Sorting Algorithms and Their Execution Times an Empirical Evaluation

Guillermo O. Pizarro-Vasquez[1(✉)], Fabiola Mejia Morales[1], Pierina Galvez Minervini[1], and Miguel Botto-Tobar[2,3]

[1] Salesian Polytechnic University, Guayaquil, Ecuador
gpizarro@ups.edu.ec
[2] Universidad de Guayaquil, Guayaquil, Ecuador
[3] Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. One of the main topics in computer science is how to perform data classification without requiring plenty of resources and time. The sorting algorithms Quicksort, Mergesort, Timsort, Heapsort, Bubblesort, Insertion Sort, Selection Sort, Tree Sort, Shell Sort, Radix Sort, Counting Sort, are the most recognized and used. The existence of different sorting algorithm options led us to ask: What is the algorithm that us better execution times? Under this context, it was necessary to understand the various sorting algorithms in C and Python programming language to evaluate them and determine which one has the shortest execution time. We implement algorithms that help create four types of integer arrays (random, almost ordered, inverted, and few unique). We implement eleven classification algorithms to record each execution time, using different elements and iterations to verify the accuracy. We carry out the research using the integrated development environments Dev-C++ 5.11 and Sublime Text 3. The products allow us to identify different situations in which each algorithm shows better execution times.

Keywords: Sorting · Sorting algorithms · Standard dataset · Integrated development environment · Execution time

1 Introduction

One of the fundamental issues related to computer science is how to perform data sorting without requiring a lot of resources and time. We can define sorting as organizing a disordered collection of items to increase or decrease order [1]. Sorting and, by extension, sorting algorithms are critical to several tasks. Sorting algorithms can help remove or merge data through sorting by the primary uniqueness criterion; they are also useful in finding out where two broad sets of elements differ. By the same logic, sorting algorithms can also determine which data appears in both datasets.

Over time sorting algorithms have been implemented in almost all programming languages; therefore, they combine multiple language components with helping new programmers learn how to code. Additionally, it is nearly impossible to discuss sorting without mentioning performance. Performance is the key for all systems to function

M. Botto-Tobar et al. (Eds.): ICAETT 2020, AISC 1302, pp. 335–348, 2021.
https://doi.org/10.1007/978-3-030-63665-4_27

efficiently; thus, we can claim that sorting helps us understand performance, which leads to an improvement in software structures and designs.

Since the early days of computer science, the sorting and classifying problem has been a prevalent topic of research due to the complexity of efficiently using precise and straightforward coding statements.

One of the purposes intended to achieve using sorting is to minimize the execution time of a group of tasks. Hence multiple algorithms have been developed and improved to sort faster, and for this, it is necessary to know the computer specifications, program design methodology, and software architecture [2].

Some algorithms can be very complex depending on their execution; as an example, we have the "Bubblesort" that since 1956 is the subject of study [3], and that can be very complex compared to the "ShellSort" that presents less execution time required to perform a sorting [4].

We selected the sorting algorithms Quicksort, Mergesort, Timsort, Heapsort, Bubblesort, Insertion Sort, Selection Sort, Tree Sort, Shell Sort, Radix Sort, Counting Sort, because they are the most recognized and used around the world. In this research, it is necessary to stipulate that we do not implement memory management algorithms, as we will only measure the performance of the classification algorithms without additional code.

There are two types of sorting data, internal sorting and external sorting. Internal sorting methods store sorted values in main memory; therefore, we assume that the time required to access any item is the same. On the other hand, external sorting methods store the values to sort in secondary memory; Assuming that the time required to access any item depends on the last position obtained.

The classification algorithms have two classifications, which are comparative and non-comparative [5]. In the comparison-based sorting algorithm, the disordered data is sort by comparing the data pairs repeatedly. If the data is out of order, they are interchanged with each other [6]. This exchange operation of this sort is known as a comparison exchange. Non-comparison ordering algorithms are responsible for classifying data using the data's specific well-established properties, such as data distribution or binary representation [7]. Four parameters are necessary for the sorting algorithms, which are determined: stability, adaptability, time complexity, space complexity [8].

Comparison-based sorting algorithms generally have two subdivisions: complexity $O(n^2)$ and complexity $O(n \log n)$. In general, the $O(n^2)$ sorting algorithms have a slower execution than the $O(n \log n)$ algorithms; despite this, the $O(n^2)$ sorting algorithms are still fundamental in computer science. One of $O(n^2)$ algorithms' benefits is that they are non-recursive, requiring much less RAM. Another application of the $O(n^2)$ ordering algorithm is in the sorting of small matrices. Because the $O(n \log n)$ sorting algorithms are recursive, it is inappropriate to sort small arrays as they perform poorly (Table 1).

We can highlight that among the $O(n^2)$ sorting algorithms, Selection Sort and Insertion Sort are the best-performing algorithms in general data distributions [9].

Several authors have carried out experiments to define which one or which of these algorithms have better execution times; most of them indicate that Quicksort is the ideal one; however, the authors of these experiments only venture to make comparisons between a maximum of 9 sorting algorithms at a time [10–13]. Also, it is necessary to

Table 1. Sorting algorithms used and their complexity.

Sorting algorithms	Complexity	Memory	Method
Bubblesort [BS]	$O(n^2)$	O(1)	Exchanging
Insertion sort [IS]	$O(n^2)$	O(1)	Insertion
Counting sort [CS]	O(n + k)	O(n + k)	Non-comparison
Mergesort [MS]	O(n log n)	O(n)	Merging
Tree sort [TrS]	O(n log n)	O(n)	Insertion
Radix sort [RS]	O(nk)	O(n)	Non-comparison
Shell sort [ST]	O(n1.25)	O(1)	Insertion
Selection sort [SS]	$O(n^2)$	O(1)	Selection
Heapsort [HS]	O(n log n)	O(1)	Selection
Quicksort [QS]	O(n log n)	O(log n)	Partitioning
Timsort [TS]	O(n log n)	O(n)	Insertion & Merging

mention that the response times may vary depending on the CPU characteristics, RAM, and other computer specifications on which are run the algorithms. In this context, to obtain concrete results, a different number of data is required to execute the sorting. It also executes the process repeatedly to verify the integrity of the results. Consequently, the existence of different options of sorting algorithms leads us to ask: What is the algorithm that gives us better execution times? Does the programming language have any impact on the performance of the sorting algorithms? Furthermore, how to verify the integrity of said results?

Due to the above reasons, it is necessary to carry out a complete experiment that indicates which of the sorting algorithms is the one with the best execution times. Thus, the research objective was to compile the various existing sorting algorithms in the C and Python programming languages to evaluate them. The research consists of three main steps. First, the algorithms' implementation helped create four types of different integer arrangements (random data, nearly sorted data, reverse sorted data, random data sorted by categories), which forms standard datasets. The second step is to implement the eleven sorting algorithms and sort the standard datasets, recording each algorithm's times using a different number of elements and iterations to check the results' integrity.

Moreover, the final step is the analysis of the obtained results. We will describe the methodology of the experiment in more detail; what were the steps to follow? We will explain the results in each stage, the tools used, and the results obtained.

2 Materials and Methods

Runtimes may vary depending on the characteristics of the CPU, RAM, and other specifications of the computer running at the time. Therefore, it is necessary to indicate the specifications of the equipment used. We used a 64-bit computer with an Intel i5 processor, 8 GB RAM, and a Windows 10 operating system in this research. To start the investigation, the codes of the algorithms that generate the integer arrays had to be studied, considering that the ordering algorithms were going to be used to sort in ascending order four different types of integer arrays. The algorithms to generate the collections are [14]:

- **Random** – It generates random numbers with uniform distribution.
- **Nearly Sorted** – It generates an array of numbers sorted in ascending order and then introduces some randomness; 20% of the data, in no specific position, is changed by altering them with other random data.
- **Reversed** – It generates an array of descending ordered numbers.
- **Few Unique** – It generates an array by setting the value of the categories m. In this case, there will only be five categories; then, select several random numbers gave a size N (N is the size of the array). M represents the size of the types and implements the formula M = N/m. Finally, repeat each random number (obtained in the second step) M times to complete the matrix N; there is no sort.

Once we implemented these algorithms, the arrays were stored in a text file (txt), to use the same dataset for each sorting algorithm. It is expected in this research that the algorithms generate data sets with 100, 1,000, 10,000, 100,000, 1,000,000 elements for each algorithm. Still, some of these algorithms threw errors at the moment of trying to generate integer arrays of more than 100.000 items in both Dev-C++ and Python (Table 2).

Table 2. List of files with data generated by algorithms

Algorithms for generating integer arrays	100	1,000	10,000	100,000	1,000,000
Random	✓	✓	✓	✓	X
Nearly sorted	✓	✓	✓	✓	X
Reversed	✓	✓	✓	✓	X
Few unique	✓	✓	✓	✓	✓

Discerning that creating data sets with 1,000,000 items was impossible with all algorithms, the best decision is to use a data set of up to 100,000 items.

To later obtain the classification algorithm (the references of the codes are in [15]). The algorithms code was modified in Dev-C++ and Python, so that they consume the previously generated data files and that the system has a certain number of iterations (Fig. 1).

The process is carried out with 1, 10, 100, 1,000, 10,000 iterations, recording the execution times in a spreadsheet file to be analyzed. It is essential to mention that the execution time of the algorithms is in milliseconds.

As a result of the analysis of execution times, we obtained four tables, one for each type of integer arrays generator algorithm, with the average values of each sorting

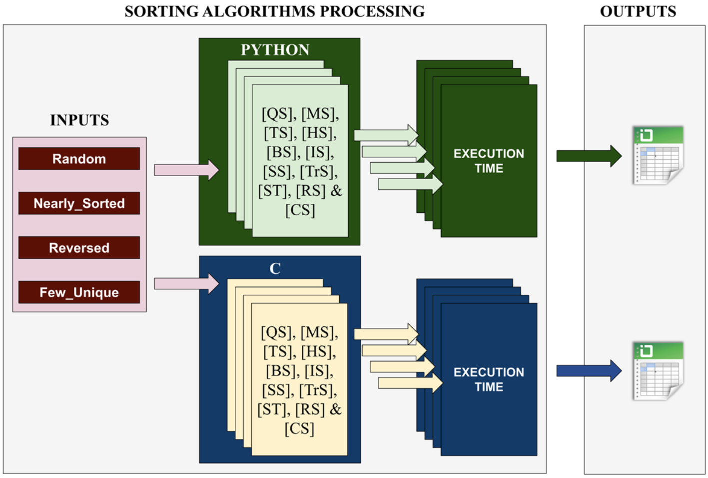

Fig. 1. Experiment process diagram

algorithm group's execution times by the number of iterations. Furthermore, with these files, graphs or figures were made using r programming. The tables and figures will show and explain in the next section.

3 Results

The eight tables with the execution times were summarized in two tables to facilitate the results' comprehension and analysis.

Appendix 1 shows the execution time averages of the random, Nearly Sorted, Few Unique, and Reversed data in C. In most algorithms, the execution and classification were satisfactory, but there were cases where there was a considerable consumption of a resource, so the program returned an error message.

Appendix 2 shows the execution time averages of the random, Nearly Sorted, Few Unique, and Reversed data in Python. An error occurred in Python due to excessive memory consumption, which did not allow the total execution of the sorting with 10.000 and 100.000 datasets.

Some algorithms have a more extensive range of execution times than others. Therefore, to organize it more thoroughly and efficiently to understand, we divide the sorting algorithms into two categories, "efficient algorithms" with a standard range of execution times and the "inefficient algorithms" with a much more extensive range of execution times. "Efficient algorithms" are defined as those whose execution times exceed that of the other algorithms by a margin of at least 100 ms.

The "inefficient algorithms" are Bubblesort, Insertion sort, and Selection sort; the remaining algorithms are considered "efficient algorithms."

There are differences between C and Python; one of the differences is that the range of Python runtimes is much more extensive.

Considering the average execution times for sorting random data for the "efficient algorithms," Heapsort is the one with the higher execution times. On the other hand, Timsort is the sorting algorithm that presents the lowest execution times in Python; in C, Tree sort is the most efficient one.

In the case of the average execution times for sorting nearly sorted data for the "efficient algorithms," Heapsort is the one with the higher execution times in C and only up to 1.000 iterations. When the number of iterations surpasses 1.000, Radix sort becomes the worst one. In Python, Tree sort is the one with the worst execution times.

In all the cases while working with "inefficient algorithms," Bubblesort was the one that got the worst execution times.

Table 3. Most efficient & least efficient algorithms

Efficient algorithms				
Algorithms	Most efficient		Least efficient	
Programming language	C	Python	C	Python
Random	Tree sort	Timsort	Heapsort	Heapsort
Nearly Sorted	Tree sort	Timsort	Radix sort	Tree sort
Reversed	Tree sort	Timsort	Heapsort	Tree sort
Few Unique	Tree sort	Timsort	Heapsort	Heapsort
Inefficient algorithms				
Algorithms	Most efficient		Least efficient	
Programming language	C	Python	C	Python
Random	Insertion sort	Selection sort	Bubblesort	Bubblesort
Nearly Sorted	Insertion sort	Insertion sort	Bubblesort	Bubblesort
Reversed	Insertion sort	Selection sort	Bubblesort	Bubblesort
Few Unique	Insertion sort	Selection sort	Bubblesort	Bubblesort

Table 3 specifies the final results for each type of integer array according to the programming language and the sorting algorithms' efficiency.

Figure 2 shows that Tree Sort is the most efficient sorting algorithm in C; however, it has higher execution times than Timsort in Python.

It is crucial to point out that Timsort and Counting sort in C had bad execution times, and they will have been the most efficient ones, but they failed to sort more than 1000 elements. Thus, we can still say that Timsort and Counting sort are the most efficient algorithms in C when we try to type a few items.

Contrary to Fig. 2, Fig. 3 shows that both programming languages have Bubblesort as the most inefficient algorithm, and its execution times are longer in Python than in C.

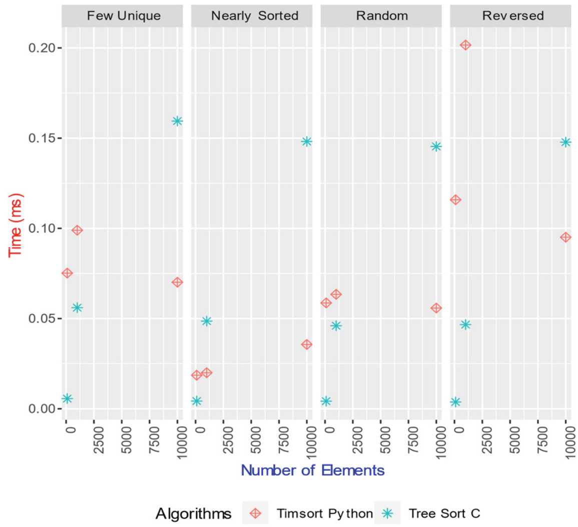

Fig. 2. Most efficient sorting algorithms

Fig. 3. Most inefficient sorting algorithms

With everything established above, we now know what algorithms give us the best execution times, no matter the number of elements and iterations.

4 Discussion

This research paper found out that the programming language significantly impacts how the sorting algorithms behave and how much data they can sort. An example of this is that Timsort is the most efficient sorting algorithm in Python, while Tree Sort is the best one in C. Both have the same complexity O(n log n). Also, Bubblesort $O(n^2)$ is the one that has the worst execution times, no matter the number of elements or iterations.

In the paper "Analysis and Review of Sorting Algorithms" [1], the authors only used five sorting algorithms. Bubblesort, Insertion Sort, Selection Sort, and Quicksort. Their research also concluded that Bubblesort is only suitable for small lists or arrays because it has the worst performance. Another paper that had similar results was "Analysis and Testing of Sorting Algorithms on a Standard Dataset" [10] in which once more, Bubblesort had the worst execution times. In their case, they worked with nine sorting algorithms programmed with C++. They also use different datasets, and the best algorithms in their case were Counting sort, which also gave us good execution times but did not run with larger arrays. Timsort and Tree Sort do not appear in the document mentioned above.

In "Experimental study on the five sort algorithms," they demonstrated that the number of items in the dataset or array has a considerable impact on the sorting algorithm's performance. Each sorting algorithm is suitable for a specific situation. If any patterns or rules are found in the input sequence, inserting and sorting bubbles is a suitable option. However, when the input scale is large, Merge Sort and Quicksort are the main choices [12].

Timsort was created in 2002 by Tim Peters [16] for use in the Python language. A hybrid classification algorithm based on the Insertion Sort and Merge Sort algorithm works are in blocks that sort using the insertion order one by one. Then the sorted blocks are merged using the merge operation used in the merge [17].

Thus, its popularity has increased, which opens the way to a series of investigations on its operation such as the investigation of "Monte Carlo simulation of polymerization reactions: optimization of the computational time" in which they analyzed the Monte Carlo simulation of a steady-state polymerization process to reduce the overall computational time, where the authors compare four ordering algorithms such as Timsort, Bubblesort, Insertion Sort, and Selection Sort, resulting in that Timsort was the most efficient algorithm in that implementation and Bubblesort the one with the worst time [18].

The authors of "Binary Tree Sort is More Robust Than Quick Sort in Average Case" [19] explained that we could use Binary tree sort if the sorted elements do not need a uniform. They proved that the robustness of Tree sort is a decisive factor instead of just focusing on the algorithm's complexity. The aforementioned makes it easier for us to explain why Trees Sort was better with larger C language arrays.

"Best sorting algorithm for nearly sorted lists" compares five algorithms, Insertion Sort, Shell sort, Merge Sort, Quicksort, and Heapsort on nearly sorted lists. Their test results showed that Insertion Sort is best for small or very nearly-sorted lists and that Quicksort is better otherwise [20]. Insertion Sort was also the best "Inefficient Algorithm" in our experiment. They concluded that there is no one sorting method that is best for every situation. For that reason, it is necessary to keep experimenting and comparing a new sorting algorithm, which is why we used more sorting algorithms.

With quantum computing, multiple implementations can be made, such as quantum treemaps used to visualize large hierarchical datasets. In an application such as using a recursive technique motivated by the Quicksort algorithm, these algorithms offer compensation, producing partially sorted designs that are reasonably stable and have relatively low aspect ratios [21].

However, the question often arises. How fast can quantum computers sort? A quantum computer only needs to compare O(0.526n log2 n) + O(n) times. Performing an improvement to the lower limit to (n log n), we obtain that the best comparison-based quantum classification algorithm can be at most a constant time faster than the best classical algorithm [22].

We encourage experimenting with those interrogations similar to those shown in this research article and implementing quantum classification methods in future work.

5 Conclusion

This research's primary purpose was to evaluate and analyze variations in execution times of sorting algorithms written in C and Python. We were interested in the resulting execution times when a sorting algorithm is run multiple times for a given dataset, and if those times differ depending on a programming language.

The experiment's orientation was for eleven classification algorithms: Quicksort, Merge-sort, Timsort, Heapsort, Bubblesort, Insertion Sort, Selection Sort, Tree Sort, Shell Sort, Radix Sort, Counting Sort. For each sorting algorithm, a range of array sizes was created and then examined.

One of the main results is that distributions of the execution times were discrete, with relatively few distinct values. Another important finding is that the execution time increased as the array size increased for all sorting algorithms. Also, organizing the data affects execution times, showing that some algorithms are better for sorting random data than inverted data, Etc. Finally, the programming language has a significant impact on how the sorting algorithms behave and how much data they can sort without the code throwing an error message. A concrete example of this is that Timsort is the most efficient sorting algorithm in Python, while Tree Sort is the best one in C. Both have the same complexity O(n log n). In both cases, Bubblesort O(n^2) is the one that has the worst execution times, no matter the number of elements or iterations.

Appendix

Appendix 1. Execution Time Averages in C

Random (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.0149	0.0102	0.0057	0.0121	0.0284	0.0074	0.0163	0.0140	0.0075	0.0037	0.0024
	10	0.0035	0.0069	0.0039	0.0072	0.0229	0.0070	0.0142	0.0036	0.0057	0.0034	0.0020
	100	0.0026	0.0061	0.0032	0.0051	0.0220	0.0072	0.0141	0.0026	0.0045	0.0034	0.0017
	1000	0.0023	0.0053	0.0027	0.0057	0.0189	0.0046	0.0136	0.0042	0.0040	0.0096	error
	10000	0.0021	0.0053	0.0022	0.0054	0.0157	0.0023	0.0133	0.0041	0.0029	0.0146	error
1000	1	0.0705	0.1129	0.2143	0.2210	2.1418	0.6657	1.2248	0.0635	0.1102	0.0471	0.0071
	10	0.0650	0.1081	0.0762	0.1509	2.1651	0.6505	1.2236	0.0401	0.1153	0.0514	0.0071

(*continued*)

(*continued*)

Random (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
	100	0.0635	0.1005	0.0712	0.1514	2.5847	0.6459	1.2322	0.0379	0.1129	0.0470	0.0072
	1000	0.0494	0.0869	0.0507	0.1325	1.6873	0.3392	1.2153	0.0460	0.0712	0.0883	error
	10000	0.0287	0.0681	0.0317	0.0816	1.2030	0.0388	1.2111	0.0470	0.0296	0.1677	error
10000	1	0.7475	1.2682	error	1.8875	298.9984	64.8266	121.1839	0.1820	1.5020	0.4770	0.0718
	10	0.7458	1.2530	error	1.9217	295.5248	66.9882	127.1354	0.1408	1.4965	0.4644	error
	100	0.7383	1.3059	error	1.9203	292.9857	65.0310	123.7623	0.1393	1.5423	0.4886	error
	1000	0.5518	1.0522	error	1.6540	205.6387	32.9524	117.5058	0.1454	0.9564	0.9526	error
	10000	0.3743	0.8348	error	1.2170	120.8390	3.3560	117.3990	0.1475	0.4116	1.3971	error
100000	1	7.5794	14.6408	error	21.4637	33941.2179	6434.9664	11729.1623	0.6891	17.9443	4.8676	0.5679
	10	8.0029	14.6789	error	22.8292	33618.3597	6464.3184	11766.5191	0.6403	17.6360	4.7118	error
	100	7.6986	14.3785	error	21.3103	33650.1493	6421.5479	11690.4792	0.6535	18.2611	4.6397	error
	1000	6.0036	12.0452	error	19.1913	22651.4197	3362.4318	11858.7491	0.6524	11.0385	0.9526	error
	10000	4.4945	9.9346	error	17.0932	12265.4345	335.2610	11444.6697	0.6552	5.0353	14.5989	error
Nearly sorted (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.0065	0.0084	0.0036	0.0591	0.0157	0.0026	0.0498	0.0152	0.0053	0.0054	0.0023
	10	0.0031	0.0072	0.0024	0.0076	0.0211	0.0021	0.0145	0.0039	0.0040	0.0142	0.0020
	100	0.0022	0.0051	0.0023	0.0055	0.0140	0.0019	0.0143	0.0028	0.0043	0.0051	0.0059
	1000	0.0023	0.0056	0.0023	0.0056	0.0148	0.0021	0.0170	0.0044	0.0032	0.0118	error
	10000	0.0022	0.0057	0.0022	0.0058	0.0150	0.0021	0.0132	0.0043	0.0030	0.0175	error
1000	1	0.0664	0.0804	0.0436	0.1232	1.4216	1.0850	5.0960	0.0648	0.1264	0.0635	0.0167
	10	0.0587	0.0746	0.0473	0.1278	1.5113	0.2016	1.2170	0.0368	0.1130	0.1570	error
	100	0.0566	0.0725	0.0385	0.1294	1.4778	0.2065	1.2323	0.0335	0.1099	0.0614	error
	1000	0.0461	0.0707	0.0343	0.1209	1.2948	0.1138	1.2140	0.0436	0.0700	0.1040	error
	10000	0.0282	0.0658	0.0303	0.0820	1.1436	0.0170	1.1859	0.0486	0.0293	0.1695	error
10000	1	0.6643	0.9435	error	1.5351	208.4337	19.6579	119.1224	0.2130	1.7079	0.6386	0.1365
	10	0.6865	0.9634	error	1.6610	193.4145	20.0551	121.1661	0.1514	1.7112	0.6426	error
	100	0.7111	0.9634	error	1.5482	190.4780	20.0197	118.8711	0.1548	1.7194	0.7056	error
	1000	0.5480	0.8773	error	1.4625	152.2325	10.1305	118.5112	0.1528	1.0949	1.1911	error
	10000	0.3759	0.8214	error	1.2002	116.1112	1.0735	120.0545	0.1482	0.4238	1.4127	error
100000	1	5.6883	11.0523	error	17.2255	14051.2450	501.8177	11832.1875	0.8658	10.1685	9.4685	1.2621
	10	5.5417	11.5409	error	17.9486	14027.0252	495.1589	11814.0204	0.6930	10.4666	9.4292	error
	100	5.6053	10.7023	error	17.3267	14028.2937	496.3286	11930.1854	0.6568	10.2494	9.2739	error
	1000	5.4688	10.2111	error	17.0733	12630.6195	255.7339	11939.0698	0.6492	8.6424	9.8717	error
	10000	4.5268	9.7984	error	16.0976	11489.6494	27.4709	12013.5563	0.6699	5.0404	17.1336	error
Few unique (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.0202	0.0102	0.0036	0.0122	0.0990	0.0082	0.0159	0.0133	0.0071	0.0053	0.0026
	10	0.0035	0.0080	0.0024	0.0067	0.0245	0.0074	0.0144	0.0042	0.0061	0.0062	0.0017
	100	0.0023	0.0057	0.0023	0.0050	0.0227	0.0073	0.0138	0.0029	0.0047	0.0051	0.0017
	1000	0.0024	0.0058	0.0023	0.0054	0.0197	0.0050	0.0138	0.0049	0.0063	0.0093	error
	10000	0.0021	0.0055	0.0023	0.0054	0.0165	0.0027	0.0135	0.0056	0.0028	0.0175	error
1000	1	0.0707	0.1079	0.0431	0.1654	2.3080	0.6995	1.8619	0.0681	0.1057	0.0459	0.0092
	10	0.0678	0.1103	0.1216	0.1584	2.2054	0.6461	1.2374	0.0417	0.1082	0.0480	error
	100	0.0655	0.1042	0.0395	0.1496	2.3190	0.6789	1.2425	0.0410	0.1055	0.0464	error
	1000	0.0473	0.0930	0.0343	0.1322	1.6716	0.3478	1.2311	0.0485	0.0677	0.0959	error

(*continued*)

(*continued*)

Few unique (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
	10000	0.0282	0.0693	0.0304	0.0809	1.1960	0.0392	1.2142	0.0560	0.0291	0.1670	error
10000	1	0.7346	1.3134	error	1.9435	303.0240	64.6072	120.0242	0.1886	1.4950	0.4800	0.0673
	10	0.7173	1.2638	error	1.8604	291.4519	65.2157	121.5685	0.1542	1.5188	0.4765	error
	100	0.7241	1.2929	error	1.8939	288.5536	64.3683	120.4502	0.3016	1.5972	0.4880	error
	1000	0.5528	1.0575	error	1.5950	202.7150	32.8495	120.1730	0.1609	0.9759	1.1033	error
	10000	0.3741	0.8454	error	1.1478	128.2706	3.3390	120.0205	0.1595	0.4123	1.3959	error
100000	1	7.5547	14.7736	error	20.7879	35097.0922	6494.1620	12027.4861	0.6967	17.5160	4.9567	0.6649
	10	7.6227	15.0838	error	22.5534	34684.1799	6477.7336	12157.8741	0.7044	17.8177	4.7438	error
	100	7.6268	14.5629	error	20.9330	33167.3059	6477.3688	12295.0094	0.6874	17.2473	4.6473	error
	1000	6.0191	12.1864	error	18.7028	22996.9892	3293.4073	11399.2589	0.6687	10.9321	9.6269	error
	10000	4.3826	10.4619	error	14.6550	12252.0630	328.5804	11635.6127	0.6777	5.0300	8.0059	error

Reversed (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.0027	0.0065	0.0063	0.0099	0.0284	0.0143	0.0135	0.0331	0.0037	0.0053	0.0013
	10	0.0017	0.0053	0.0068	0.0197	0.0276	0.0140	0.0130	0.0041	0.0075	0.0050	error
	100	0.0078	0.0050	0.0053	0.0187	0.0300	0.0137	0.0140	0.0024	0.0029	0.0052	error
	1000	0.0022	0.0051	0.0039	0.0055	0.0226	0.0083	0.0132	0.0041	0.0031	error	error
	10000	0.0019	0.0052	0.0023	0.0055	0.0159	0.0030	0.0130	0.0038	0.0029	error	error
1000	1	0.0226	0.1109	0.0647	0.2971	2.7470	1.3273	1.1790	0.0615	0.0480	0.0618	0.0127
	10	0.0237	0.0673	0.0667	0.5297	2.8306	1.3362	1.1401	0.0409	0.0568	0.0625	error
	100	0.0611	0.0655	0.0657	0.2134	2.8199	1.3104	1.1621	0.0382	0.0472	0.0630	error
	1000	0.0277	0.0665	0.0481	0.1321	1.9682	0.6799	1.1688	0.0458	0.0379	error	error
	10000	0.0267	0.0656	0.0314	0.0832	1.2151	0.0732	1.1825	0.0467	0.0259	error	error
10000	1	0.3014	0.8716	error	1.5572	276.1865	130.9751	112.9974	0.2141	2.7750	0.7980	0.0938
	10	0.3137	0.8425	error	1.4730	272.3418	132.0642	113.7755	0.1464	0.0266	0.7612	error
	100	0.3342	0.9739	error	1.4701	271.2186	130.0438	112.7412	0.1429	0.0322	0.8062	error
	1000	0.3610	0.8166	error	1.3609	193.0997	66.9404	114.9121	0.1435	0.0781	error	error
	10000	0.3577	0.8106	error	1.1514	120.3492	6.6871	116.7057	0.1478	0.3679	error	error
100000	1	4.3624	10.2862	error	17.1909	27093.0421	13115.8318	11257.0213	0.6813	8.5972	9.3526	0.9130
	10	4.1092	10.0653	error	17.2984	27077.2859	13081.0810	11274.6340	0.7093	8.4274	9.2778	error
	100	4.0350	9.7103	error	17.3921	27186.5398	14016.1420	11416.3516	0.6549	8.4440	9.1002	error
	1000	4.7252	9.6941	error	17.0638	19120.7744	6722.9272	11126.5332	0.6694	8.1070	error	error
	10000	4.4514	9.5313	error	14.1699	11716.5482	701.1256	11351.2102	0.6577	5.0600	error	error

Appendix 2. Average Execution Times in Python

Random (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.3253	0.3447	0.0591	0.4410	1.1796	0.5673	0.5574	0.3174	0.1840	0.2072	0.1802
	10	0.3017	0.3551	0.0669	0.4495	1.2176	0.6299	0.5537	0.3325	0.1903	0.1953	0.1933
	100	0.3021	0.3713	0.0619	0.4433	1.2246	0.5906	0.5450	0.3358	0.1988	0.1963	0.1975
	1000	0.2894	0.3552	0.0587	0.4239	1.1696	0.5829	0.5387	0.3265	0.1901	0.1884	0.1887
	10000	0.2854	0.3510	0.0574	0.4171	1.1606	0.5755	0.5348	0.3225	0.1871	0.1862	0.1857
1000	1	6.8990	5.7842	3.2887	27.7787	126.4296	60.8604	54.0734	4.4141	3.6280	3.6659	3.5635

(*continued*)

(*continued*)

Random (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
	10	5.2033	4.8432	0.0599	6.8569	117.3365	67.3682	57.1244	4.5659	3.6835	3.7556	3.8659
	100	5.1562	4.9112	0.0632	6.7423	118.1921	62.6277	55.6794	4.6610	3.7779	3.8228	3.8511
	1000	5.2211	5.0086	0.0634	6.8367	118.8780	62.8142	55.6440	4.6790	3.8310	3.8301	3.8293
	10000	5.1168	4.8983	0.0618	6.6996	116.7939	61.8774	54.6691	4.5610	3.7491	3.7441	3.7489
10000	1	65.7770	64.1239	0.0826	92.0450	12220.2270	6697.4468	5814.6917	66.2586	61.9572	61.7803	59.2173
	10	65.4447	63.9647	0.0557	91.5492	11902.5500	6492.1609	5677.3800	66.9012	59.5979	59.4189	59.5188
	100	65.5597	63.9213	0.0570	91.6201	11895.0791	6488.8347	5668.4324	66.8955	59.7671	59.8399	59.7862
	1000	64.8800	63.1248	0.0558	90.4387	11782.7659	6437.5941	5604.8044	66.4650	59.1533	59.1234	59.2495
Nearly sorted (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	1.1844	1.0452	0.0676	1.3179	1.6930	0.2420	1.3455	2.1746	0.3287	0.3209	0.3215
	10	0.7243	0.5087	0.0345	0.6955	0.9980	0.1390	0.7778	1.2902	0.2029	0.2069	0.2066
	100	0.4552	0.3148	0.0179	0.4435	0.7052	0.1023	0.5591	0.9060	0.1393	0.1369	0.1355
	1000	0.4654	0.3222	0.0186	0.4527	0.7142	0.1019	0.5728	0.9260	0.1412	0.1382	0.1362
	10000	0.4716	0.3275	0.0189	0.4556	0.7184	0.1029	0.5726	0.9268	0.1412	0.1406	0.1389
1000	1	10.8394	4.6520	0.0205	7.0100	69.8064	6.8512	56.4958	10.9039	2.8341	2.8221	2.7761
	10	10.9654	4.7921	0.0189	7.1691	70.2443	6.8958	56.7438	11.1606	2.8516	2.8236	2.8254
	100	11.4077	4.9532	0.0211	7.3079	72.9078	7.2542	58.1838	11.5669	2.9910	3.5081	2.9799
	1000	11.1879	4.8518	0.0200	7.3080	72.8122	7.0869	57.4603	11.3708	3.0061	3.0101	3.0027
	10000	10.9778	4.7643	0.0199	7.1918	71.3234	6.9853	56.8876	11.2388	2.9574	2.9469	2.9455
10000	1	114.5071	66.4808	0.0300	96.9319	8102.2887	1585.7202	6222.0002	167.6363	53.5893	53.5187	52.6985
	10	117.0330	68.2153	0.0593	103.4408	8083.3021	1580.7920	6064.5560	172.4053	53.5670	52.6310	53.3516
	100	120.6905	68.6117	0.0331	104.5879	8640.4002	1626.8096	5884.9889	171.6287	59.5668	58.7390	58.7437
	1000	123.7589	70.8472	0.0357	107.8657	8954.4318	1656.2141	5880.9480	174.2174	61.6672	61.6568	61.9305
Few unique (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.3721	0.3560	0.0556	0.4137	1.4227	0.5369	0.5403	0.3509	0.1649	0.1860	0.1669
	10	0.4057	0.3991	0.0680	0.4757	1.3225	0.5262	0.5530	0.3265	0.1851	0.1822	0.2084
	100	0.5676	0.6119	0.0696	0.4768	1.3199	0.8597	1.0294	0.5607	0.2395	0.2119	0.1994
	1000	0.5602	0.5160	0.0752	0.6533	1.7388	0.5955	0.7250	0.4020	0.2350	0.2094	0.2768
	10000	0.5746	0.5548	0.0908	0.6689	1.6950	0.7248	0.7463	0.4634	0.2872	0.2697	0.2813
1000	1	5.7344	5.0763	0.0744	7.5250	205.9111	84.5285	80.0704	5.6699	4.2742	13.8807	4.8759
	10	7.9406	7.4175	0.1013	13.6953	207.4326	99.4368	83.0363	6.7028	5.5434	9.5328	6.5981
	100	8.3593	8.7713	0.1213	10.8606	202.8462	99.3244	77.9542	7.5102	6.0344	5.8915	5.8995
	1000	6.9183	6.8396	0.0989	9.2002	170.5048	84.4819	65.6271	6.3029	4.9429	5.0778	5.0393
	10000	6.6683	6.6184	0.0998	9.0623	162.9227	81.4049	64.7323	6.0444	4.7744	4.7598	4.6863
10000	1	80.3642	67.9960	0.0608	9.1993	14215.9962	7218.1729	5719.4402	74.5561	59.7071	71.6940	61.3581
	10	80.9916	69.8744	0.0727	99.7600	13450.4785	6995.5644	5610.6605	77.7963	64.7814	64.0847	59.8606
	100	82.1668	68.2175	0.0671	98.8513	13436.4983	6998.2024	5619.0589	78.1411	65.0833	64.8308	64.8462
	1000	82.4485	69.0838	0.0701	99.6724	13432.9928	6984.6595	5626.1705	78.1879	64.5584	64.7239	65.0979
Reversed (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
100	1	0.9847	0.3123	0.1072	0.4099	1.7469	1.1418	0.5919	1.5278	0.1836	0.1529	0.1860
	10	1.0336	0.3278	0.1047	0.3778	1.7754	1.1482	0.5783	1.5313	0.1573	0.1600	0.1901
	100	0.9726	0.3119	0.1100	0.3828	1.6976	1.1333	0.5612	1.5143	0.1626	0.1708	0.1569
	1000	1.0845	0.3518	0.1158	0.4133	1.9028	1.2136	0.5934	1.6322	0.1764	0.2057	0.2171
	10000	1.3242	0.4352	0.1520	0.5311	2.2667	1.3632	0.7002	1.9623	0.2422	0.2267	0.2269
1000	1	75.7457	5.6057	0.1055	11.1159	270.5590	235.2450	106.6613	163.7858	3.6045	3.6041	4.3029

(*continued*)

(*continued*)

Reversed (ms)												
Data	Iter	QS	MS	TS	HS	BS	IS	SS	TrS	ST	RS	CS
	10	94.1954	7.9541	0.1553	10.7789	280.5937	179.2550	75.0042	154.2006	4.5140	4.4940	5.0139
	100	95.9501	7.1001	0.1571	10.0475	297.5478	198.9861	85.0806	160.2506	5.7619	5.0435	5.7760
	1000	89.6183	6.3754	0.2016	9.4727	266.8225	175.1879	74.5492	141.6040	5.0087	4.9125	4.8975
	10000	70.1469	4.8713	0.1340	7.3407	209.4537	138.6884	60.2487	111.9173	3.7716	3.7280	3.7273
10000	1	986.7406	56.0200	0.0809	99.4892	20500.1853	14292.7860	5826.8927	1665.4201	39.2121	43.4552	56.8798
	10	1015.5719	56.0925	0.0862	91.3458	19832.2244	13878.7156	5695.0407	1675.0777	47.4680	48.8717	45.0589
	100	1013.4446	57.3001	0.0939	94.1872	19825.9403	13865.9834	5687.1142	1676.9498	46.1496	47.3867	47.0581
	1000	1016.1134	58.0542	0.0951	94.4595	19894.5795	13881.4600	5685.3320	1680.7228	47.5433	47.6109	47.5726

References

1. Kocher, G., Agrawal, N.: Analysis and review of sorting algorithms. Int. J. Sci. Eng. Res. **2**(3), 81–84 (2014)
2. Mittermair, D., Puschner, P.: Which sorting algorithms to choose for hard real-time applications. In: Proceedings of the Ninth Euromicro Workshop on Real Time Systems, Toledo (2002)
3. Astrachan, O.: Bubble sort: an archaeological algorithmic analysis. SIGCSE Bull. (2003). (Association for Computing Machinery, Special Interest Group on Computer Science Education)
4. Khamitkar, S., Bhalchandra, P., Lokhande, S., Deshmukh, N.: The folklore of sorting algorithms. Int. J. Comput. Sci. Issues (IJCSI) **4**(2), 25–30 (2009)
5. Downey, R.G., Fellows, M.R.: Parameterized Complexity, vol. 3. Springer, Heidelberg (1999)
6. Salaria, R.S.: Data Structures & Algorithms Using C. Khanna Book Publishing Co.(p) Ltd. (2004)
7. Knuth, D.E.: The Art of Computer Programming. Volume 4A: Combinatorial Algorithms, Part 1. Pearson Education India (2011)
8. Pahuja, S.: A Practical Approach to Data Structures and Algorithms. New Age International (2007)
9. Bansal, V.K., Srivastava, R., Pooja: Indexed array algorithm for sorting. In: 2009 International Conference on Advances in Computing, Control, and Telecommunication Technologies, Trivandrum, Kerala, pp. 34–36 (2009)
10. Faujdar, N., Ghrera, S.P.: Analysis and testing of sorting algorithms on a standard dataset. In: 2015 Fifth International Conference on Communication Systems and Network Technologies, Gwalior, pp. 962–967 (2015)
11. Edjlal, R., Edjlal, A., Moradi, T.: A sort implementation comparing with bubble sort and selection sort. In: 2011 3rd International Conference on Computer Research and Development, Shanghai, pp. 380–381 (2011)
12. Yang, Y., Yu, P., Gan, Y.: Experimental study on the five sort algorithms. In: 2011 Second International Conference on Mechanic Automation and Control Engineering, Hohhot, pp. 1314–1317 (2011)
13. McMaster, K., Sambasivam, S., Rague, B., Wolthuis, S.: Distribution of execution times for sorting algorithms implemented in Java. In: Proceedings of Informing Science & IT Education Conference (InSITE), pp. 269–283 (2015)
14. GeeksforGeeks, Sorting Algorithms. https://www.geeksforgeeks.org/sorting-algorithms/. Accessed 21 Feb 2020

15. Toptal, Sorting Algorithms Animations. https://www.toptal.com/developers/sorting-algorithms. Accessed 21 Feb 2020
16. Peters, T.: Timsort description. http://svn.python.org/projects/python/trunk/Objects/listsort.txt. Accessed June 2015
17. Devi, O.R.: Int. J. Adv. Trends Comput. Sci. Eng. **4**, 15–21 (2015)
18. Amaral, A., Serpa, A., Rego, D.C., Valim, S., Soares, J.: Monte Carlo simulation of polymerization reactions: optimization of the computational time (2019)
19. Chakraborty, S., Sarkar, R.: Binary tree sort is more robust than quick sort in average case. Int. J. Comput. Sci. Eng. Appl. **2**, 115–123 (2012)
20. Cook, C.R., Kim, D.J.: Best sorting algorithm for nearly sorted lists. Commun. ACM **23**(11), 620–624 (1980)
21. Odeh, A., Elleithy, K., Almasri, M., Alajlan, A.: Sorting N elements using quantum entanglement sets. In: 3rd International Conference on Innovative Computing Technology, INTECH 2013, no. 1, pp. 213–216 (2013)
22. Shi, Y.: Quantum lower bound for sorting, pp. 1–11. Arxiv Preprint (2000). http://arxiv.org/abs/quant-ph/0009091. Accessed 21 Feb 2020