Sorting algorithms: how are they affected by memory faults?

Alexandro Vladno
Centre for Informatics and Systems
University of Coimbra
Coimbra, Portugal
alexandro.vladno@gmail.com

Fabiano Papaiz
Centre for Informatics and Systems
University of Coimbra
Coimbra, Portugal
fabianopapaiz@gmail.com

Leo Moreira Silva
Centre of Informatics and Systems
University of Coimbra
Coimbra, Portugal
leo.moreira@me.com

Abstract

Although most people in the world use technology devices for many tasks, they don't know how the devices work and how they deal with faults. When those faults occur in memory, software behavior could be affected. Together with the software-specific algorithms are the sorting algorithms used to solve problems like ordering a list of products by their price. This work presents a discussion about how quicksort, mergesort, insertionsort and bubblesort algorithms are affected by memory faults.

Index Terms

sorting algorithms, memory faults.

I. Introduction

Technology is deeply introduced in people's quotidian supporting a massive number of tasks, for example: searching for a shared car, surfing on the web, sending a message to someone, automating the company's production or using the company's software. Nevertheless, most people don't know that devices are continually dealing with memory failures, faults and errors. These devices were made with large and inexpensive memories, which are also error-prone [1].

Software behavior may be affected by the problems mentioned before, especially those from memory. We have a memory fault when the correct value that should be stored in a memory location gets altered because of a soft failure. In particular, the content of a location can change unexpectedly, i.e., faults may happen at any time: real memory faults are indeed highly dynamic and unpredictable [2].

In the beginning steps of software development, the designer has a general idea of the structure and functions. For each one of these, some algorithms will be produced or used. In the following stages, the outcome software (and its algorithms) will be tested and, then, delivered to the user. Different kinds of algorithms could be written or used in the software, and one of these is the sorting algorithms.

A good algorithm is that which gives satisfactory results for every range of data set. Sorting is a fundamental concept and important for solving other problems like is prerequisite for Binary Search. Sorting is often used in a large variety of critical applications and is a fundamental task that is used by most computers [3].

In this paper, we present a discussion about how these sorting algorithms, particularly Quicksort, Mergesort, Insertion Sort and Bubblesort, are affected by memory faults.

II. Background

In this section we describe basic concepts about memory faults and sorting algorithms.

A. Memory Faults

Even the best digital system, with high-quality components and design techniques, may not be infallible to faults. Despite the title of this subsection, when the entire digital system (or software) is considered, there are three terms for computing fault and they have different meanings: failure, fault and error [4].

- *Error*: An error is a manifestation of a fault in a system, in which the logical state of an element differs from its intended value. An error occurs for a particular system state and input when an incorrect next state and/or output results.
- *Fault*: A fault is an anomalous physical condition. Causes include design errors, manufacturing problems, damage, fatigue, or other deterioration. Faults resulting from design errors and external factors are especially difficult to model and protect against because their occurrences and effects are hard to predict. A fault in a system does not necessarily result in an error;
- *Failure*: A failure denotes an element's inability to perform its functions because of error in the element itself or its environment, which in turn are caused by various faults;

B. Sorting Algorithms

Sorting algorithms are widely used in many aspects of data processing, information searches, business finance, computer encryption, etc. This work uses four sorting algorithms: quicksort, mergesort, insertionsort, and bubblesort. In the following subsections, we'll give an overview of them.

1) Quicksort: Quicksort algorithm, created by Hoare [5], is considered as one of the fastest and best sorting algorithms [6]. The algorithm is based on the paradigm of divide and conquer.

This algorithm has a execution time of $\theta(n^2)$ in the worst case over *n* numbers as input. Despite that execution time, quicksort is often the best option for sorting because of its remarkable average efficiency: $\theta(nlgn)$ [7].

The basic steps of this algorithm are [6]:

- Pick an element, which is called a pivot, from the list waiting to be sorted;
- Perform partition operation to realize that all elements in the list with values smaller than the pivot came before the pivot. Otherwise, all elements in the list with values bigger than the pivot come after it (elements which are equal to pivot can go either way). After this partition, the pivot is in the final position of the list;
- Recursively sort the sub-list of smaller elements and the sub-list of the bigger elements.

2) Mergesort: Mergesort was invented by John Von Newman and is one of the most elegant algorithms to appear in the sorting literature. It is the first sorting algorithm to have $\theta(nlgn)$ execution time bound. It is important to observe that this algorithm spends a lot of time on data transfer operations. In fact, standard Mergesort incurs about 2n data move operations [8].

Conceptually, Mergesort works as follows [8]:

- Divide the unsorted array into two sub arrays of about half the size;
- Sort each sub array recursively;
- Merge the two sub arrays back into one array.

3) Insertionsort: This algorithm sorts the array by shifting the elements one at time. It is efficient in sorting a small number of elements. The overall execution time of this algorithm is $\theta(n^2)$ [7]. The basic sorting steps are:

- If there are more than one element, pick the next element;
- Compare with all the elements in sorted sub-list;
- Shift all the elements in sorted sub-list that is greater than the value to be sorted;

- Insert the value;
- Repeat until list is sorted.

4) Bubblesort: The bubble sort is the oldest and simplest sorting method in use. It works by comparing each item in the list with the item next to it, and swapping them if required. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items (in other words, all items are in the correct order) [9].

Table I below shows the time complexity comparison between the sorting algorithms presented. The *n* is the number of input elements.

TABLE I: Sorting algorithms complexity time comparison [10]

Algorithm	**Time Complexity**		
	Best Case	***Average Case***	***Worst Case***
Bubblesort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(nlgn)$	$O(nlgn)$	$O(n^2)$
Mergesort	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$

III. Experimental Setup

We first state our problem, then describe our data, showing all its characteristics. Then, we provide our hypothesis, define the setup, and perform the testing. Finally, we collect and perform data analysis.

A. Problem Statement

As introduced in the first section of this paper, sorting is a fundamental concept and essential for solving other problems. The content of memory location can change unexpectedly, i.e., faults may happen at any time. Considering this, the main objective of this work is to design experiments to answer the following question: *How are sorting algorithms affected by memory faults?*

B. Variables

For this experimental study, we assume that the independent and dependent variables are as shown in Table II and Table III below:

TABLE II: Independent variables.

Variable	**Description**
Probability of failure	Probability of a fault to occur
Array size	Size of the array of integers to be sorted
Sorting algorithm	Algorithm used to sort the array

TABLE III: Dependent variables.

Variable	Description
Largest subarray size	Size of the largest sorted subarray produced under the memory fault
Percentage of largest subarray size	Percentage of *largest subarray size* related to *array size* independent variable
Unordered elements quantity	Quantity of elements out of position after sorting algorithm execution. Adapted of the *k-unordered sequence* measure of disorder defined in [11]
Percentage of unordered elements quantity	Percentage of *unordered elements quantity* related to *array size* independent variable

C. Hypothesis

The set of hypothesis defined to test and draw some conclusions about this experiment are listed below. The confidence degree defined for hypothesis testing was 95% ($\alpha = 0.05$ and $\alpha - 1 = 0.95$).

- **Hypothesis 1:** For a given probability of failure and array size, tested algorithms will produce a different percentage of unordered elements quantity.
- **Hypothesis 2:** For a given probability of failure and array size, tested algorithms will produce a different percentage of the largest subarray size.
- **Hypothesis 3:** For each algorithm, the array size and probability of failure have a significative impact on the percentage of unordered elements quantity.
- **Hypothesis 4:** For each algorithm, the array size and probability of failure have a significative impact on the percentage of the largest subarray size.

D. Dataset

To conduct the proposed study, we define the values of the independent variables, as shown in Table IV:

TABLE IV: Values of the independent variables.

Variable	Values
Probability of failure	1%, 2% and 5%
Array size	100, 1000 and 10000
Sorting algorithm	Bubblesort, Quicksort, Mergesort and Insertion sort

Based on these variables, we ran an existing script *gen.py* to produce input files. We define that our sample was composed by 30 input files for a given combination of the probability of failure and array size. So, considering this, we ran 30 times for each combination of these independent variables, producing 30 inputs, totalizing 270 files. Figure 1 shows an example of produced input files:

```
0.01 100 9 48 37 6 26 7 24 44 17 50 48 30 49 33 22 13 42 29 39 13 19 13 9 28
34 1 33 27 14 45 48 40 11 17 6 50 9 44 20 16 37 45 23 14 38 29 10 49 44 46 35
45 15 2 22 1 46 40 8 48 23 23 32 35 3 15 8 36 17 24 27 48 28 5 28 50 44 4 25
6 9 1 11 44 26 50 44 12 7 20 30 20 37 20 6 8 13 15 20 49
```

Fig. 1: Example of input file.

The input data shown in the Figure 1 is divided as follows:

- *Probability of Failure*: the first number of the sequence (`0.01`) is the probability of memory failure when sorting;
- *Sequence size*: the second number (`100`) means the size of the integers sequence used by sorting;
- *Sequence*: the rest of the numbers indicates the sequence itself.

With this input data, we ran, for each one of these, all four algorithms considered in this study. The sorting algorithms used already existed. For example, using all 270 input files, we ran bubblesort, creating 270 output files, and so on for the other algorithms. At the end of executions, we get a total of 1080 output files. An output file look like shown in Figure 2:

```
[1]  9 48 37 6 26 7 24 44 17 50 48 30 49 33 22 13 42 29 39 13 19 13 9 28 34 1
33 27 14 45 48 40 11 17 6 50 9 44 20 16 37 45 23 14 38 29 10 49 44 46 35 45
15 2 22 1 46 40 8 48 23 23 32 35 3 15 8 36 17 24 27 48 28 5 28 50 44 4 25 6 9
1 11 44 26 50 44 12 7 20 30 20 37 20 6 8 13 15 20 49
[2]  1 1 1 2 3 4 5 6 6 6 6 7 7 8 8 8 9 9 9 9 10 11 11 12 13 13 13 13 14 14 15
15 15 16 17 17 17 19 20 20 20 20 22 22 23 23 23 24 24 25 26 26 27 27 28 28 28
29 29 30 30 32 33 33 34 35 35 36 37 37 37 38 39 40 40 42 44 44 44 44 44 44 20
45 45 45 46 46 48 48 48 48 48 49 49 50 49 50 50 50
[3]  1 1 1 2 3 4 5 6 6 6 6 7 7 8 8 8 9 9 9 9 10 11 11 12 13 13 13 13 14 14 15
15 15 16 17 17 17 19 20 20 20 20 20 22 22 23 23 23 24 24 25 26 26 27 27 28 28
28 29 29 30 30 32 33 33 34 35 35 36 37 37 37 38 39 40 40 42 44 44 44 44 44 44
45 45 45 46 46 48 48 48 48 48 49 49 49 50 50 50 50
[4]  82
```

Fig. 2: Example of output file.

The output file gives four essential data, as enumerated below:

- *[1]*: the original sequence of integers contained in the input file;
- *[2]*: the sequence processed by the sorting algorithm under the memory fault model;
- *[3]*: the sequence sorted correctlty;
- *[4]*: the size of the largest sorted subsequence in [2]. This number can be interpreted as the quality of sorting. As higher, most successful was the sorting operation.

After generating the dataset, we developed a Python script that reads the 1080 output files and produces a single CSV file (first lines showed in Figure 3 below), containing the following columns:

- *algorithm*: the algorithm used to sort the array;
- *probability_of_failure*: the probability of failure used when sorting;
- *size_of_array*: the size of the array to be sorted;
- *largest_sorted_subarray*: the largest sorted subarray after sorting;
- *k_unordered_sequence*: number of unordered sequence after sorting.
- *percentage_k_unordered*: percentage of unordered sequence after sorting related to original array;
- *percentage_largest_sorted_subarray*: percentage of largest sorted subarray after sorting related to original array.

```
algorithm;probability_of_failure;size_of_array;largest_sorted_subarray;
k_unordered_sequence;percentage_k_unordered;
percentage_largest_sorted_subarray
quick;0.01;100;35;4;4.00;35.00
quick;0.01;100;36;8;8.00;36.00
quick;0.01;100;31;5;5.00;31.00
quick;0.01;100;20;6;6.00;20.00
quick;0.01;100;31;5;5.00;31.00
```

Fig. 3: Example of output CSV file.

We use Python libraries to make data analysis and plot graphs. These libraries were:

- *Pandas*[1]: open source library providing data structures and data analysis tools;
- *NumPy*[2]: library for scientific computing with Python;
- *SciPy*[3]: ecosystem of open-source software for mathematics, science, and engineering;
- *StatsModels*[4]: module that provides classes and functions for the estimation of many different statistical models, as well as for conducting statistical tests, and statistical data exploration;
- *Matplotlib*[5]: plotting library;
- *Seaborn*[6]: data visualization library based on matplotlib.

E. Conclusions

IV. Data Analysis

In this section, we present our results after the execution of sorting algorithms over the input files. We analyzed only the dependent variables *percentage of the largest subarray size ((%LSS)* and *percentage of unordered elements quantity (%UEQ)*. These variables, because they are a percentage value, already were normalized (i.e., the same order of magnitude) related to dependent variable array size.

A. Exploratory Data Analysis (EDA)

Firstly, we performed an analysis of the distribution of the dependent variables %LSS and %UEQ. To help in this task, we produced histograms, boxplot graphs, tables containing data about mean, median, standard deviation, and the minimum and maximum values.

The following Figures 4, 5, 6 and 7 illustrates examples of histograms and boxplot graphs for each combination of *Algorithm X Probability of Failure X Array Size*. In each of those figures, the graphs were exhibited over the dependent variables %LSS and %UEQ. In the histogram, the red vertical line means the mean, and the blue vertical line means the median. On the other hand, in the boxplot, the red horizontal line means the mean, and the blue horizontal line means the median.

[1]https://pandas.pydata.org
[2]https://numpy.org
[3]https://www.scipy.org
[4]https://www.statsmodels.org
[5]https://matplotlib.org
[6]https://seaborn.pydata.org

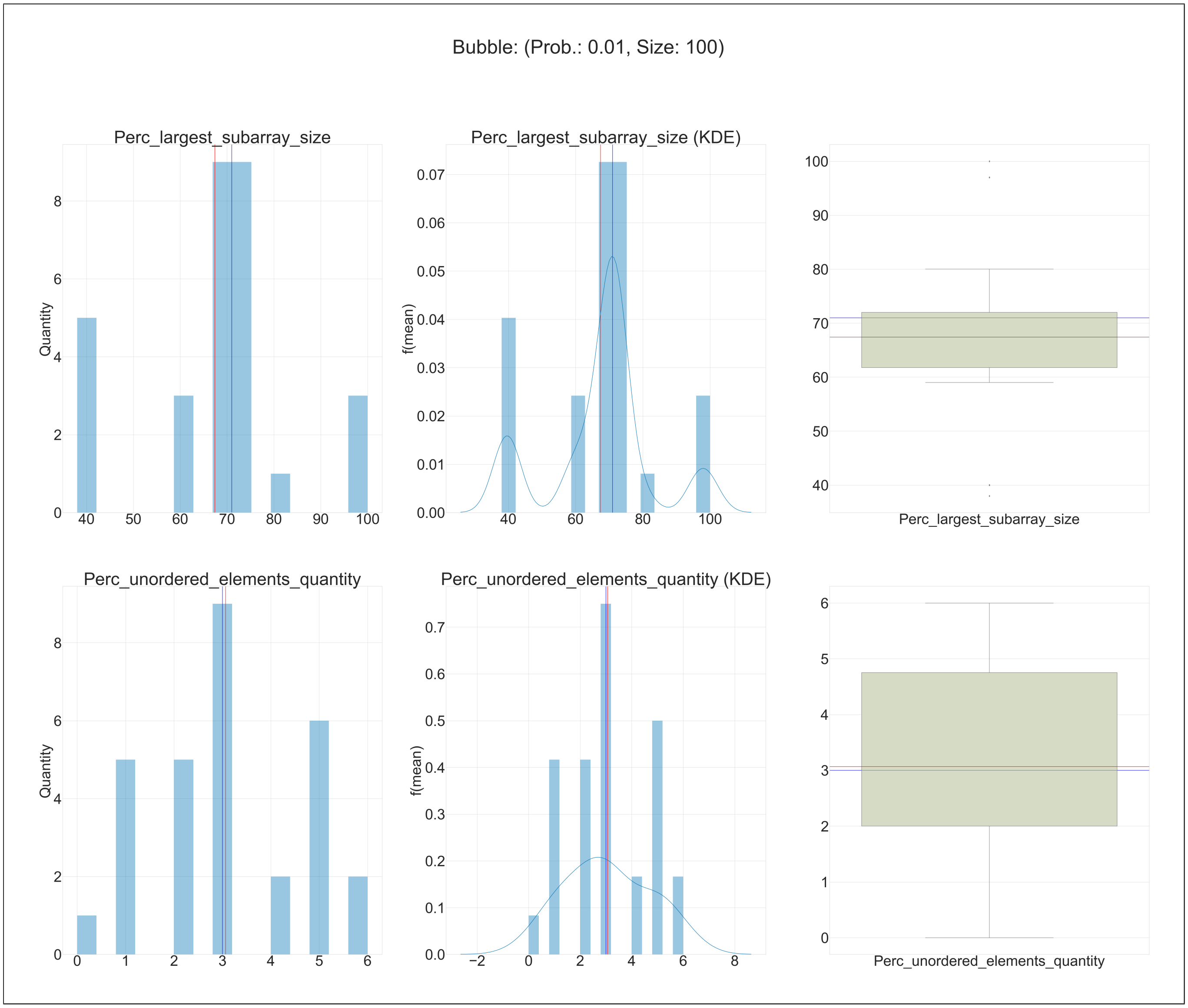

Fig. 4: Histograms and Boxplot for Bubblesort, with *probability of failure* of 0.01 and *array size* of 100.

Based on Figures 4, 5, 6 and 7, the following Tables V and VI illustrates the information about the dependent variables (%LSS and %UEQ) distribution.

TABLE V: Table Type Styles

Prob. of Failure	**Array Size**	**Algorithm**	**Percentage of unordered elements quantity (%UEQ)**				
			Mean	***Median***	***Std. Deviation***	***Minimum***	***Maximum***
0.01	100	bubble	3.07	3.0	1.64	0.0	6.0
0.01	100	insertion	10.87	11.0	1.59	8.0	14.0
0.05	10000	merge	28.32	28.34	0.31	27.69	28.86
0.05	10000	quick	19.56	19.58	0.26	18.95	20.07

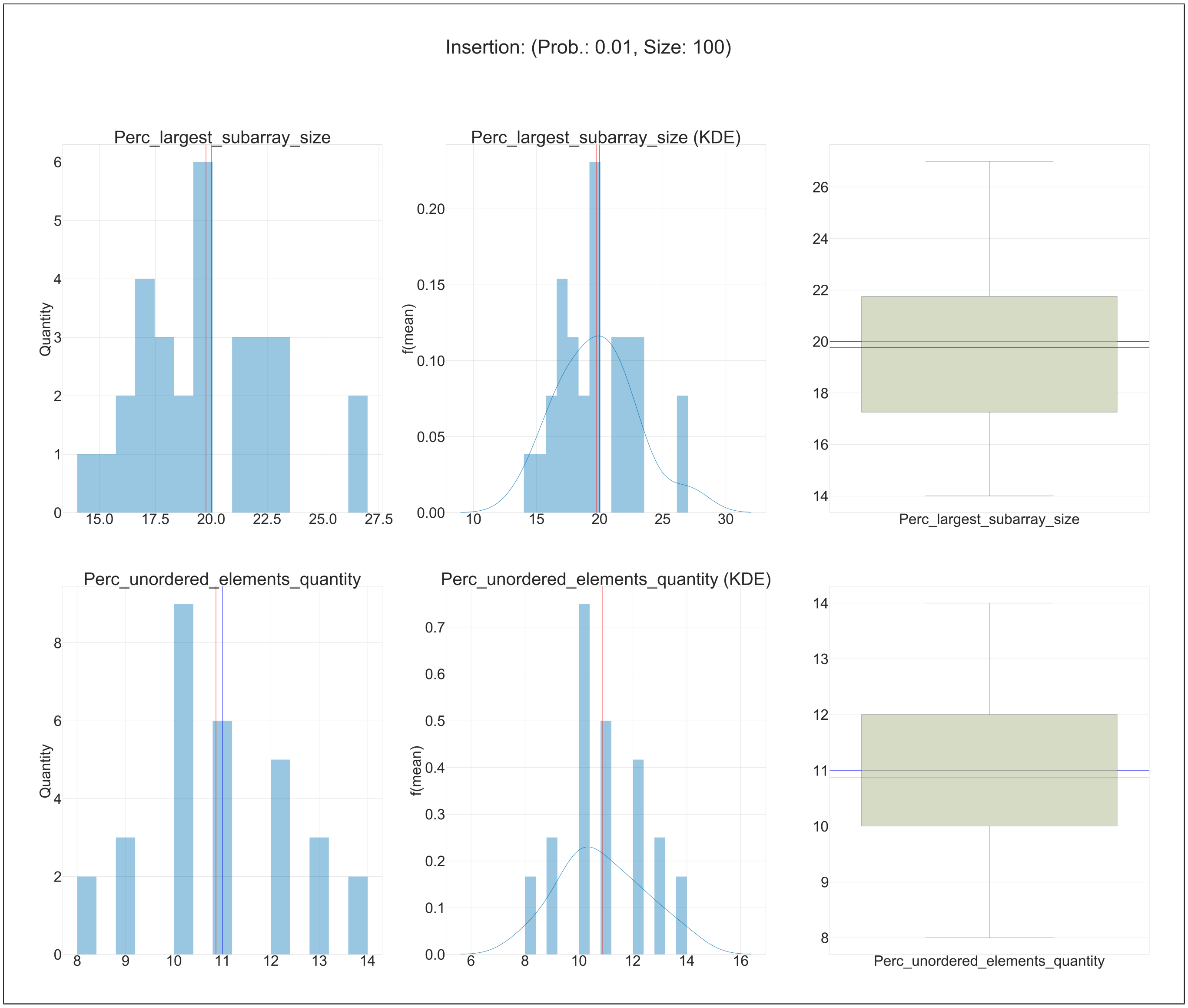

Fig. 5: Histograms and Boxplot for Insertion sort, with *probability of failure* of 0.01 and *array size* of 100.

TABLE VI: Table Type Styles

Prob. of Failure	**Array Size**	**Algorithm**	**Percentage of largest subarray size (%LSS)**				
			Mean	***Median***	***Std. Deviation***	***Minimum***	***Maximum***
0.01	100	bubble	67.4	71.0	15.84	38.0	100.0
0.01	100	insertion	19.77	20.0	3.11	14.0	27.0
0.05	10000	merge	0.19	0.2	0.03	0.15	28.86
0.05	10000	quick	0.3	0.3	0.03	0.23	0.34

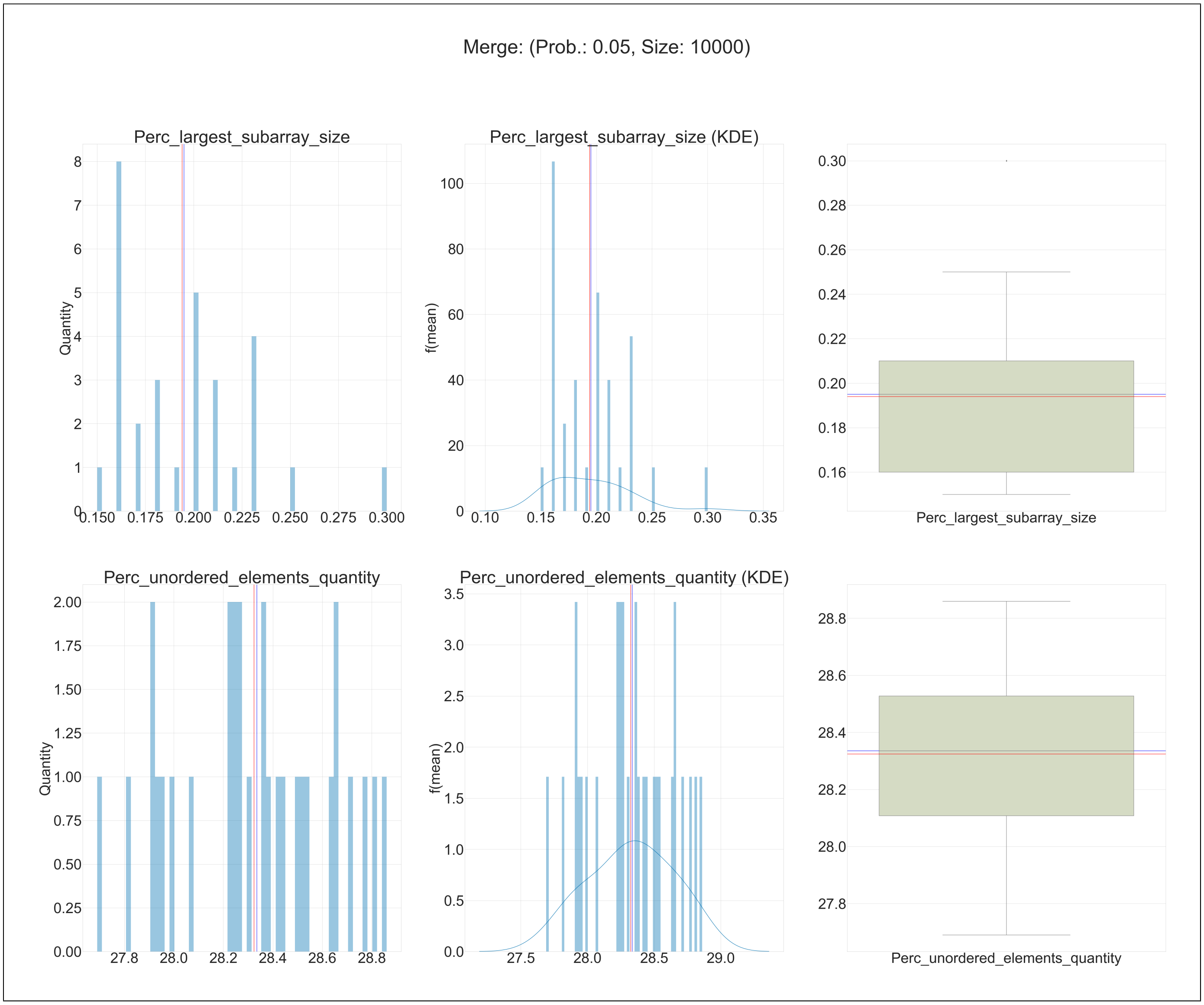

Fig. 6: Histograms and Boxplot for Mergesort, with *probability of failure* of 0.05 and *array size* of 10000.

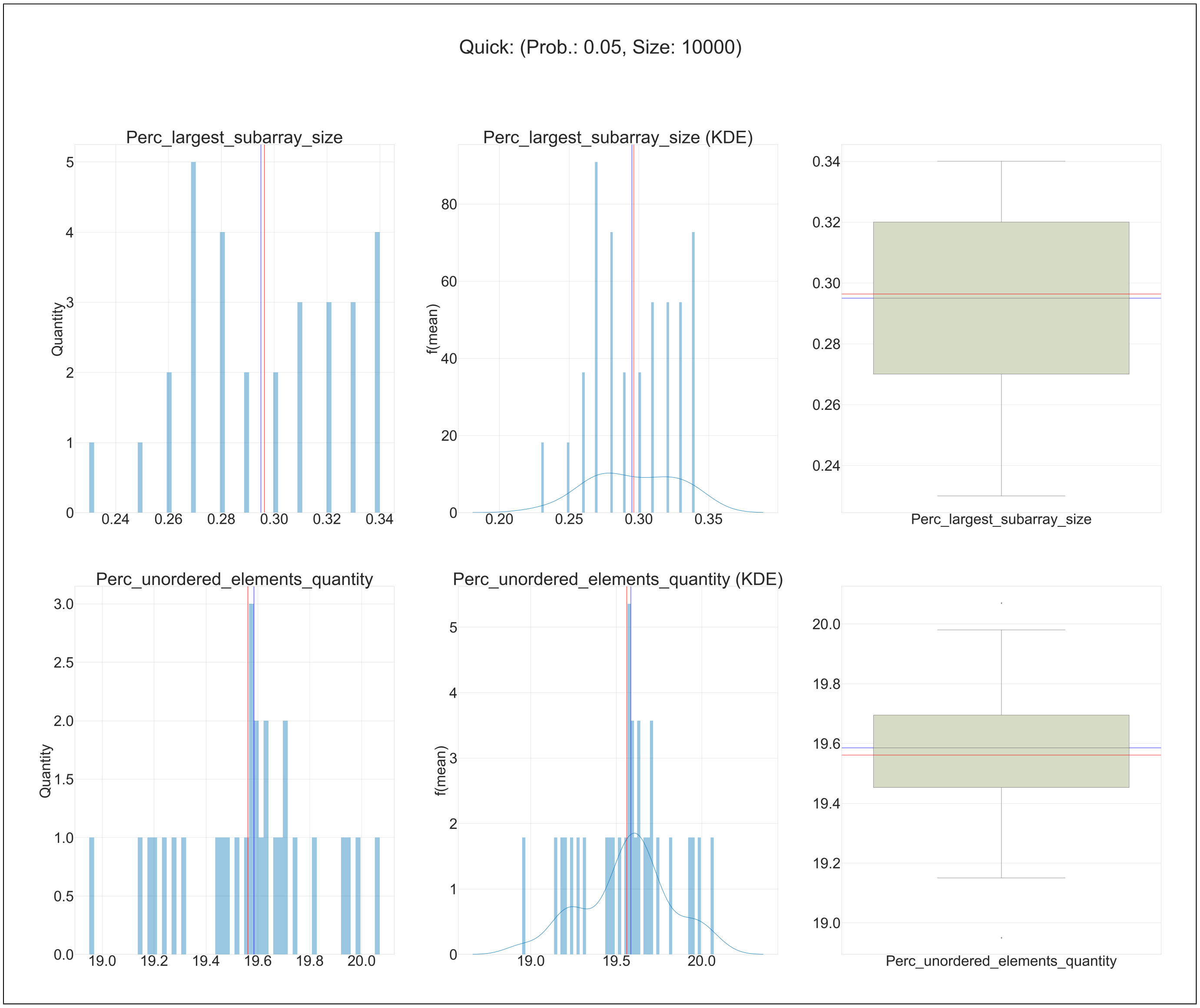

Fig. 7: Histograms and Boxplot for Quicksort, with *probability of failure* of 0.05 and *array size* of 10000.

We produced graphs with the same data showed in Tables V and VI. Figure 8 shows an example of these graphs. On it, we illustrate on the top-left graph that, considering the mean, bubblesort produces fewer unordered elements related to the total than other algorithms.

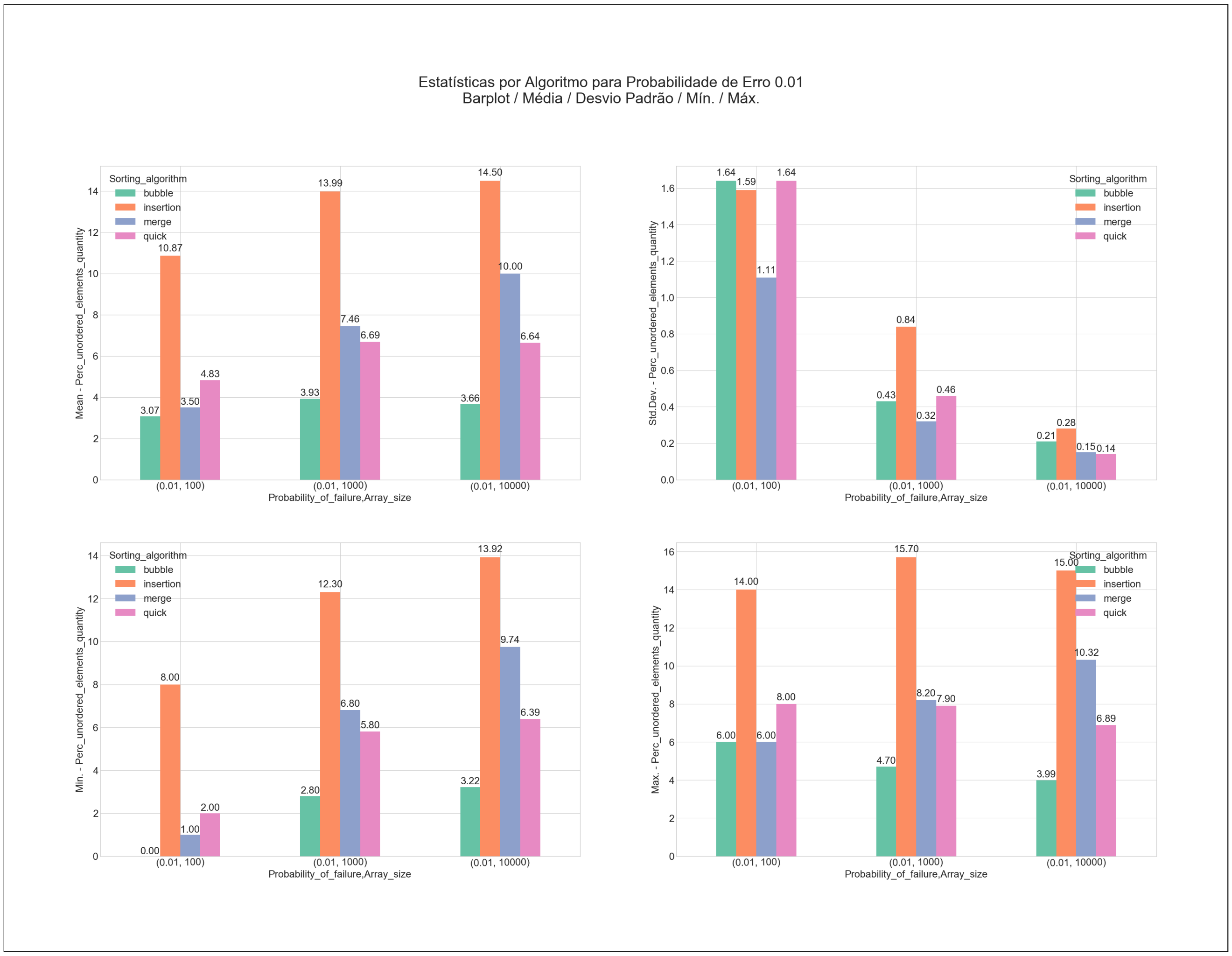

Fig. 8: Distribution data with *probability of failure* of 0.01, all algorithms and *array sizes*.

B. Dependente Variables Normality

An essential step to continue the analysis is to verify if dependent variables have a normal distribution. In this work, we produced the Q-Q plot to analyze the normality visually. We also ran the Shapiro-Wilk [12] test to confirm if they follow a normal distribution or not.

After the Shapiro-Wilk test, we determine that only the dependent variable %UEQ (*percentage of unordered elements quantity*) has a normal distribution related to mean for all algorithms. This situation occurs considering all possible combinations between independent variables: *sorting algorithm*, *probability of failure* and *array size*.

Based on these results, we choose to test just the hypothesis associated with the dependent variable %UEQ. We make use of the ANOVA method to test the hypothesis, and this method is premised the normal distribution of variables values.

Tables VII and VIII shows examples of results we get running Shapiro-Wilk test. The independent variables assume these values: *probability of failure* of 0.01 and *array size* of 100. The bold *p-values* confirms that the data is normally distributed.

TABLE VII: Shapiro-Wilk test for %LSS with *probability of failure* of 1% and *array size* of 100.

Algorithm	W	p-value
Bubblesort	0.854840	0.0007
Insertion Sort	0.961790	**0.3439**
Mergesort	0.937173	**0.0763**
Quicksort	0.869708	0.0016

TABLE VIII: Shapiro-Wilk test for %UEQ with *probability of failure* of 1% and *array size* of 100.

Algorithm	W	p-value
Bubblesort	0.934980	**0.0666**
Insertion Sort	0.949881	**0.1678**
Mergesort	0.936667	**0.0739**
Quicksort	0.936049	**0.0712**

Q-Q plot shows that how much more blue points close to the red line, most normal is the distribution. Figure 9 below presents this graph for the dependent variable *percentage of the largest subarray size* considering a *probability of failure* of 1% and a *array size* of 100 for all sorting algorithms.

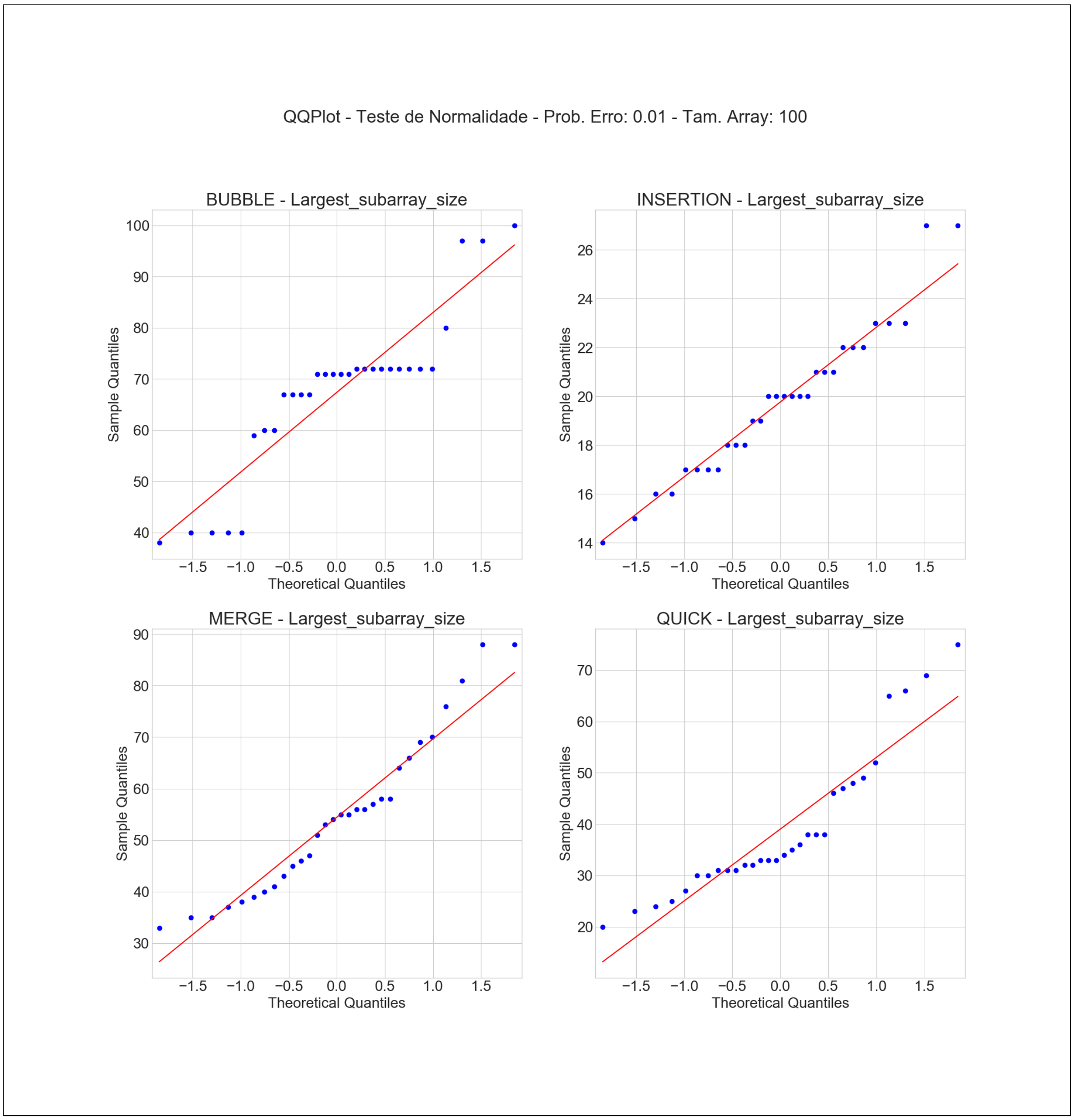

Fig. 9: Q-Q plot for %LSS with *probability of failure* of 1% and *array size* of 100.

Figure 10 shows the same graph for the dependent variable *percentage of unordered elements quantity* considering same values for independent variables.

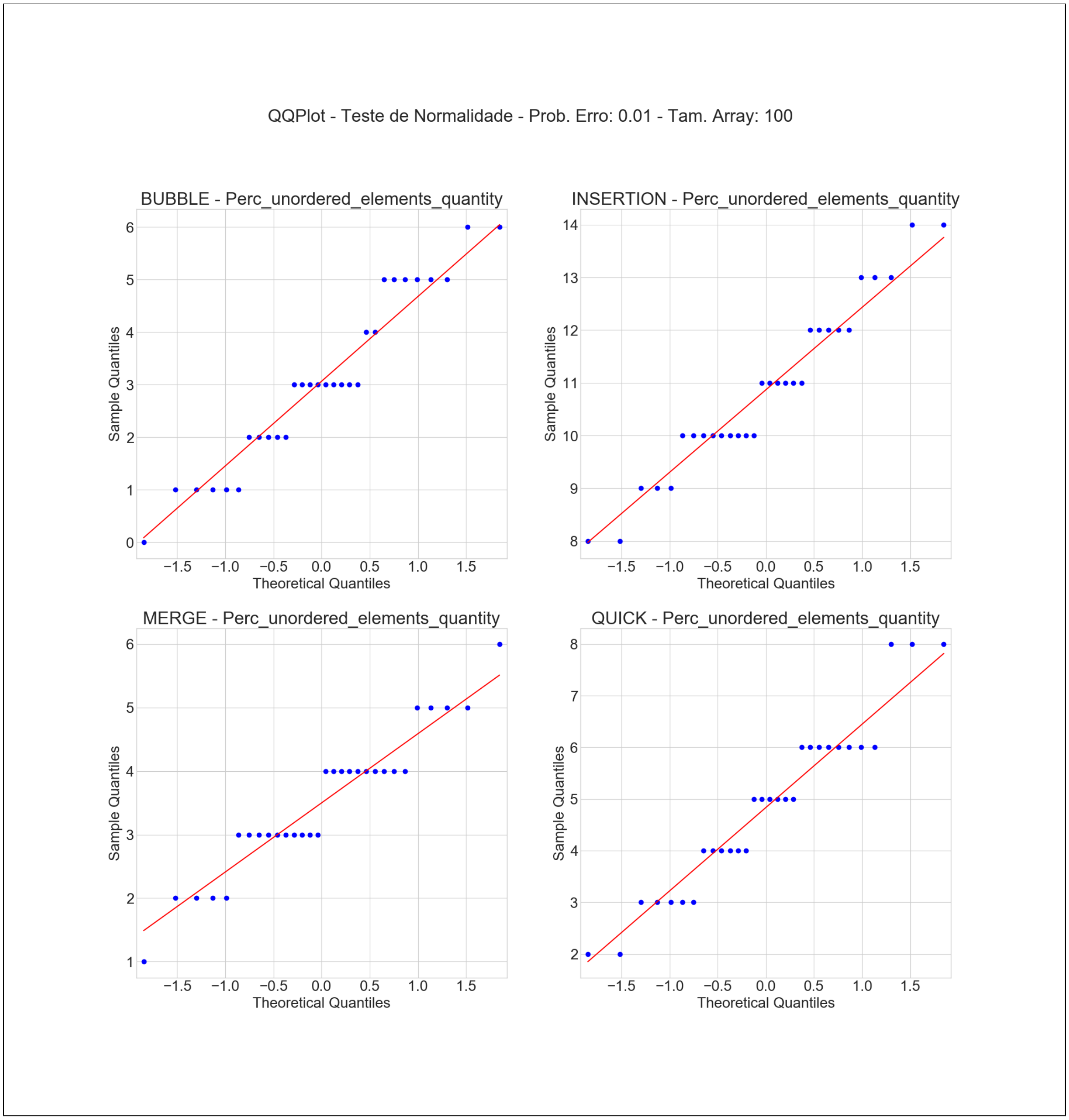

Fig. 10: Q-Q plot for %UEQ with *probability of failure* of 1% and *array size* of 100.

V. Results

VI. Discussion

VII. Conclusion

References

[1] I. Finocchi and G. F. Italiano, "Sorting and searching in the presence of memory faults (without redundancy)," in *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, no. January, pp. 101–110, 2004.

[2] S. Hamdioui, Z. Al-Ars, A. J. Van De Goor, and M. Rodgers, "Dynamic faults in random-access-memories: Concept, fault models and tests," in *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 19, pp. 195–205, 2003.

[3] NitinArora and S. Kumar, "A Novel Sorting Algorithm and Comparison with Bubble sort and Insertion sort," *International Journal of Computer Applications*, vol. 45, no. 1, p. 2.

[4] V. P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990.

[5] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[6] X. Wang, "Analysis of the time complexity of quick sort algorithm," *Proceedings - 2011 4th International Conference on Information Management, Innovation Management and Industrial Engineering, ICIII 2011*, vol. 1, pp. 408–410, 2011.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms.* MIT Press, 2009.

[8] D. Abhyankar and M. Ingle, "A Novel Mergesort," vol. 1, no. 3, pp. 17–22, 2011.

[9] K. Mansotra, V; Sourabh, "Implementing Bubble Sort Using a New Approach," pp. 1–6, 2011.

[10] P. Prajapati, N. Bhatt, and N. Bhatt, "Performance Comparison of Different Sorting Algorithms," vol. VI, no. Vi, pp. 39–41, 2017.

[11] U. Ferraro-Petrillo, I. Finocchi, and G. F. Italiano, "The price of resiliency: A case study on sorting with memory faults," *Algorithmica (New York)*, vol. 53, no. 4, pp. 597–620, 2009.

[12] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.