# Exploring Execution Time Patterns in Python Sorting Algorithms: A Statistical Approach

**Sammra Habib**[1]

## Abstract

*Organizing objects in computing? Otherwise, I hope that you get lucky in that digital mess. In my plan to examine this project, therefore, I was interested in test how some Python sorting algorithms actually perform in relation to speed. We mean the classics: bubble sort (rough), selection sort, merge sort and short quicksort as all obviously. I simply wrote code to execute each of them in Python and gave them a pile of random lists (large ones, small ones, whichever you want), executing each one and recorded the runtime. All those numbers? Goldmine to figure out which algorithms do make your time valuable, and which just a waste of it. In fact, what this is doing is to have a feel of how these sorting gimmicks apply to real-world data so that the next time somebody is trembling under the choice of algorithm, we actually have the receipts. The following sections discuss the existing literature, identify the methodology, report and discuss the results, and conclude about the inferences and recommendations.*

**Keywords:** *algorithm, sorting, performance, distribution, variation, Python, Bubble sort, selection sort, Merge sort, quick sort, Efficiency*

## Introduction:

An algorithm is like a set of instructions in a wholesome procedure used in solving a given problem. Every step should be clearly specified in the form of a finite set of rules and should be proven to have a finite number of applications of rules. The operations that are called by a rule are usually one or multiple operations that should be executed" (Berman & Paul, 2005).

Sorting is the reorganization of an unordered or partially ordered collection of objects into ascending or descending order to solve real-world information organization and management problems (Zeyad, 2016). In computer science, sorting algorithms play an important role in the efficient organization and retrieval of data. Sorting has been extensively applied in practical scenarios such as industrial waste

[1] Lecturer, University of Central Punjab, Lahore. Email: sammra.habib@ucp.edu.pk.

https://doi.org/10.58932/MULL0005

management, video game development, feature dimensionality reduction, and supercomputer benchmarking (Elmenreich et al., 2009; Buhmann et al., 2011).

In practice, not all algorithms perform satisfactorily because of factors such as computational cost, complexity, and accuracy (Deepthi & Birunda, 2018; Fagbola et al., 2013). This paper aims to analyze the execution time distributions of sorting algorithms implemented in Python. The study considers four standard sorting algorithms: Bubble Sort, Selection Sort, Merge Sort, and Quick Sort. The execution times of these algorithms are evaluated using different input sizes by implementing them in Python and testing them on real datasets. Through the analysis of execution-time distributions, this study provides insights into the behavior and efficiency of each algorithm. The findings contribute to the understanding of algorithm performance and provide useful guidance for selecting efficient sorting algorithms in Python applications. The subsequent sections review the existing literature, describe the research methodology, discuss the experimental findings, and present conclusions and future research directions.

**Related Work:**

Khairullah (2013) evaluated several sorting algorithms, including Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Merge Sort, using datasets containing up to 100,000 elements. The algorithms were implemented in Microsoft Visual C++ and tested on five different computers. The study concluded that the modified Heap Sort consistently outperformed the other algorithms, while Merge Sort demonstrated the slowest execution time.

Aremu et al. (2013) compared the time and space efficiency of Median Sort, Quick Sort, and Heap Sort using the C programming language with datasets containing up to 200,000 elements. Their results showed that Heap Sort achieved lower execution time and memory consumption than the other sorting algorithms.

Anwar (2016) implemented Merge Sort, Selection Sort, Quick Sort, Insertion Sort, and Bubble Sort in C++ and evaluated their execution times using datasets ranging from 100 to 30,000 elements. The study highlighted how input size significantly affects the performance of different sorting algorithms.

**Table 1:** *Experiment.Results*

| Algorithm | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Bubble Sort | O(n) | O(n^2) | O(n^2) |
| Insertion Sort | O(n) | O(n^2) | O(n^2) |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) |
| Quick Sort | O(n^2) | O(n^2) | O(n.log.(n)) |
| Merge Sort | O(n.log.(n)) | O(n.log.(n)) | O(n.log.(n)) |

## Literature Review

Sorting algorithms are fundamental to data organization and retrieval. Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Quick Sort are among the most widely used sorting techniques. Their performance is generally evaluated using three criteria: time complexity, space complexity, and stability. Since searching and many other computational operations depend on sorted data, selecting an efficient sorting algorithm is essential. Different sorting methods rearrange data differently, leading to variations in execution time and computational complexity depending on dataset size and characteristics (Buhmann et al., 2011). Merge Sort and Quick Sort are generally preferred because of their efficient average-case time complexity. Python has become one of the most popular programming languages for implementing sorting algorithms due to its simple syntax and rich library support. Previous studies have measured the runtime performance of sorting algorithms implemented in Python and found that Quick Sort performs efficiently on large datasets. Other research has investigated runtime behavior in distributed computing environments to improve load balancing and reduce energy consumption (Deepthi & Birunda, 2018). Runtime distributions provide a more comprehensive understanding of algorithm efficiency because they reveal variations, outliers, and performance trends that cannot be observed through average runtime analysis alone. However, limited research has focused specifically on runtime distributions of Python sorting algorithms. Therefore, this study addresses this gap by providing a detailed analysis of the runtime distributions of commonly used Python sorting algorithms. The findings are expected to assist researchers and software developers in selecting and optimizing sorting algorithms for practical applications.

## Methodology

### Sorting algorithms selection:

The research paper will go through a variety of sorting algorithms. These algorithms will consist of bubble sort, selection sort, merge sort and quicksort. The list of algorithms has a wide range of collection and provides a full picture of the efficiency of algorithms. The flow charts of both of the algorithms are as follows.

**Figure 1:**

*Bubble Sorting Flow*

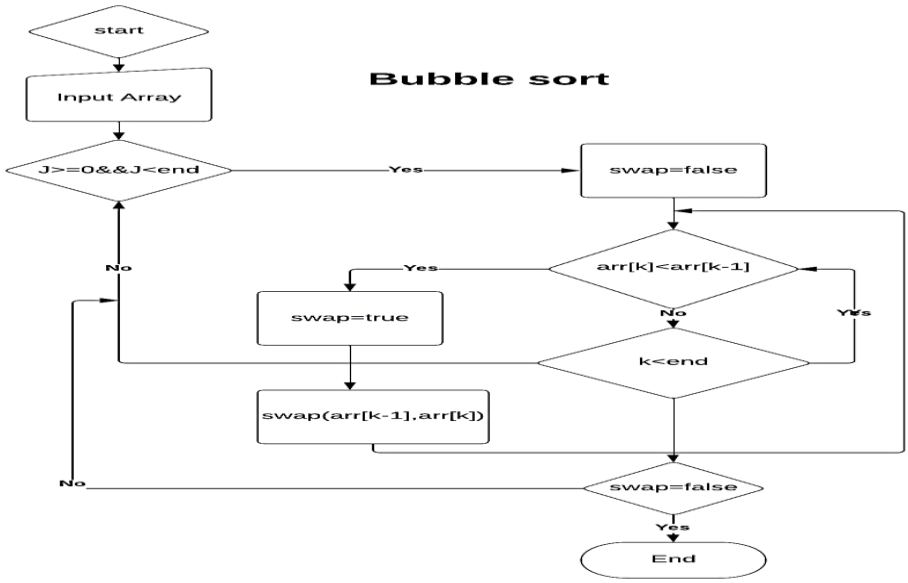



**Figure 2:**

*Selection Sorting Flow Chart*

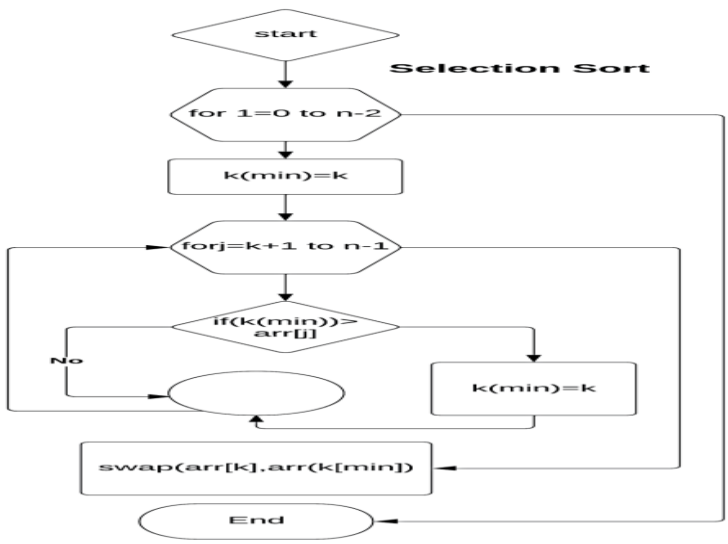


**Figure 3:**

*Quick Sorting Flow Chart*

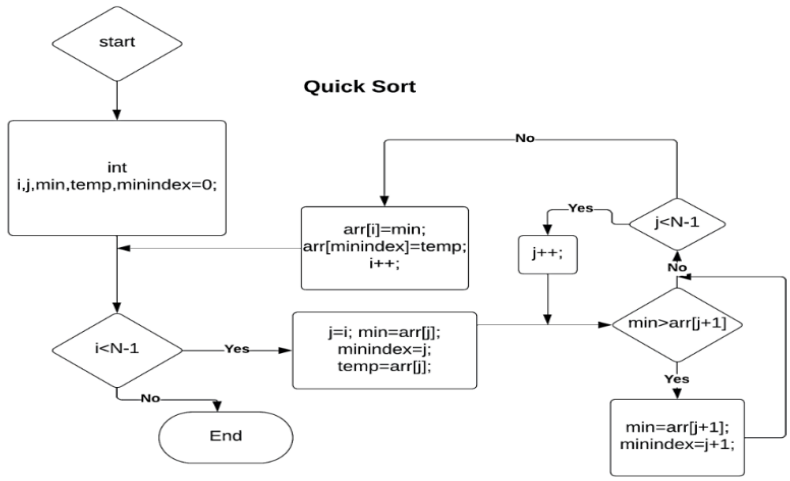
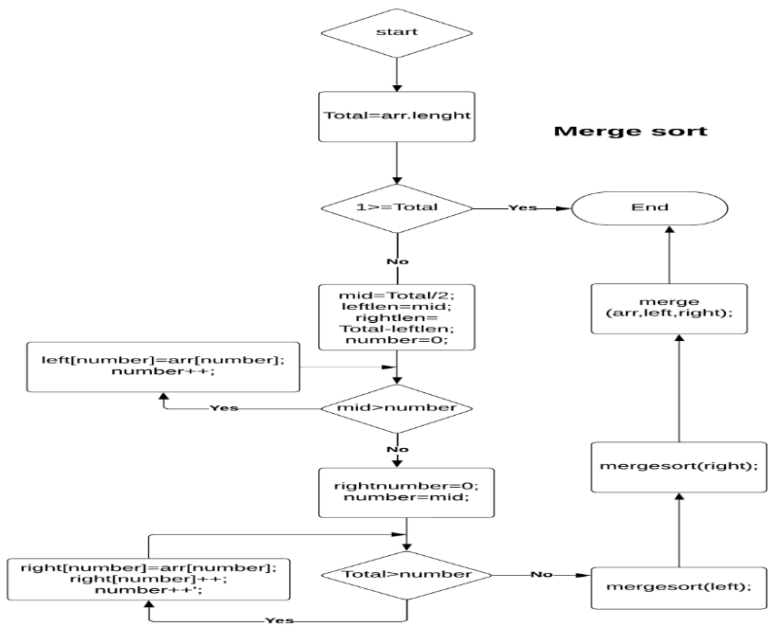



**Figure 4:**

*Merge Sorting Flow Chart*



**Implementation in Python:**

All the sorting algorithms shall be coded up in python programing language. Python was taken due to its simplicity, readability, and available data structures and plenty of modules. The implementations will follow the typical pattern of algorithms, and will be performance optimized

**Experimental Setup:**

The study will be conducted through computer system with the required hardware and software. The components of the experiment setup will be the CPU, RAM and operating systems. The version of Python that was used on the implementation will also be stated.

**Table 2**

*The Experimental Setup*

| | |
|---|---|
| Processor | Intel(R) Core (TM) i5-4460 CPU @ 3.20GHz 3.20 GHz |
| Ram | 8.00 GB |
| System Type | 64-bit operating system, x64-based processor |
| Windows | Windows 11 Pro 22H2 |
| Python | v3.11.4 |
| VS Code | v1.79.2 |

**Dataset Selection:**

The performance of the sorting algorithms will be determined by inputting random arrays in Python with the use of the library functions built-in. The datasets will consist of random numbers and will be of different sizes to test the effectiveness of the algorithm with a great variety of sizes. The peculiarities of the datasets (their size, range of values, distribution) will be mentioned.

**Table 3**

*Dataset Selection*

| **int arrays** | **sizes** |
|---|---|
| arr1 | 10000 |
| arr2 | 100000 |
| arr3 | 500000 |
| arr4 | 1000000 |

**Measurement of Execution Times:**

The execution time of a sorting algorithm is commonly influenced by several factors, including the algorithm type, the randomness of the input data, the dataset size, and the characteristics of the data being sorted (Khairullah, 2013).

The timing of testing of the sorting algorithms will be possible facilitated by appropriate implementation of timing procedures. The average time of each algorithm to sort the same dataset will be registered. The sort time of each sort algorithm used to sort the same data set will be measured on several attempts to check the reliability and accuracy of the system. The average amount of time of execution will be computed among the trials. Timing It will be reported as a number of seconds, dependent on the precision of the timing mechanism.

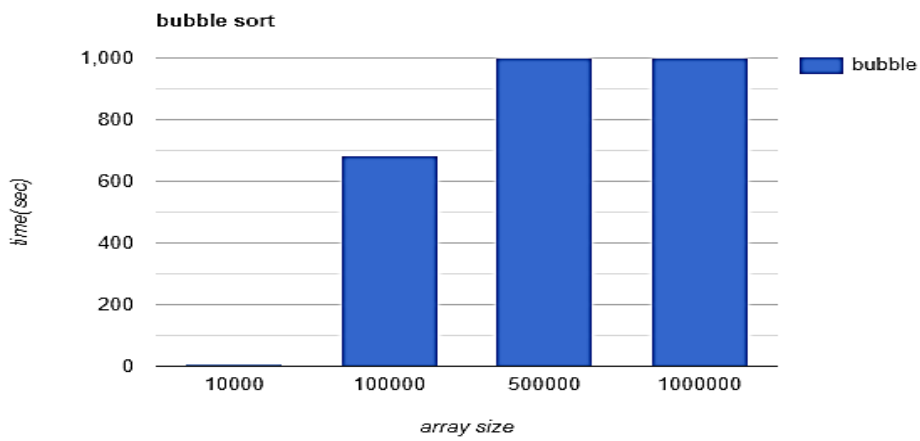
**Results and Discussion:**

**Bubble Sort Execution Time Distributions:**

The analysis of the execution time distributions of the bubble sort algorithm, concerning the various data sizes, was done. The findings indicated that the time to execute longer datasets grew extremely rapidly. This is even to be desired, given the fact bubble sort cannot be very efficient when dealing with large data sets due to its time complexity of $O(n^2)$.

**Figure 5:**

*The Examination Of Execution Time Distributions For The Bubble Sort Algorithm*



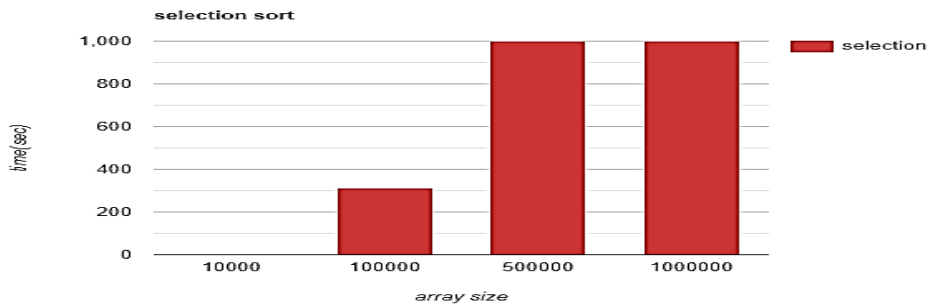

**Execution Time Distributions of Selection Sort:**

The distributions of the selection sort algorithm execution time were evaluated when the datasets were of varying sizes. Donation, like bubble sort, the

selection sort had time scales which rose with the size of the datasets. The execution time of selection sort was however quicker than the execution time of the bubble sort.

**Figure 6:**

***T**he Examination Of Execution Time Distributions For The Bubble Sort Algorithm*

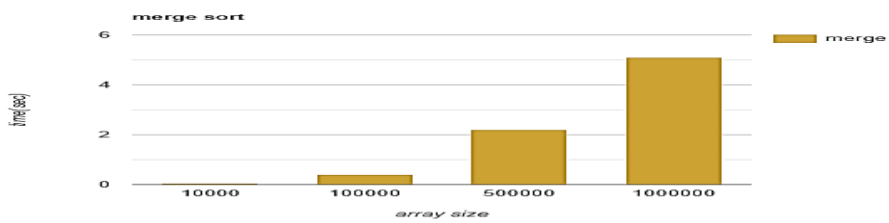



**Execution Time Distributions of Merge Sort:**

The execution time distributions of the method were tested regarding the size of datasets. As expected, merge sort was more efficient compared to bubble and selection sorts. A more regular distribution of the execution time was associated with a performance that was both efficient and uniformly comparable across various sizes of datasets.

**Figure 7:**

*Execution Time Distributions of Merge Sort*

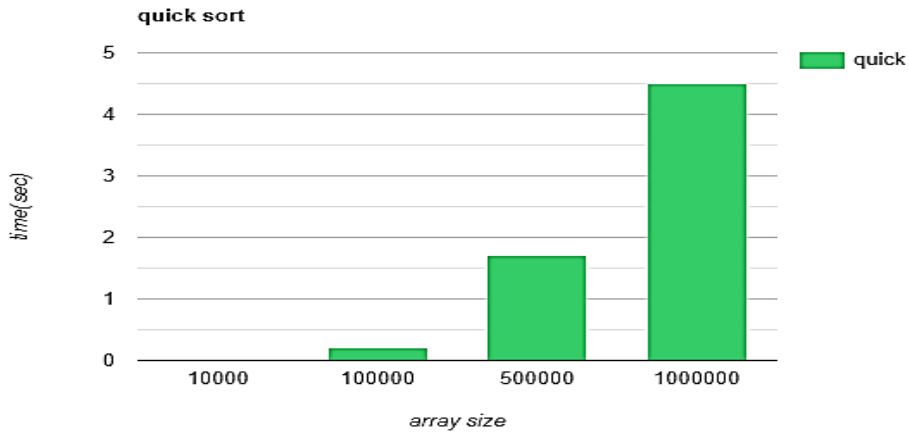



**Execution Time Distributions of Quicksort:**

The distributions of the execution time of the quicksort algorithm were evaluated to various sizes of datasets. Quicksort did remarkably well and it is rightly so because it is said to have an efficient time complexity of O(n log n) on average cases. There also seemed to be a slight bias on the distribution of execution times being more on low execution times. This meant that quicksort more often than not did efficient sorting even in large datasets.

**Figure 8:**

*Execution Time Distributions of Quicksort:*




**Comparing Execution Time Distributions:**

The implementation performance of the four sorting algorithms was provided by a comparison of the distribution of the execution time distributions of the techniques. Worse performance of bubble sort and selection sort over large dataset sets as a property of longer execution times and eventual imbalance of the distributions were demonstrated. The appropriateness of the quicksort as well as the merge sort to sort larger sets of data was also demonstrated by their superior result and the distribution balance.

**Figure 9:**

*Comparing Execution Time Distributions:*

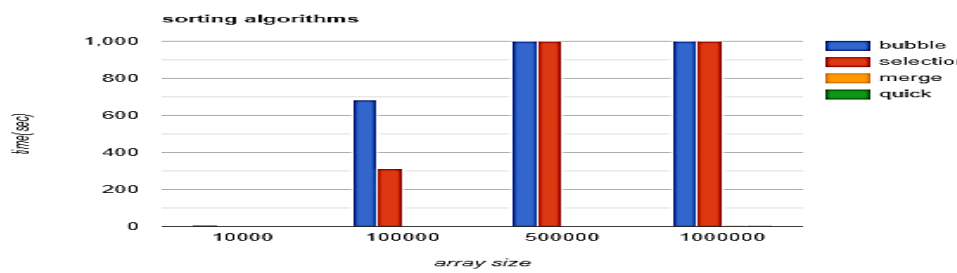


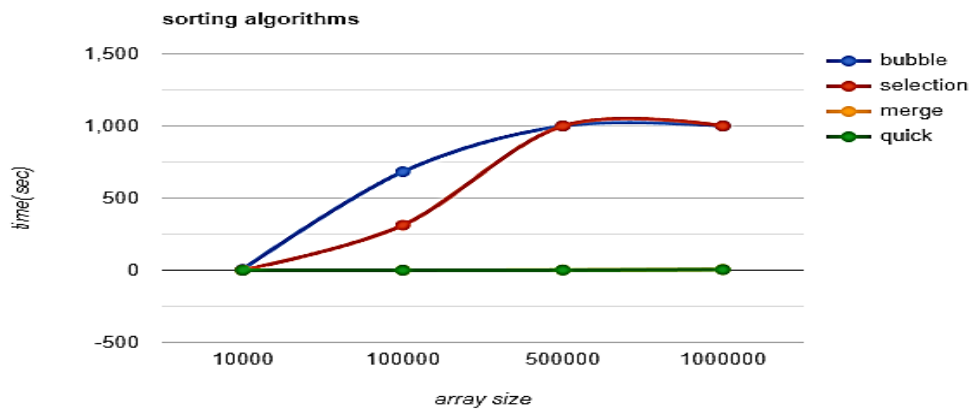
**Table 4**

*Integer Arrays Of Sizes*

| | Integers, Arrays of Sizes | | | |
|---|---|---|---|---|
| | 10000 | 100000 | 500000 | 1000000 |
| Bubble | 6.1s | 683.1s | no result | no result |
| Selection | 2.5s | 312.4s | no result | no result |
| Merge | 0.05s | 0.4s | 2.2s | 5.1s |
| Quick | 0.01s | 0.2s | 1.7s | 4.5s |

**Figure 10:**

*Sorting Algorithm Graph*

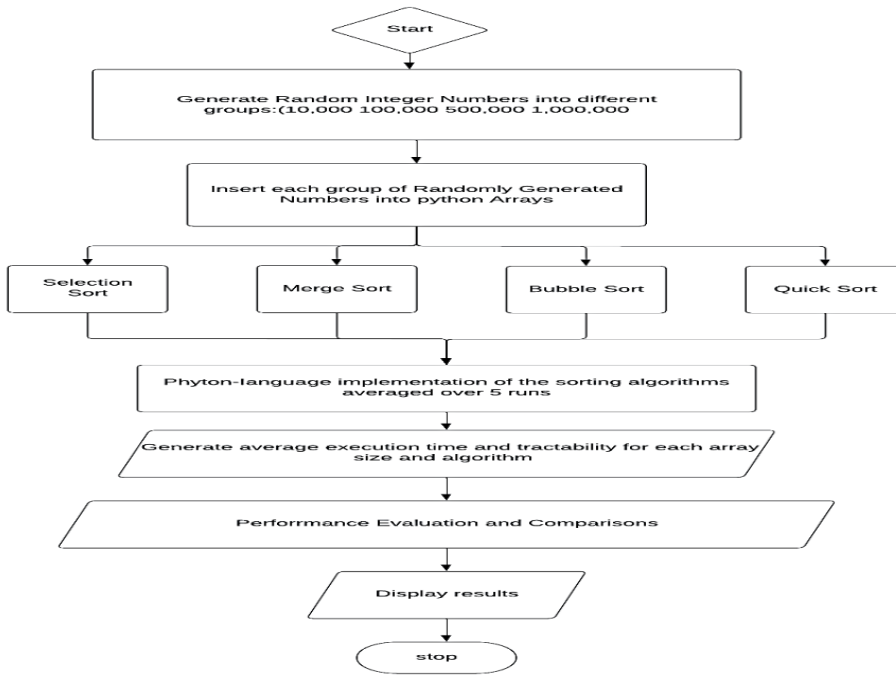



**Conclusion:**

The objective of this research study was on the distribution of the execution time of the sorting algorithms written in Python. The findings showed that the merge sort and the quicksort were equally optimal in addition to being more predictive in regard to the execution time and that both the sorting algorithms were fit in the processing of large datasets. The time consumptions in both bubble sort and the positively skewed distributions in selection sort showed the ineffectiveness in both methods with increasing data. Load balancing, the improvement of performance, and the choice of an approach in a distributed computing environment should be thought about the consequences of this study. Having an understanding of the execution time distributions of sorting algorithms, people who are at work in practice ought to be able to operationalize the execution time distributions safely with the aim of maximizing performance and reducing the cost of computing.

**Figure 11:**

*The flow control of the Experimental Evaluation of the Sorting Algorithms*




The future studies need to determine the efficacy of sorting algorithms and other sorting algorithm types in the actual world scenario, as well as test the various optimization procedures and evaluation of programming languages. According to these recommendations, the work of sorting algorithms will eventually get better and will enlarge our knowledge about the distributions of the execution time. The study at hand emphasizes demonstrates the practical data on the distributions of execution times of the sorting algorithms coded in Python. One of the implications of the analysis is to add value in practice and research in order to inform the future research and work on designing sorting algorithms.

**References:**

Aremu, D. R., Adesina, O. O., Makinde, O. E., Ajibola, O., & Agbo-Ajala, O. O. (2013). A comparative study of sorting algorithms. African Journal of Computing & ICT, 6(5), 199–206.

Berman, K. A., & Paul, J. L. (2005). Algorithms: Sequential, parallel, and distributed. Course Technology.

Buhmann, J. M., Chehreghani, M. H., Streich, A. P., & Frank, M. (2011). Information

theoretic model selection for pattern analysis. In G. Gordon, D. Dunson, & M. Dudík (Eds.), Proceedings of the ICML Workshop on Unsupervised and Transfer Learning (pp. 61–74). JMLR Workshop and Conference Proceedings, 27. https://proceedings.mlr.press/v27/buhmann12a.html

Deepthi, T., & Birunda, A. M. (2018). Time and energy efficiency: A comparative study of sorting algorithms implemented in C. In Proceedings of the International Conference on Advancements in Computing Technologies (ICACT) (Vol. 4, No. 2, pp. 25–27).

Elmenreich, W., Ibounig, A., & Fehervari, I. (2009). Robustness versus performance in sorting and tournament algorithms. Acta Polytechnica Hungarica, 6(5), 193–206.

Fagbola, T. M., Babatunde, R. S., & Oyeleye, A. (2013). Image clustering using a hybrid GA-FCM algorithm. International Journal of Engineering and Technology, 3(2), 99–107.

Khairullah, M. (2013). Enhancing worst sorting algorithms. International Journal of Advanced Science and Technology, 56, 13–26.

Vandana, V. S., Parvinder, S. S., Satwinder, S., & Baljit, S. (2008). Analysis of modified heap sort algorithm on different environment. World Academy of Science, Engineering and Technology, 2(6), 1143–1145.

Zeyad, A. A. (2016). Comparison study of sorting techniques in dynamic data structure (Master's thesis, Universiti Tun Hussein Onn Malaysia). Faculty of Computer Science and Information Technology.