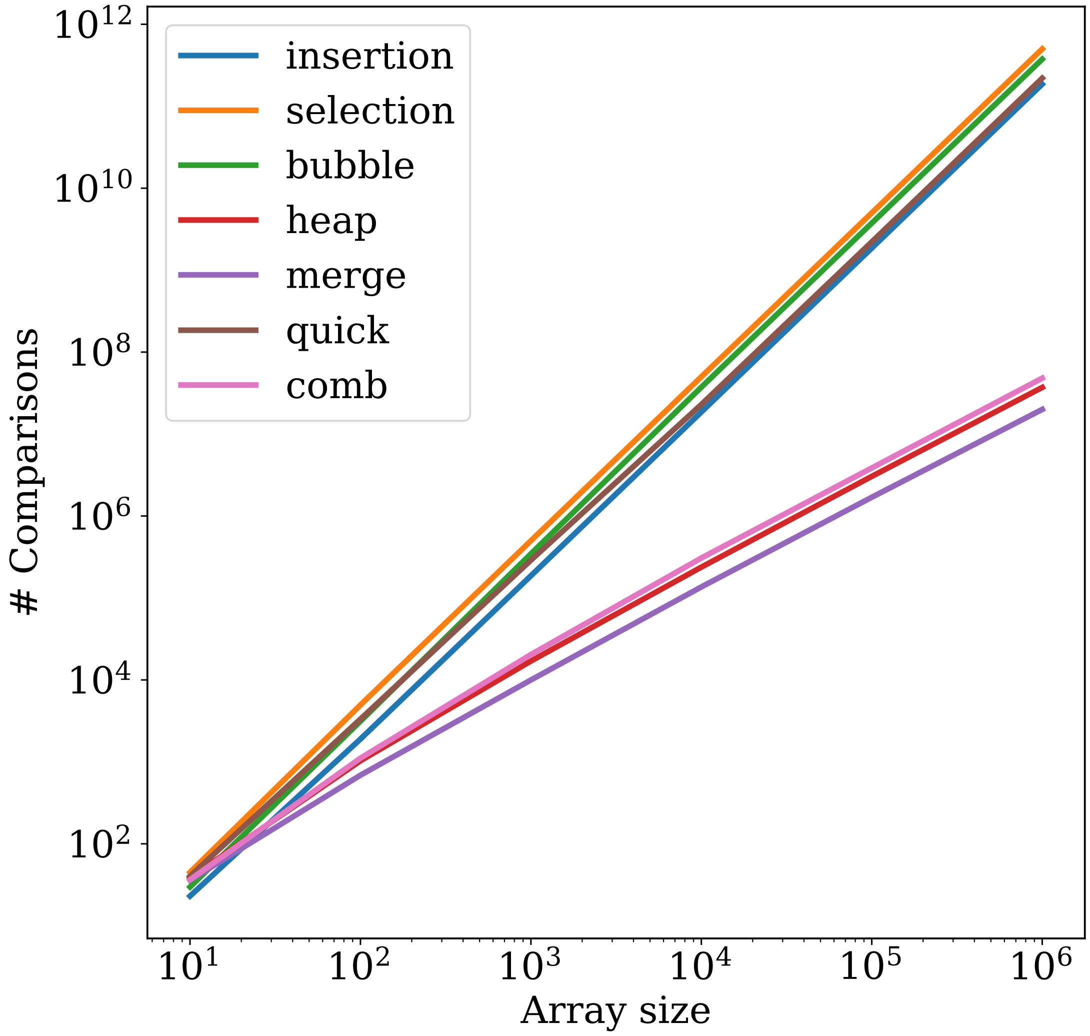

Figure S7: Average number of comparisons made by each one of the implemented algorithms to sort the Array Set 1

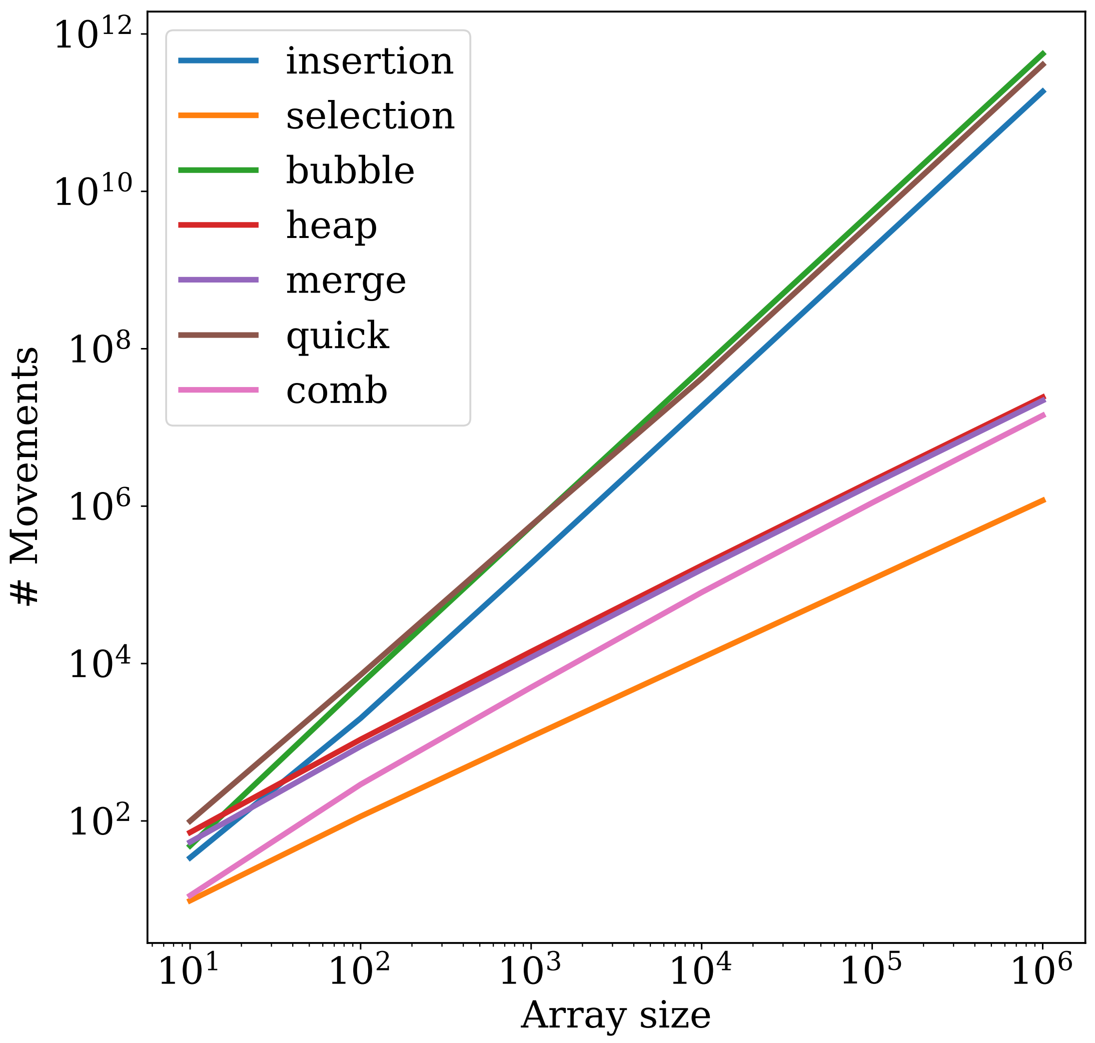

Figure S8: Average number of movements made by each one of the implemented algorithms to sort the Array Set 1

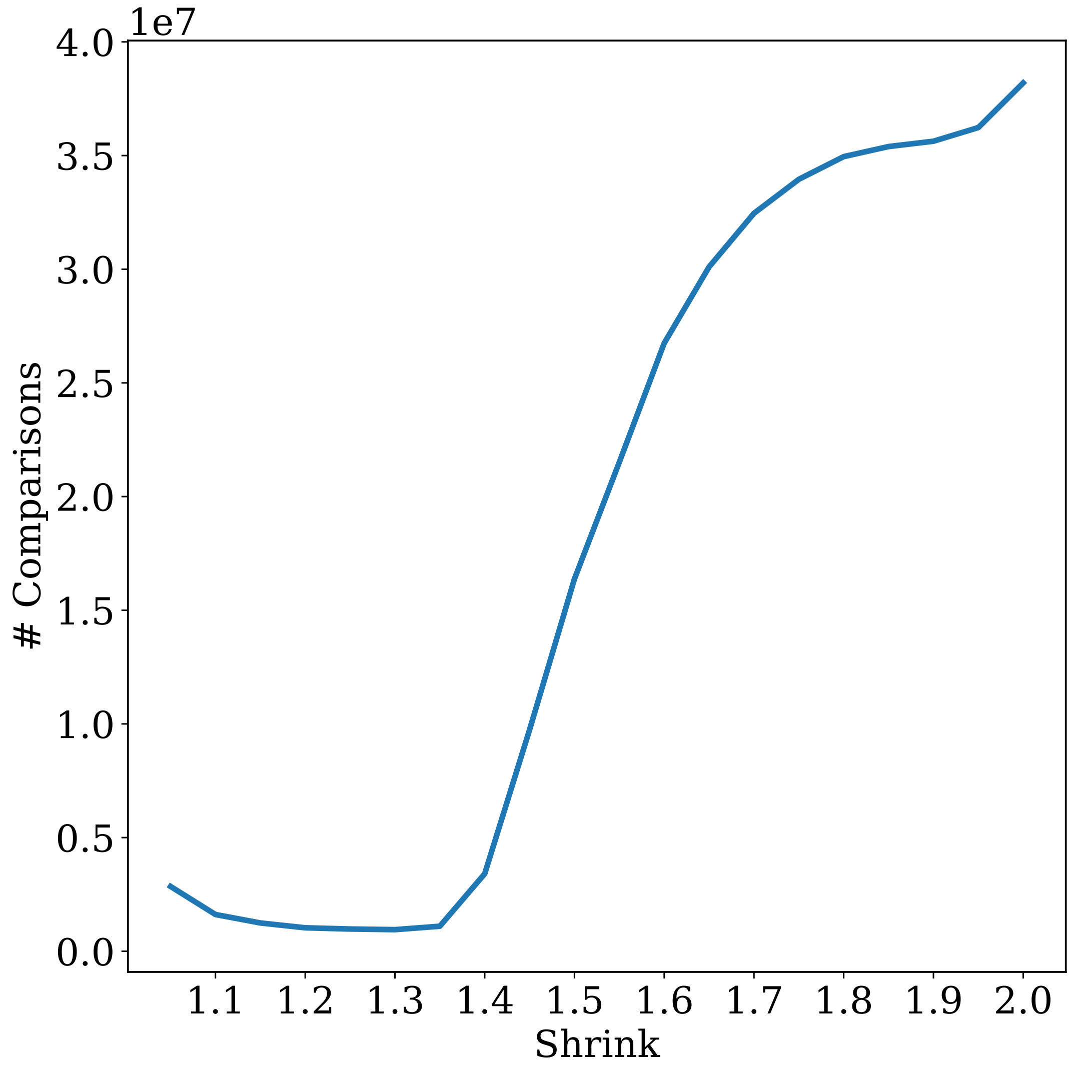

Figure S9: Average number of comparisons made by Comb Sort, using different values for *shrink*, to sort the Array Set 2

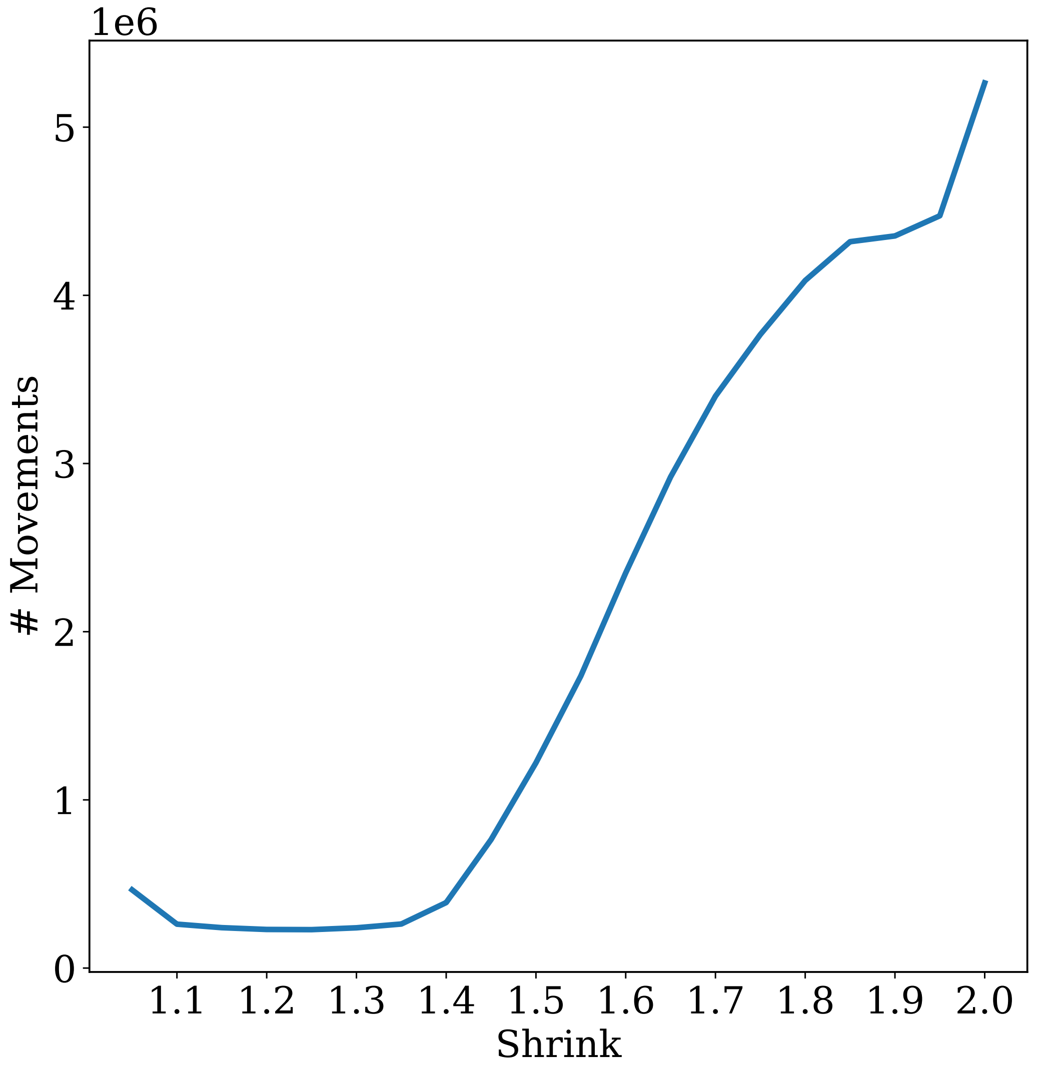

Figure S10: Average number of movements made by Comb Sort, using different values for *shrink*, to sort the Array Set 2