An experimental and visual comparison of the most well-known sorting algorithms

Alexander Hauer
Department of Computer Science
West University of Timişoara, România,
Email: alexander.hauer05@e-uvt.ro

May 2024

Abstract

This paper aims to find notable differences between some of the most taught sorting algorithms, to determine their strengths and weaknesses and to draw reasonable conclusions.

This is mostly an experimental comparison ran on python, and its purpose is to give an adequate answer to the question "what sorting algorithm should I use?".

Contents

List of Tables

List of Figures

1 Introduction

In educational settings, studies are more leaned towards a theoretical comparison. Sorting algorithms are seldom experimentally compared to one another, making it hard to truly understand their differences in performance.

For example, a computer science student probably knows that *Bubble Sort* is not recommended due to its lesser performance, but they don't know what that *truly* means in terms of actual time he would potentially waste.

Using Python, we implemented different sorting algorithms, as well as different methods of creating lists, and tested them in such a way that it becomes clear where each sorting technique shines.

My contribution is that we implemented different ways of creating lists, ran the experiments and analysed the data to make it **easier** to read and understand. We did not come up with any of the sorting algorithms.

This paper begins by explaining the specific tests we have done, then it proceeds on the inner workings of them, as well as instructions on how do them yourself. Next, will follow the results presented in graphs, and finally some conclusions. Feel free to skip to the part you are more interested in.

2 Formal Description of Problem and Solution

The problem in question is understanding *the most optimal way* different sorting algorithms should be used. In order to do this, we came up with several ways of creating lists, in order to test the sorting algorithms in varying environments.

I used the following sorting techniques:

1. **Bubble Sort:**
   Bubble Sort[1] is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

2. **Cocktail Shaker Sort:**
   Cocktail Shaker Sort[2], also known as Bidirectional Bubble Sort, is a variation of Bubble Sort that sorts in both directions, from the beginning to the end and then from the end to the beginning, in each pass.

3. **Insertion Sort:**
   Insertion Sort[3] is a simple sorting algorithm that builds the final sorted array one item at a time. It repeatedly takes the next element from the unsorted part and inserts it into its correct position in the sorted part.

4. **Selection Sort:**
   Selection Sort[4] is a simple sorting algorithm that divides the input list into two parts: the sorted part and the unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part.

5. **Counting Sort:**
   Counting Sort[5] is an integer sorting algorithm that works by determining the number of occurrences of each unique element in the input list and then sorting the elements based on these counts.

6. **Radix Sort:**
   Radix Sort[6] is a non-comparison sorting algorithm that sorts integers by first grouping the individual digits of the same place value together and then sorting the numbers based on each digit, from the least significant digit to the most significant digit.

7. **Merge Sort:**
   Merge Sort[7] is a comparison-based sorting algorithm that divides the input list into two halves, sorts each half recursively, and then merges the sorted halves to produce the final sorted list.

8. **Heapsort:**
   Heapsort[8] is a comparison-based sorting algorithm that builds a max-heap from the input list and repeatedly extracts the maximum element from the heap and rebuilds the heap until the list is sorted.

9. **Quicksort:**
   Quicksort[9] is a comparison-based sorting algorithm that partitions the input list around a pivot element, sorts the sublists recursively, and then combines the sorted sublists to produce the final sorted list.

10. **Strand Sort:**
    Strand Sort[10] is a sorting algorithm that repeatedly pulls sorted sublists from the input list and merges them together until the entire list is sorted.

Name	**Worst Case**	**Stability**
Bubble Sort	$O(n^2)$	Stable
Cocktail S. Sort	$O(n^2)$	Stable
Insertion Sort	$O(n^2)$	Stable
Selection Sort	$O(n^2)$	Unstable
Counting Sort	$O(n + 2^k)$	Stable
Radix Sort	$O(n \cdot \frac{k}{s})$	Stable
Merge Sort	$O(n \cdot \log n)$	Stable
Heapsort	$O(n \cdot \log n)$	Unstable
Quicksort	$O(n^2)$	Stable
Strand Sort	$O(n^2)$	Stable

Table 1: Sorting Algorithms Properties

Note: The variable k represents the size of each key, and s represents the chunk size.

These tests were ran on n numbers, using the following methods:

- Each number from 1 to n appearing only once, in a shuffled manner
- An almost ordered list, with 0.5 percent of its elements unordered
- A high-density list, with values ranging only from 1 to 0.5 percent of n
- A list mostly populated by 0, where only 0.5 percent of elements are different than 0
- A list ordered in reverse
- A list made of presorted chunks, each chunk being a tenth of the original list
- n lists of 10 numbers

All the sorting algorithms were used up to 100 thousand elements, after that only some of them were tested even more due to their greater efficiency, and for the sake of not wasting time.

3 Model and Implementation of Problem and Solution

In order to do these experiments yourself, one must install the most recent version of python from here. Then, it is strongly advised that you install an IDE, such as PyCharm, for writing the code in a suitable environment.

The link to all my code can be found here.

The different methods of creating lists, just like the sorting algorithms, are all contained in their respective functions making it easier to debug and read.

The list creating functions take as argument the number of elements to be added, and the sorting functions take as argument the list to be sorted.

Starting the program, the starting number and end number are requested to be given.

It starts a loop, going from the starting number to the end number, incrementing tenfold. Each time, a list with the respective amount of elements in created, as well as a copy of it.

The program goes through the algorithms, each one sorting the same list, such that the data is more accurate.

To input the data, simply type the requested information, starting with the starting number, the the end number.

Output looks like this:

```
For n = 10000, the following results are obtained:
10k

================================================

Bubble Sort: Execution time: 4.976732015609741 seconds
Cocktail Shaker Sort: Execution time: 4.142573833465576 seconds
Insertion Sort: Execution time: 2.1026768684387207 seconds
Selection Sort: Execution time: 2.282494306564331 seconds
Counting Sort: Execution time: 0.0048215389251708984 seconds
Radix Sort: Execution time: 0.013196706771850586 seconds
Merge Sort: Execution time: 0.0227358341217041 seconds
Quick Sort: Execution time: 0.01834726333618164 seconds
Heap Sort: Execution time: 0.030495405197143555 seconds
Strand Sort: Execution time: 0.07539176940917969 seconds
[4.976732015609741, 4.142573833465576, 2.1026768684387207, 2.282494306564331,
Done

Process finished with exit code 0
```

Figure 1: Output Image

4 Case Study

The presented logarithmic-logarithmic plots (log-log plots) illustrate the performance of various sorting algorithms concerning the size of input lists, focusing on randomly generated data. Each algorithm's execution time, measured in seconds (s), is depicted against the number of elements in the input list.

4.1 Random list

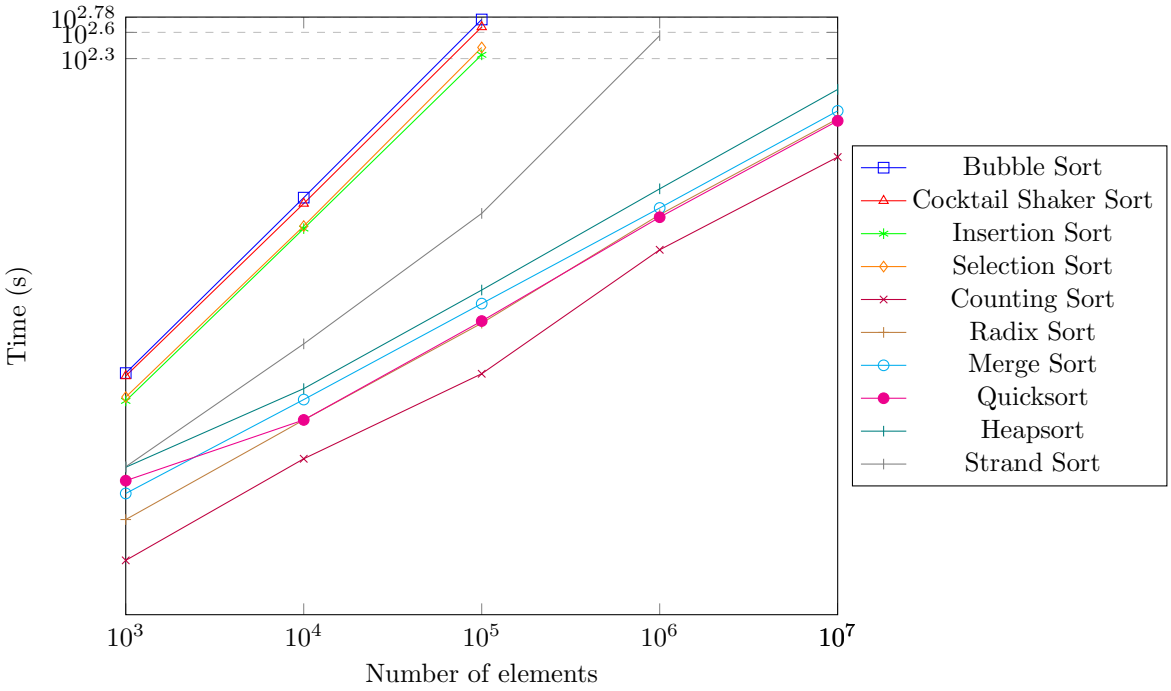

Figure 2: Random List Sorting Algorithm Performance

Observing the graph, several key insights can be discerned:

- Bubble sort and Cocktail Shaker Sort show poor scalability for large datasets.
- Insertion Sort and Selection Sort, although twice as fast as the previous two, are still not performing well as the number of elements increases.
- Strand Sort stands out for its initially efficient performance, but its execution time eventually escalates significantly for larger input sizes
- Quicksort, Radix Sort, Merge Sort and Heapsort, exhibit efficient performance across all input sizes.
- Counting Sort takes the first place, being almost 3 times as fast as second place(Quicksort), showcasing its remarkably efficient performance.

4.2 Almost ordered list

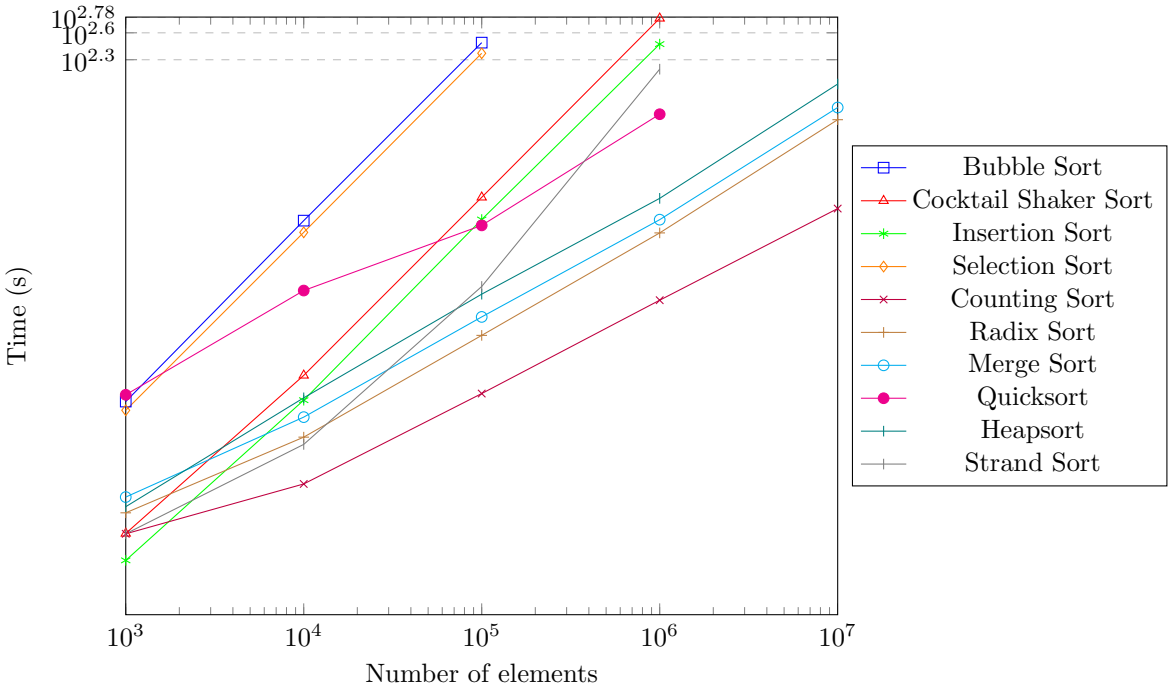

Figure 3: Almost Ordered List Sorting Algorithm Performance

Key observations from the graph are as follows:

- Most of the sorting algorithms exhibit increased performance on ordered lists, compared to the previous test.
- Bubble Sort and Selection Sort perform only slightly better than the previous test.
- Strand Sort, Cocktail Shaker Sort and Insertion Sort show much better performance than previously. However, they remain inferior to other more efficient sorting techniques.
- Counting Sort once again stands out due to its remarkable performance, taking the top place once again, performing nearly 10 times as fast as second place, Radix Sort.
- Radix Sort, Merge Sort and Heapsort perform relatively similar, albeit ever so slightly worse than the previous test.
- Quicksort performs worse, increasing from previous test:3 seconds, to 50 seconds(for 1 million elements).

4.3 High density list

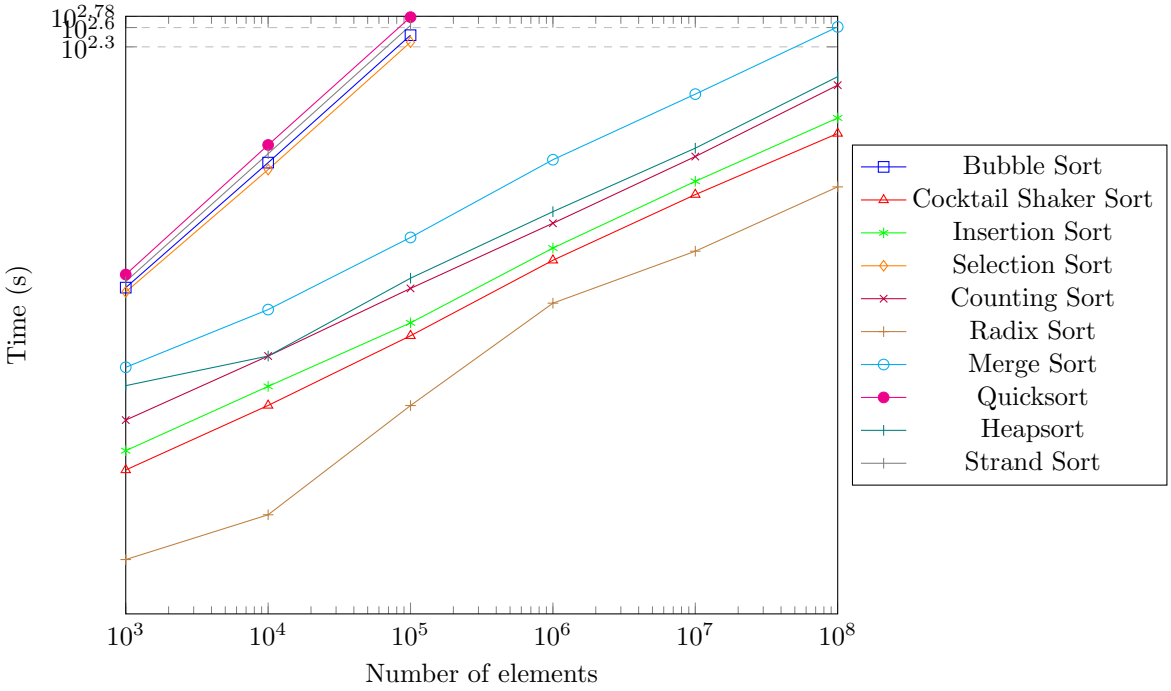

Figure 4: High Density List Sorting Algorithm Performance

Key observations from the graph are as follows:

- Bubble Sort, Selection Sort, Quicksort and Strand Sort exhibit the worst performance, rendering them inefficient for larger datasets.
- The other sorting techniques all showcase remarkable efficiency for high density lists, most notably Radix Sort, followed by Cocktail Shaker Sort and Insertion Sort.
- Counting Sort and Heapsort display great efficiency as well, albeit not as efficient as the previous three.
- Merge Sort's scalability is affected as the input size grows, despite it's good performance for smaller lists.

4.4 Sparse data

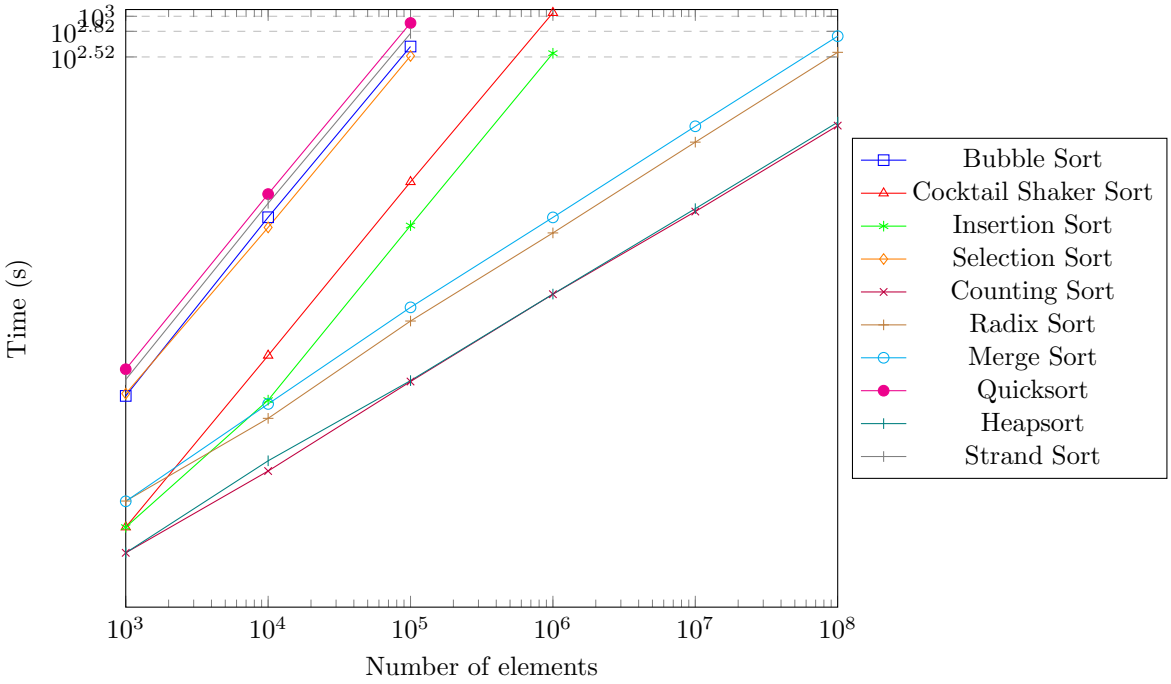

Figure 5: Sparse Data List Sorting Algorithm Performance

Key observations from the graph are as follows:

- Just like in the previous test, Bubble Sort, Selection Sort, Quicksort and Strand Sort all exhibit the worst performance, making them the most inefficient for larger datasets. This is to be expected, as sparse data could be considered in a way a "high density list".
- Cocktail Shaker Sort and Insertion Sort perform better than the previous examples. Their inefficiency becomes apparent on larger datasets, making them an unfavorable choice.
- Radix Sort and Merge Sort display great performance overall, but compared to the remaining two, they are almost 100 times slower.
- The remaining two, Counting Sort and Heapsort, exhibit the best performance out of all the sorting techniques.

4.5 Reverse list

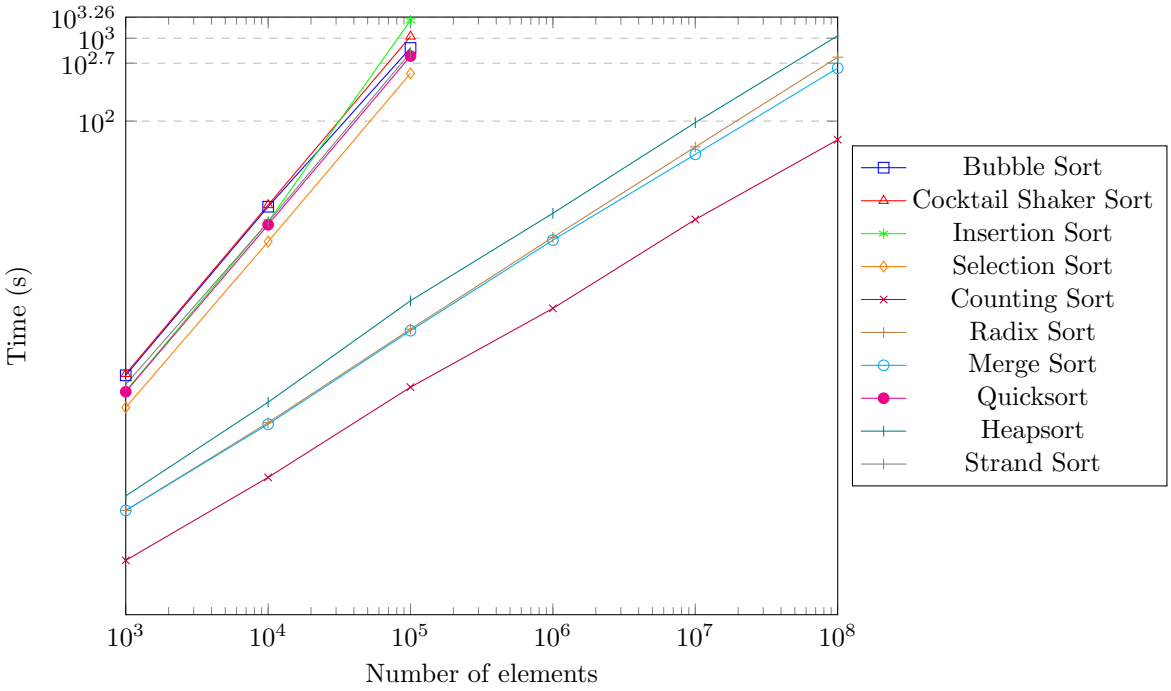

Figure 6: Reverse List Sorting Algorithm Performance

Key observations from the graph are as follows:

- Cocktail Shaker Sort and Insertion Sort display the worst performance.
- Bubble Sort, Strand Sort, Quick Sort and Selection Sort closely follow by, exhibiting relatively poor performance, rendering them inefficient for large datasets with elements sorted in reverse order.
- Heapsort, Radix Sort and Merge Sort showcase efficient performance on reverse lists, with execution time scaling moderately as the input size increases.
- Counting Sort demonstrates superior performance on reverse lists, highlighting it's suitability for large datasets with elements in reverse order.

4.6 Presorted Chunks

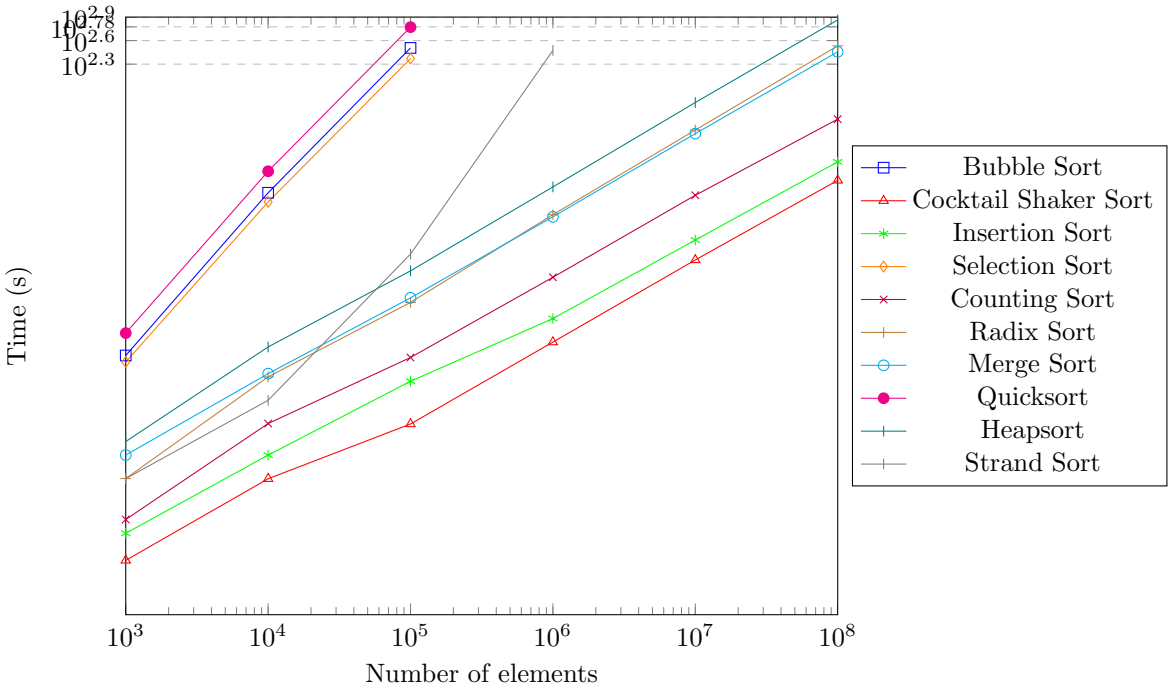

Figure 7: Presorted Chunks Sorting Algorithm Performance

Key observations from the graph are as follows:

- Bubble Sort, Selection Sort and Quicksort perform poorly on datasets with presorted chunks, their execution times increasing significantly as the input size grows.
- Strand Sort, despite it's early similar performance to the other, more efficient sorting techniques, displays an unsuitable scalability.
- Heapsort, Merge Sort and Radix Sort exhibit consistent performance improvements on datasets with presorted chunks. However, their scalability is affected as the input size grows, albeit remaining efficient compared to less optimized. algorithms.
- Counting Sort, Insertion Sort and Cocktail Shaker Sort showcase the most remarkable results, making them the best choices for such task.

4.7 Small lists

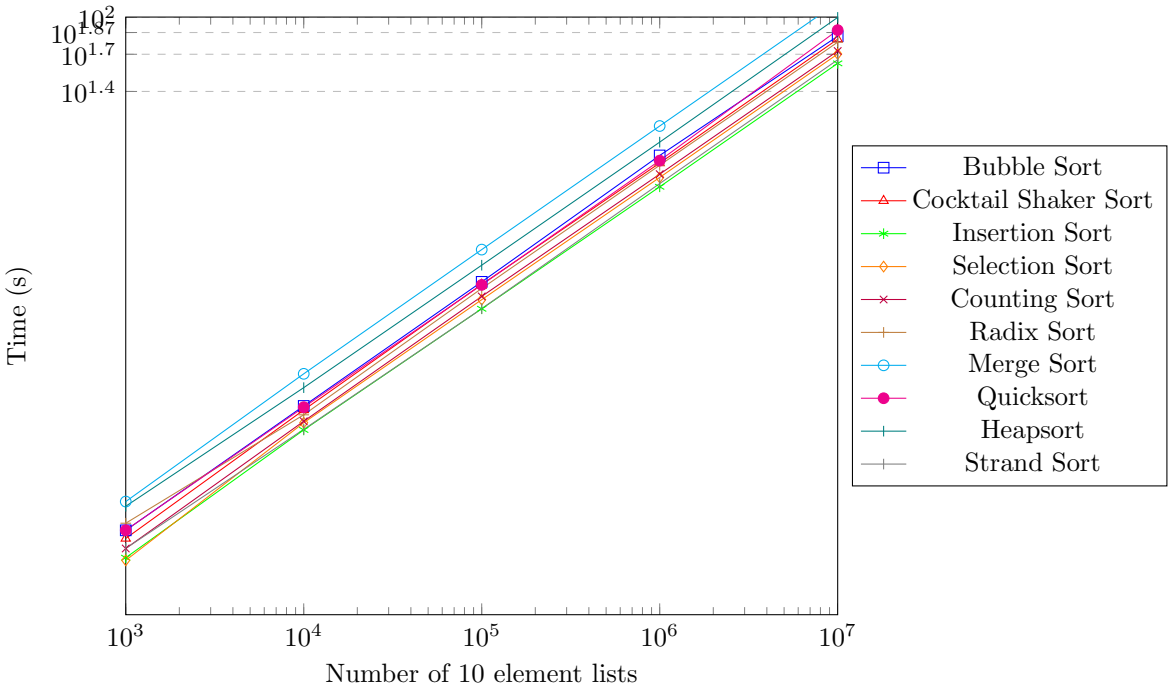

Figure 8: Small Lists Sorting Algorithm Performance

Key insights from the graph include:

- On a first glance, it appears that on a list with 10 elements, all sorting techniques perform similarly. While they all display good performance, as the input size increases, their execution time will be more apparent.
- Merge Sort, Heapsort, Quicksort, Bubble Sort and Cocktail Shaker Sort, compared to the other sorting algorithms, have a poorer performance.
- Selection Sort, Counting Sort and Radix Sort display slightly better performance.
- Strand Sort and, Insertion Sort take the second, respectively first place when tasked with sorting many small lists.

5 Related Work

Comparison of sorting algorithms is a wide research topic, and we have searched for other papers focusing on them, with the intent of comparing, as well as listing some advantages and disadvantages of either side.

This paper[11] is a short, theoretical comparison of sorting algorithms, focusing on describing each sorting technique by average case, worst case and stability, and listing some advantages or disadvantages to each one.

Its size is an *advantage*,it is a quick and easy read for the interested people.

However, its small size could also be interpreted as a *disadvantage*, because it is lacking in raw experiment data and graphs, and it only serves as a general idea of their efficiency.

Overall, this is an interesting read, and it summarizes well the basics of all of the mentioned sorting algorithms, in a neat table format.

This other research paper[12] is a 73 page study on a very wide range of sorting techniques, analyzing each sorting algorithm one by one in a remarkably detailed fashion.

It represents its biggest *advantage*, compared to this research paper, that only focuses on only a couple of sorting algorithms, and in not as great detail.

However, it is also a rather advanced research paper, and it would prove hard to read for people unacquainted with this topic, which could be considered a *disadvantage*.

This is a respectable paper and worth a read for the ones interested in this topic, who are not scared long papers.

6 Conclusions and Future Work

So, in this paper we went through different sorting algorithms to find their unique characteristics, and we concluded that for each task there is a suitable sorting technique to be used.

It is also now understandable why Timsort, the default sorting algorithm for python, is so incredibly efficient; because it uses Merge Sort to split the list into smaller lists, then it uses the one most efficient for small lists: Insertion Sort.

While working on this paper, when writing the code, making it easy to read and understand took time to neatly organize, as we didn't want to have the the code to be filled with redundant lines.

I also attempted to do a comparison using parallel processing, but it did not work. This is due to bad implementation, so it will have to be done another time. It would require much more research on parallel processing on the whole.

Another problem we encountered was a memory error caused by lists of 1 billion elements. This wasn't much of an issue, as this paper focused on comparing all of them together. But continuing the work and testing only the most efficient sorting algorithms up to 1 billion elements could be worthwhile.

References

[1] Jon Bentley. *Column 11: Sorting*. ACM Press / Addison-Wesley, 2nd edition, 2000.

[2] Donald E. Knuth. *Sorting by Exchanging*, volume 3. Addison-Wesley, 1st edition, 1973.

[3] Howard B. Demuth. *Electronic Data Sorting*. PhD thesis, Stanford University, 1956.

[4] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison–Wesley, 1997. Pages 138–141 of Section 5.2.3: Sorting by Selection.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001. See also the historical notes on page 181.

[6] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1997. Section 5.2.5: Sorting by Distribution, pp. 168–179.

[7] Jyrki Katajainen and Jesper Larsson Träff. Algorithms and complexity. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 217–228, Rome, March 1997.

[8] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1997. §5.2.3, Sorting by Selection, pp. 144–155.

[9] Sir Antony Hoare. Computer history museum. Archived from the original on 3 April 2015. Retrieved 22 April 2015.

[10] I. C. (Ishwar Chandra) Gupta and Deepak Jaroliya. *IT enabled practices and emerging management paradigms*. Prestige Institute of Management and Research, 1st edition, 2008. Prestige Institute of Management and Research.

[11] Aditya Dev Mishra and Deepak Garg. Selection of best sorting algorithm. *International Journal of intelligent information Processing*, 2(2):363–368, 2008.

[12] Ramesh Chand Pandey. Study and comparison of various sorting algorithms. *Computer Science and Engineering*, 2008.