Finding the Fastest Sorting Algorithm on Various Data Sets

Francesca Drăguț
Department of Computer Science,
West University,
Timişoara, Romania,
Email: `francesca.dragut01@e-uvt.ro`

Abstract

In this paper, we present the results of comparing Insertion, Merge, Heap, Quick and Bubble Sort. The purpose was finding the fastest algorithm to sort various data sets, organized as lists of integers. In this regard, we took into consideration three file types - random, nearly sorted and decreasing input -, three ranges of numbers - big, medium and narrow - and several input sizes - four of them for overall analysis, three of them for additional graph data and two of them for higher performance analysis. In order to manage the total number of tests, a time tracking algorithm was designed to store the time of each test into a file, from which the results were taken and processed. Theoretical aspects from literature were presented and based on them, a set of expectations was formulated. Analysis of individual results provided the best context in which to use each algorithm, while the overall results provided the fastest algorithm, on average, for all considered data sets. The fastest algorithm, on average, proved to be Merge sort. Only in specific contexts were Quick Sort and Insertion Sort faster. Positive results were found, regarding the better performance of Heap Sort on narrow ranges of numbers and on decreasing input. The main objective of this study was achieved, most of the expectations were met and new results were found regarding Heap Sort.

Keywords— insertion, merge, heap, quick, bubble, sort, various data

1 Introduction

A sorting algorithm is a procedure that solves a sorting problem. A sorting problem receives as input a sequence of n numbers: $(a_1, a_2, ..., a_n)$ and outputs a permutation $(a'_1, a'_2, ..., a'_n)$ of the sequence, such that $(a_1 \leq a_2 \leq ... \leq a_n)$ (see chapter 1.1 in [1]).

People have been sorting for a long time. Nowadays, sorting is considered to be the most fundamental problem in the study of algorithms. It is a study field that is not only of computer science interest. It is also of mathematical, historical, engineering and statistical interest. And the list is open, because sorting appears in every part of our lives. Several of our everyday activities have a sorting element (for example schedules, priorities, deadlines) Similarly, many applications and algorithms have an inherent sorting component (see chapter 1.1 in [1]).

A lot of work has been done regarding sorting algorithms and their comparisons, of which [1] and [2] are turning points in the field.

In [1], each sorting algorithm is presented separately, according to the following framework – introduction to the specific algorithm, a description, a theoretical performance analysis (accompanied by mathematical proofs for time complexity), a range of subchapters including particularities and special cases of the specific algorithm and finally, chapter notes and further reading. There are comparison hints in the book, but generally, each algorithm is treated separately.

In [2], each sorting algorithm is described and implemented in Pascal and C. There is a theoretical analysis of each algorithm, which is backed by bound proofs and practical results, followed by a conclusion. There is a strong focus on the implementation of the algorithms and the results are presented in tables.

However, in the reference literature, the focus is rather on the theoretical comparison and mathematical proof. In [2], there are tables of running times for the algorithms, but only for Pascal and C implementation. The gap detected here is that there are rather little experimental results shown in the literature and even fewer results of Python implementation. Also, there is little work done on showing the behaviour of sorting algorithms on different ranges of numbers. The studies focus mainly on showing how they behave on randomized input for various sizes (see [3] and [4]).

The main objective of this paper is to find the fastest comparison-based sorting algorithm for a list in Python, based on average running time comparisons on various data sets. Another objective is to find the contexts in which each algorithm works the fastest. In these regards, we have designed an algorithm to run tests on Insertion, Merge, Heap, Quick and Bubble Sort for each data set and store the results in a file. The experiments imply using a wide range of data, size-wise and structure-wise.

2 Theoretical Analysis of the Algorithms

In this section, each algorithm is studied individually, from a theoretical point of view. Complexity is considered and expectations are formulated, regarding the time performance.

2.1 Insertion Sort

From a theoretical analysis, we know that Insertion Sort runs in $O(n^2)$ time, as proven in chapter 2 of [1]. Quadratic complexity is manageable for a computer (compared to exponential complexity, for example) and can run on large instances, but still at a significant cost. This is why this algorithm is worth implementing only for relatively small files ($n < 1000$), as explained in chapter 4.1.2 in [2].

In chapter 2.2 in [1], the authors underline that Insertion Sort's running time can vary (considering the same input size), depending on how nearly sorted the file is. This is a natural consequence, as Insertion Sort only creates time cost when it detects two interchanged elements. For ordered elements, it only iterates through the list once and does no extra work.

Expectations:

- Insertion sort should work better on nearly sorted input than on random input and should work worst on decreasing input (E1)[1];
- Insertion sort should take much longer than the algorithms with complexity $O(n\ logn)$ (E2).

2.2 Merge Sort

As mathematically proven in chapter 2.3.2 in [1] and in chapter 4.2.1 in [2], in theory, Merge Sort runs in $O(n\ logn)$ time. In fact, $O(n\ logn)$ is proven to be the lowest achievable bound for comparison-based sorting algorithms [5].

What is interesting about Merge Sort is that even in the worst case, it provides $O(n\ logn)$ running time. Also, for sorted or reversely sorted files, the algorithm will only pass the file once. For these reasons, merge sort is considered to be one of the best options for list sorting (see chapter 4.2.1 in [2]).

Expectations:

- Merge sort should work better on nearly sorted and reversely sorted input (E3);
- Considering that the sorting is done on lists, the expectation is that Merge sort will be among the fastest (E4).

2.3 Heap Sort

Heap sort implements a priority queue. Heaps can be implemented as max-heaps or as min-heaps. In [1], Heap sort is implemented using max-heaps. A max-heap can be viewed as a binary tree, where $A[parent(i)] \geq A[i]$, i being an arbitrary node.

Studies have shown that Heap sort is guaranteed to run in $O(n\ logn)$, even in the worst-case scenario. However, compared to Merge sort, Heap sort does not seem to work significantly better depending on the file type. This means that sorting on a sorted file, for example, is not faster than sorting on any other file (for example considering sortedness in the file - random, nearly sorted, decreasing - or the range in which the numbers in the file are) [2].

[1]The (En) notation will be used to conveniently denote expectations that will be later checked and discussed in the results section.

Expectations:

- Considering that Heap sort guarantees $O(n\, logn)$ even in the worst-case, it is expected to run in about the same amount of time as Merge Sort and Quick Sort (E5);
- However, Heap sort is not known to work better on any particular type of file, so it is expected to run slower than Merge sort (at least on nearly sorted and decreasingly sorted input) (E6).

2.4 Quick Sort

There is an entire philosophy on the implementation of Quick Sort and it all reduces to how the pivot is chosen. In a previous paper, we implemented Quick Sort in a recursive manner, but the recursion was too deep for some files, especially the ones where the range was (0,5) or where the numbers were not randomly distributed. In order to fix this problem, we implemented an iterative version of Quick Sort in this paper, presented in [6].

Apart from these details, Quick sort has other particularities, as well.

First, even though the worst case of Quick sort is $O(n^2)$, the average case is closer to what happens in practice. The worst case occurs when the pivot is the last element, therefore the partition is done on a subarray with $n-1$ elements and the other one is done on a subarray with 0 elements. In practice, though, it is unlikely for all the partitions to be bad, since the partitions are a mix of good and bad splits. This is intuitively and formally shown in figure and chapter 7.4 from [1].

Second, Quick sort has a memory advantage over the similarly designed Merge Sort (as proven in [7]). Even though theoretically, Merge Sort has better time complexity, Quick Sort exploits cache locality better, as opposed to Merge sort, which uses a significant amount of extra memory (as seen in [8]).

Third, the worst case occurs when the file is already sorted (see chapter 4.1.3 in [2]).

Fourth, Quick Sort works best on randomized input, as shown in chapter 7.3 from [1].

Expectations:

- Quick sort should run fastest on randomized input and slowest on nearly sorted input (E7);
- Quick sort should run fast and should be comparable to Merge and Heap sort. Given the more efficient memory allocation and the same average run time, it should be faster than Merge and Heap sort (E8).

2.5 Bubble Sort

In [9], it is shown why the running time of Bubble Sort is $O(n^2)$. It uses two nested `for` loops, which means that for each current element in the first loop, the algorithm will iterate n times, leading to an $O(n^2)$ complexity. Even though it is slow, Bubble Sort is a popular sorting algorithm, because the implementation is quite simple.

It is also efficient on sorted input, working in $O(n)$ time and it works well on nearly sorted input [10]. This happens because Bubble sort only interchanges the pairs of adjacent elements placed incorrectly [2]. Consequently, if there are no pairs in reversed order, the algorithm will just pass through the input, in linear time, making no changes, thus no work.

Expectations:

- The algorithm should run faster on nearly sorted input than in any other case (E9);
- Bubble sort is expected to run the slowest out of all algorithms, given the amount of interchanges it has to make (E10).

2.6 Time Keeping Algorithm

For the files of sizes 10^2 up to 10^5, we designed an algorithm to help us manage the files and do the sorting procedures in an automated way. This algorithm consists of two parts: a part handling file names, so that the files should not be manually introduced for each step and another part, which does each sort and writes the running time in an output file.

The complexity of the file naming part is insignificant, as it does precisely 63 steps. The consistent time is consumed by the second part. For each of the 63 files, there are executed 5 sequential procedures. Considering the average running time of each procedure, we would have 3 procedures to run in $O(n\,logn)$ time (Merge, Heap and Quick Sort) and 2 procedures to run in $O(n^2)$ time, where n is considered to be the size of the file at a given point.

Thus, the overall complexity of the algorithm should be $63\,(3\,O(n\,logn) + 2\,O(n^2))$, where n is the average size of the files.

3 Experimental Design

The goal of this paper is to show a comparison between sorting algorithms. This cannot be done without the experimental part, where the theoretical aspects are checked.

The first step of the experimental design was collecting the pseudo codes, which were taken from [1].

The second step was translating the pseudo codes into Python language, in the PyCharm environment.

The third step was creating the files. For the experiments, we used 3 types of files, regarding the input randomness in the file: pure random, nearly sorted and sorted decreasingly. Each file type was associated with 3 different ranges of numbers: (-10^8,10^8), (-10^4,10^4), (0,5). For each range, we considered seven sizes. We used the first four (sizes of 10^2, 10^3, 10^4 and 10^5) for a comparison between all of the algorithms. We only used sizes $4 * 10^4, 6 * 10^4, 8 * 10^4$ for providing supplementary data for the graphs. Additionally, we added two sizes, namely 10^6 and 10^7, which we used for comparing the most efficient algorithms on a large scale perspective.

The pure random files were generated by a program, using a code of this template (the listed code is for a file of size 100, in a number range of 0 up to 5):

```python
import random
with open("file.txt", "w") as f:
    for x in range(100):
        print(random.randint(0,5), file=f)
```

The nearly sorted files were based on the sorted files from previous tests, with 5 elements manually moved randomly in other parts of the file. And the decreasing sorted files were obtained by running a variation of Merge Sort on the randomly generated files.

We used the same files on all of the algorithms, to provide the same preconditions. We tested each algorithm on a total of 63 files (of sizes 10^2 up to 10^5, $4 \cdot 10^4, 6 \cdot 10^4, 8 \cdot 10^4$). We tested additional files of sizes 10^6 and 10^7 on Merge Sort and Heap Sort, for all of the file types. We also tested these files on Insertion Sort in the cases of nearly sorted input.

3.1 Time Keeping Algorithm Design

The time keeping algorithm was the final part of the program.

The total number of files reached 63. This led to a total of 315 tests. In order to make file handling manageable, we named the files in a suggestive manner.

The first part of the name referred to the range: 'big' (for $(-10^8, 10^8)$), 'med' (for $(-10^4, 10^4)$) and 'few' (for (0,5)). The second part referred to the level of sortedness in each file: 'R' (pure random files), 'NS' (nearly sorted), 'D' (decreasingly sorted). And the last part consisted of the size of the file.

We created a file-naming algorithm, to build an array with strings consisting of file names. Then, the array was parsed and the content of each file was processed and sorted. The code is listed below.

```
ranges = ['big','med','few']
types = ['R','NS','D']
lengths = ['100','1000','10000','40000','60000','80000','100000']

files = []
for i in range(len(ranges)):
    for j in range(len(types)):
        for k in range(len(lengths)):
            string = ranges[i] + types[j] + lengths[k] + '.txt'
            files.append(string)
```

For each file in the list of files created above, the file was processed, being converted into a list of integers. Then, a start time was set, the sorting procedure was called and an end time was defined, This sequence was used for all of the algorithms, sequentially. The output file contained the name of the file, which contained all information about it, the name of the algorithm and the time.

```
with open("out.txt","w") as f1:
    for file_name in files:
        with open(file_name,"r") as f:
            array = f.read().split()
            for i in range(0, len(array)):
                array[i] = int(array[i])

            for i in range(5):
                start = time.time()
                if i == 0: mergesort(array); algorithm = "merge"
                if i == 1: heapsort(array); algorithm = "heap"
                if i == 2: quicksort(array); algorithm = "quick"
                if i == 3: insertion(array); algorithm = "insertion"
                if i == 4: bubblesort(array); algorithm = "bubble"
                end = time.time()
                print(file_name, algorithm, end - start, file=f1)
```

We took the running times from the output file and stored them into tables, which we processed in order to outline specific features for each algorithm. The resulting tables were the primary tool for the overall comparison of the algorithms.

We ran this program in the following conditions. The Operating System is Windows 10, on an Asus UX305U Notebook PC. The processor is Intel® Core™ i5-6200U CPU @ 2.30 Ghz, 2400 Mhz, 2 Cores, 4 Logical Processors. The installed RAM is 8GB, out of which 1.30 GB available.

4 Results

This section is composed of two subsections: one where results for each algorithm are presented and another one where the overall results are shown.

The results are presented in tables and graphs and they are interpreted. Also, the expectations formulated in section 2 are checked in this section.

4.1 Algorithm Oriented Results

In this section, we present the tables for each algorithm. There are three tables for each algorithm, each corresponding to a file type (random, nearly sorted, decreasing). The tables contain the results for sizes $10^2, 10^3, 10^4, 10^5$, for ranges $(-10^8, 10^8)$, $(-10^4, 10^4)$, (0,5).

The row *Col.avg.* contains the averages for each range, done on results obtained for each size.

The column *Row avg.* contains the average for each size, done on results obtained for each range.

The cell placed in the right bottom corner represents the row average of the row *Col.avg.*. This means that the cell contains the total average, used as an overall performance index between the three tables (random, nearly sorted and decreasing), in order to see on which file type does the algorithm run faster.

4.1.1 Insertion Sort

Insertion Sort worked best on nearly sorted input, as seen in Table 2. This is because Insertion Sort interchanges any pair of unordered elements. Since there were few such pairs, the algorithm worked the fastest on this input, considering not only the other cases (see Table 1 and Table 30, but all of the algorithms.

As expected in (E2), Insertion Sort worked best on nearly sorted input and worst on decreasing input (see Table 3).

Random input - Insertion sort				
Input size	Range			Row avg.
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.108	0.169	0.149	0.142
10000	11.531	10.453	9.124	10.369
100000	725.026	736.914	814.522	758.821
Col. avg.	184.16625	186.884	205.9488	192.333

Table 1: Results for Insertion Sort on random input.

Nearly sorted input - Insertion sort				
Input size	Range			
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	Row avg.
100	0.000	0.004	0.000	0.001
1000	0.003	0.006	0.000	0.003
10000	0.042	0.034	0.032	0.036
100000	0.241	0.245	0.230	0.239
Col. avg.	0.072	0.072	0.066	0.070

Table 2: Results for Insertion Sort on nearly sorted input.

Decreasing input - Insertion sort				
Input size	Range			Row avg.
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
100	0.001	0.002	0.002	0.002
1000	0.200	0.171	0.140	0.170
10000	14.915	14.818	12.324	14.019
100000	1222.326	1249.334	1002.683	1158.114
Col. avg.	309.360	316.081	253.787	293.076

Table 3: Results for Insertion Sort on decreasing input.

4.1.2 Merge Sort

The results for Merge Sort were rather uniform. The differences were slight, but as seen in Table 5, Merge Sort had the best results on nearly sorted input, which was expected in (E4). Nevertheless, it had the worst results on decreasing input (see Table 6), contrary to our expectations. It also had satisfactory results on random input, as seen in Table 4.

Random input - Merge sort				
Input size	Range			Row avg.
	(-$10^8, 10^8$)	(-$10^4, 10^4$)	(0,5)	
100	0.000	0.005	0.000	0.002
1000	0.020	0.019	0.010	0.016
10000	0.189	0.164	0.133	0.162
100000	1.060	1.289	1.057	1.135
Col.avg.	0.317	0.369	0.300	0.329

Table 4: Results for Merge Sort on random input.

Nearly sorted input - Merge sort				
Input size	Range			Row avg.
	(-$10^8, 10^8$)	(-$10^4, 10^4$)	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.031	0.015	0.015	0.020
10000	0.100	0.115	0.113	0.109
100000	1.226	0.935	1.188	1.116
Col.avg.	0.339	0.266	0.329	0.312

Table 5: Results for Merge Sort on nearly sorted input.

Decreasing input - Merge sort				
Input size	Range			Row avg.
	(-$10^8, 10^8$)	(-$10^4, 10^4$)	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.015	0.040	0.005	0.020
10000	0.122	0.115	0.117	0.118
100000	1.582	1.144	0.892	1.206
Col.avg.	0.429	0.324	0.253	0.336

Table 6: Results for Merge Sort on decreasing input.

4.1.3 Heap Sort

Heap Sort presented a positive finding. As seen in Table 7, Table 8 and Table 9, the running time decreased significantly as the range narrowed. In Table 8 and Table 9, the time on the range (0,5) was nearly half of the running time on range $(-10^8, 10^8)$.

Considering the file type, Heap Sort worked best on decreasing input and worst on nearly sorted input. In (E6) we expected that Heap Sort should not work significantly better depending on the file type. We made another positive finding here, as the results on decreasing input were significantly better than the ones on nearly sorted input and slightly better than the results on random input.

Random input - Heap sort				
Input size	Range			Row avg.
	(-$10^8, 10^8$)	(-$10^4, 10^4$)	(0,5)	
100	0.000	0.003	0.005	0.003
1000	0.024	0.024	0.015	0.021
10000	0.204	0.171	0.181	0.185
100000	1.550	1.501	1.198	1.416
Col.avg.	0.445	0.425	0.350	0.406

Table 7: Results for Heap Sort on random input.

Nearly sorted input - Heap sort				
Input size	Range			Row avg.
	$(-10^8, 10^8)$	$(-10^4, 10^4)$	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.038	0.023	0.015	0.025
10000	0.209	0.219	0.164	0.197
100000	2.318	1.520	1.111	1.650
Col.avg.	0.855	0.587	0.430	0.624

Table 8: Results for Heap Sort on nearly sorted input.

Decreasing input - Heap sort				
Input size	Range			Row avg.
	$(-10^8, 10^8)$	$(-10^4, 10^4)$	(0,5)	
100	0.000	0.015	0.000	0.005
1000	0.028	0.031	0.007	0.022
10000	0.155	0.162	0.103	0.140
100000	1.860	1.278	1.026	1.388
Col.avg.	0.511	0.372	0.284	0.389

Table 9: Results for Heap Sort on decreasing input.

4.1.4 Quick Sort

Quick Sort raised a number of issues. In a previous paper, we implemented the recursive version of the algorithm, which failed to produce any results on nearly sorted input and on decreasing input.

Even though in (E7), we expected it to be faster than Merge Sort and Heap Sort, due to better cache allocation, Quick Sort met this expectation only on random input in range $(-10^8, 10^8)$ and $(-10^4, 10^4)$, as seen in Table 10. In all of the other cases (see Table 11 and Table 12), Quick Sort proved to be exceedingly slow.

The unexpected behaviour of Quick Sort might come from too deep recursions on range (0,5). In (E8), we expected it to run fastest on randomized input, but we did not expect such a leap from the best results on random input on large ranges to some of the worst results in the other cases.

Random input - Quick sort				
Input size	Range			Row avg.
	$(-10^8, 10^8)$	$(-10^4, 10^4)$	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.012	0.015	0.047	0.025
10000	0.100	0.113	3.919	1.377
100000	0.664	0.654	248.977	83.432
Col.avg.	0.194	0.196	63.236	21.208

Table 10: Results for Quick Sort on random input.

Nearly sorted input - Quick sort				
Input size	Range			Row avg.
	$(-10^8, 10^8)$	$(-10^4, 10^4)$	(0,5)	
100	0.008	0.007	0.000	0.005
1000	0.199	0.124	0.184	0.169
10000	5.885	3.399	11.845	7.043
100000	142.827	896.513	527.895	522.411
Col.avg.	37.229	225.010	134.981	132.407

Table 11: Results for Quick Sort on nearly sorted input.

Decreasing input - Quick sort				
Input size	Range			Row avg.
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.119	0.131	0.078	0.109
10000	7.845	6.613	4.964	6.474
100000	1034.194	504.173	255.781	598.049
Col.avg.	260.540	127.729	65.206	151.158

Table 12: Results for Quick Sort on decreasing input.

4.1.5 Bubble Sort

Bubble Sort proved to be the slowest algorithm from the ones we studied. We obtained the best results on nearly sorted input, as seen in Table 14. The average running time doubled on random input, as seen in Table 13 and almost tripled on decreasing input, as seen in Table 15.

An interesting fact to notice is that on input size 100, the running time was insignificant. Also, for input size 1000, the results were reasonable. For input sizes smaller than 1000, Bubble Sort is still a feasible choice, as it is undemanding to implement.

Pure random bubble time				
Input size	Range			Row avg.
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
100	0.002	0.002	0.000	0.001
1000	0.241	0.222	0.154	0.206
10000	20.379	17.679	16.697	18.252
100000	1471.978	1434.792	1142.735	1349.835
Col.avg.	373.150	363.174	289.897	342.073

Table 13: Results for Bubble Sort on random input.

Nearly sorted input - Bubble sort				
Input size	Range			Row avg.
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
100	0.002	0.003	0.001	0.002
1000	0.174	0.107	0.105	0.129
10000	9.013	9.578	8.541	9.044
100000	650.419	707.113	705.064	687.532
Col. avg.	164.902	179.200	178.428	174.177

Table 14: Results for Bubble Sort on nearly sorted input.

Decreasing input - Bubble sort				
Input size	Range			Row avg.
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
100	0.005	0.004	0.002	0.004
1000	0.287	0.249	0.239	0.258
10000	21.692	29.210	22.012	24.305
100000	2289.537	2027.947	1723.231	2013.572
Col. avg.	577.880	514.353	436.371	509.535

Table 15: Results for Bubble Sort on decreasing input.

4.2 Overall Results

The overall results consider those results that include all of the studied algorithms. They consist of average running times for the file types (average of results on ranges) for the algorithms themselves (average of results on average running times on file types).

For a visual representation, we plotted three graphs, each corresponding to a file type. The results for size 100 were not presented in the graph, because they were all either 0 or almost 0.

Random input - Quick sort				
Input size	Range			Row avg.
	(-10^8,10^8)	(-10^4,10^4)	(0,5)	
100	0.000	0.000	0.000	0.000
1000	0.012	0.015	0.047	0.025
10000	0.100	0.113	3.919	1.377
40000	0.379	0.372	39.611	13.454
60000	0.507	0.571	69.841	23.640
80000	0.687	0.732	122.431	41.283
100000	0.847	1.231	209.716	70.598

Table 16: Example of how the results are stored into tables for each algorithm. The blue column *"Row avg."* was plotted on the graph in *Figure 1*.

In Table 16, we can see how the graphs were plotted. We see an example of a results table, for Quick Sort, on random input, where the results on the ranges were averaged for each size. We plotted the average results (blue column) for each corresponding size on the graph. The same steps were used on the other algorithms, as well.

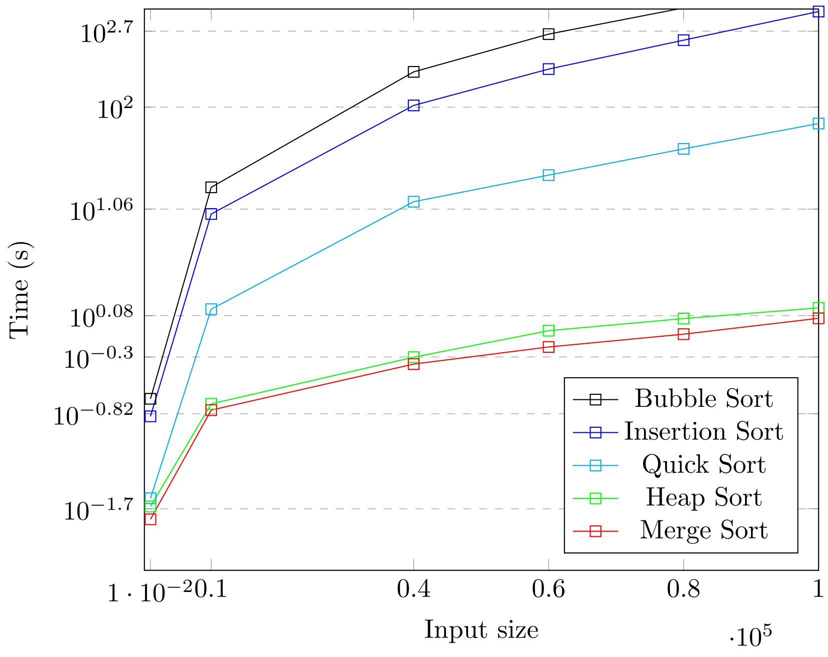

Figure 1: Random input

In the first graph, in Figure 1, we see the results for randomly distributed numbers. Reading the graph, we observe that on random input, Merge Sort was the fastest, followed closely by Heap Sort. Quick Sort was in the middle, even though it obtained the best results for this file type on ranges (-10^8, 10^8) and (-10^4, 10^4) (see Table 10 for Quick Sort and tables 1, 4, 7 and 13 for a comparison). The results on range (0,5) decreased the average performance of Quick Sort on random input, as the results on this range were about 325 times larger than on the first two ranges. Insertion Sort was on the fourth place, followed by Bubble Sort, both with exceptionally high running times.

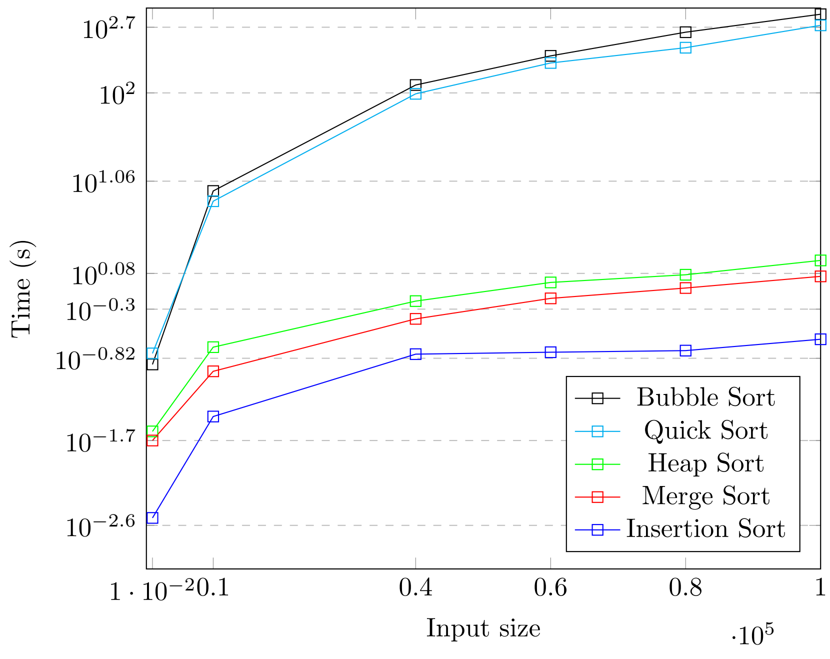

Figure 2: Nearly sorted input

In the second graph, in Figure 2, we see the results for nearly sorted input. We observe that in this case, Insertion Sort was the fastest. There are small variations in this line on the graph, which arise from the level of sortedness of the files. Merge Sort and Heap Sort still had high performance. However, Quick Sort ran significantly slow on this file type, being comparably close to Bubble Sort, which worked best on nearly sorted input.

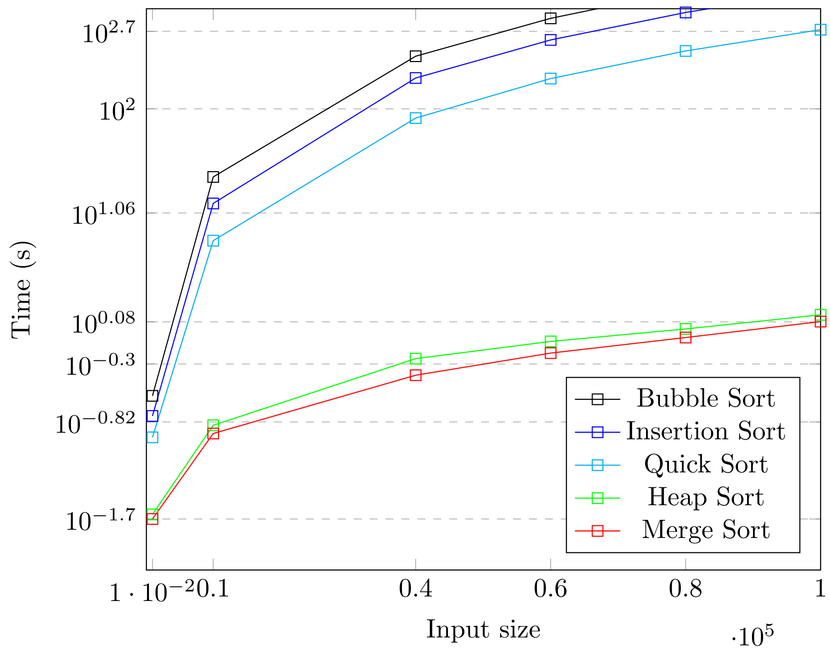

Figure 3: Decreasing input

In the third graph, in Figure 3, we see the results for decreasing input. Merge Sort was the fastest, followed closely by Heap Sort. Quick Sort was on the third place, but it is quite close in the graph to Insertion Sort and Bubble Sort, which both ran the slowest on decreasing input.
What remains constant in the three graphs is that Merge Sort is consistent, having the best results for random and decreasing input. Merge Sort is also a good choice on nearly sorted input, but here Insertion Sort yields to be the best choice. Another constant in the three graphs is the poor performance of Bubble

Sort and the rather poor performance of Quick Sort, with the exception of random input, in Figure 1.

The averages computed in Tables 1–15, precisely the values in the row *Col.Avg.*, were placed into tables, similar to Table 17. Table 17 is one of the three such tables: one for random input, one for nearly sorted input and one for decreasing input. The column *Row avg.* from each of the three tables was placed to the corresponding file type from Table 18, where we can see the behaviour of the algorithms on each file type and the average of their running times, in the column *Row avg.*.

Algorithm	Average results on ranges for each algorithm on random input			Row avg.
	Range			
	(-10^8, 10^8)	(-10^4, 10^4)	(0,5)	
Insertion	184.166	186.884	205.949	192.333
Merge	0.317	0.369	0.300	0.329
Heap	0.045	0.425	0.349	0.273
Quick	0.194	0.195	63.235	21.208
Bubble	373.150	363.174	289.897	342.074

Table 17: Average results on ranges for each algorithm on random input.

As we can observe from Table 18, Merge Sort proves to be the fastest sorting algorithm, considering the three ranges, the three file types and the four input sizes. In this regard, Merge Sort exceeds expectation (E4), being the fastest algorithm in our study. Merge Sort is closely followed by Heap Sort. This checks (E5), as Heap Sort is almost as fast as Merge Sort. Both of these algorithms proved to run in a remarkably short time for all data sets. Quick Sort is on the third place, but the difference between Quick Sort and Heap Sort, which is on the second place is highly significant, as it is about 267 times larger. Quick Sort proved to be the best option for random input combined with wide range (see Table 10), but the average performance is poor, contradicting expectation (E8). Insertion Sort stands close to Quick Sort, being about 1.4 times slower. Insertion Sort proved to be the best option for nearly sorted input, but its average performance is about 440 worse than the performance of Merge and Heap Sort. On the last place in our ranking is Bubble Sort, which is about 2.3 times slower than Insertion Sort and about 1050 times slower than Merge Sort. The performance was poor on all data sets, with no exception, presenting advantages only for sizes below 10^3, due to the undemanding implementation.

Overall average				
Algorithm	File type			Row avg.
	Random	Nearly sorted	Decreasing	
Insertion	192.333	0.070	239.076	143.826
Merge	0.329	0.311	0.336	0.325
Heap	0.273	0.468	0.389	0.377
Quick	21.208	132.407	151.158	101.591
Bubble	342.074	174.177	509.535	341.929

Table 18: Overall average. The average of the random, nearly sorted and decreasing input was computed, over the 3 ranges and the results were placed in this table, which were in turn, averaged (Row avg.).

In the last table, Table 19, we see which algorithms managed to produce results on sizes 10^6 and 10^7, for each range and file type. As expected, Merge Sort and Heap Sort managed to produce good results on all of these inputs. For nearly sorted input, Insertion Sort also provided the best results in this study. As above, on smaller sizes, the ranking is the same, with Merge Sort being on the first place (except for nearly sorted input), and Heap Sort following.

We can notice that the performance of Merge Sort is the best on decreasing input, followed by nearly sorted input. This means that (E3) is partially met, because this case was not true for smaller sizes. However, the larger the input is, the more visible the differences become.

In the case of Heap Sort, we can notice again how the time decreases as the range narrows. The best performances of Heap Sort are on the range (0,5), regardless of the file type.

Algorithm	Range			Row avg.
	(-10^8,10^8)	(-10^4,10^4)	(0,5)	
Random input				
Merge	101.890	65.500	75.512	80.967
Heap	126.954	122.854	88.995	112.934
Nearly sorted input				
Insertion	12.653	7.882	9.240	9.925
Merge	61.665	62.857	60.819	61.780
Heap	99.899	93.050	75.491	89.480
Decreasing input				
Merge	55.581	66.472	52.456	58.170
Heap	94.361	89.829	74.388	86.193

Table 19: Average results for sizes 10^6 and 10^7 for the algorithms that managed to run on these inputs.

5 Related Work

In this study we wanted to find the fastest sorting algorithm on various data sets. The gap noticed in literature was the lack of results for Python implementations on shorter ranges and unrandomized data sets. However, these studies provided us with a starting point and we can relate our work to them.

The article that helped us significantly in terms of methodology was [3]. Even though the studied algorithms were slightly different (Insertion, Quick and Bubble Sort were included, but instead of Merge and Heap Sort, the author studied Shell and Selection Sort). The approach was similar, as comparison tables and graphs were built and the criterion of comparison was the running time. However, since the study was performed on randomized lists, Quick Sort was the first in the ranking.

Another article that our study can relate to is [**?**]. This study shows the comparison of Merge, Quick, Insertion, Bubble and Selection Sort. A strong point of this article is that each algorithm is run on each data set five times, to provide accuracy of the results, even though the data sets were not varied.

We found the mentioned articles to be the most related to our study. We found similarities especially in the work frame and how we processed the results and we also found differences, especially in the way we considered data sets. However, each study has its individual purpose and considering different approaches only contributes to widening the limits of knowledge.

6 Conclusions and Future Work

The problem we addressed in this paper is finding the fastest comparison-based sorting algorithm, on average, for lists in Python. In order to establish this, we implemented a program that contains the subroutines for the five studied algorithms (Insertion, Merge, Heap, Quick and Bubble Sort) and a time-keeping algorithm. As there were 315 tests to be done, the time-keeping algorithm automated the process.

We considered a wide variety of data, regarding the sortedness of the file, the range of the numbers in the file and the input size. This broadened the perspective on how each algorithm behaves in different contexts.

For further analysis, the results were stored into tables and processed, through computing different averages of the results. We have computed averages for input sizes, where we averaged the results for each range, but also averages for ranges, where we averaged the results for each size. We have shown the relevant results for each algorithm, individually and we have presented an overall comparison.

As a result of analyzing the results, Merge sort proved to be the fastest. Its time performance was only exceeded by Quick sort (for random input in ranges (-$10^8, 10^8$) and (-$10^4, 10^4$)) and by Insertion Sort in the nearly sorted case. We have proven that contrary to our expectations, Quick sort performs poorly on average and its use is only justified under certain conditions. The running time of Heap sort was close to Merge sort, as they both guarantee $O(nlogn)$ worst-case complexity, the difference lying in implementation subtleties. Even though Insertion sort is known to be an inefficient algorithm, because of the quadratic complexity, it performed better on nearly sorted input than any algorithm on any other data set. In the other cases, it was slow and inefficient. Bubble sort was the slowest algorithm, its advantages lying only in the simplicity of the implementation.

In section 2, we formulated ten expectations, out of which eight were met (E1, E2, E4, E5, E7, E8, E9, E10), (E1) was exceeded and two were not met exactly as stated, (E3) and (E6). Regarding (E1), Insertion sort not only worked best on nearly sorted input, but it produced the best results out of all algorithms on any other data set. However, (E3) was met, but only for inputs large enough (of sizes 10^6 and 10^7) to make a significant difference.

Regarding (E6), a positive discovery was made, regarding the better performance of Heap sort on narrow ranges. It is a clear pattern that Heap Sort worked significantly better on the range (0,5), on all file types, than on wider ranges.

Future work will be done in order to find ways to make Quick Sort more flexible. Even though we implemented an iterative version of Quick Sort in this study, it still did not raise to our expectations. We will pursue further implementations, test them and compare the results.

Acknowledgements

We would like to mention [11] for the detailed help in structuring and writing this paper.

References

[1] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Aglorithms. Third Edition.* The MIT Press, 2009.

[2] G. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison - Wesley Publishing Company, 1991.

[3] Juliana Peña Ocampo. An empirical comparison of the runtime of five sorting algorithms. 2008.

[4] Muhammad Idrees. Sorting Algorithms – A Comparative Study. *International Journal of Computer Science and Information Security*, 14:930–936, 2016.

[5] David Luebke. Cs 332: Algorithms. linear-time sorting algorithms. University Lecture, 2013.

[6] Razvan Certezeanu, Sophia Drossopoulou, Benjamin Egelund-Muller, K. Rustan M. Leino, Sinduran Sivarajan, and Mark Wheelhouse. Quicksort Revisited. Verifying Alternative Versions of Quicksort. *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday Lecture Notes in Computer Science*, 9660:407–426, 2016.

[7] Oluwakemi Abikoye, Aderonke Kayode, Akande Oluwatobi, Esau Oladipupo, and Adeniyi Jide. COMPARATIVE STUDY OF TWO DIVIDE AND CONQUER SORTING ALGORITHMS: QUICKSORT AND MERGESORT. *Procedia Computer Science*, 171:2532–2540, 2020.

[8] Davide Pasetto and Albert Akhriev. A comparative study of parallel sort algorithms. pages 203–204, 2011.

[9] Ping Yu, You Yang, and Yan Gan. Bubble Sort: An Archaeological Algorithmic Analysis. 154:95–102, 2012.

[10] Owen Astrachan. Experiment Analysis on the Bubble Sort Algorithm and Its Improved Algorithms. *SIGCSE Bulletin (Association for Computing Machinery, Special Interest Group on Computer Science Education)*, 2003.

[11] Tomislav Hengl and Michael Gould. Rules of Thumb for Writing Reseach Articles. 2002.