Algorithms speed test

By Filippo Piloni and Alessandro Zanzi

Experiment 1, Experimentation & Evaluation 2023

Abstract

The aim of the experiment was the prove which between BubbleSortPassPerItem, BubbleSortUntilNoChange and BubbleSortWhileNeeded algorithms had the best time performances.
To obtain valid results, the three algorithms have been tested in different scenarios, changing the size of the arrays, the ordering of the elements inside the arrays and the type of elements to sort using sizes of 100, 1000 and 5000 and as objects Integers, Bytes and Strings.
From the data gathered during the experiment, it has been clear that the algorithm BubbleSortWhile needed presents higher performances than the other algorithms in 26 out of 27 of the different scenarios

1. Introduction

Sorting algorithms play a crucial role in various computer science applications, influencing the efficiency and performance of data processing tasks. It is important to choose the correct and faster algorithm according to the situation and cases of the program to maximize the efficiency. For this reason, the aim of this experiment is to demonstrate which among the three selected algorithms, namely, BubbleSortPassPerItem, BubbleSortUntilNoChange, and BubbleSortWhileNeeded, provides superior speed and efficiency? This inquiry drives the motivation behind our experiment, aiming to uncover the comparative performance of these algorithms and shed light on their behavior under varying conditions such as the size of the array, the type of the elements inside and the order of the elements.
These algorithms, despite performing the same task, have significant different approaches to the problem:

- The BubbleSortPassPerItem algorithm iterates through the array multiple times, swapping adjacent elements if they are in the wrong order. It continues this process until the entire array is sorted.
- The BubbleSortUntilNoChange algorithm also swaps adjacent elements until the array is sorted. However, it repeats this process until no more swaps are needed, indicating that the array is fully sorted. This approach potentially reduces the number of iterations compared to pass-per-item.
- The BubbleSortWhileNeeded algorithm iterates through the array, swapping adjacent elements until no more swaps are needed. It keeps track of the maximum index where a swap occurred and reduces the size of the array to that index in each iteration. This approach aims to optimize the sorting process by avoiding unnecessary comparisons.

The pursuit of this investigation is driven by a curiosity about the practical implications of sorting algorithms in real-world scenarios. Understanding the nuances of algorithmic

behavior is essential for making informed decisions in software development, where optimal performance is often a critical factor.

The primary focus of our investigation is the evaluation of the three sorting algorithms in question, each representing a distinct approach to sorting. In this introduction, we provide a glimpse into the overarching goal of our experiment: to identify the algorithm that excels in terms of speed and efficiency.

To conduct our experiment, we chose to evaluate each algorithm using arrays of varied sizes and types of elements. Additionally, we varied the order of these elements to assess the algorithm's performance with arrays of random values, sorted values, and inverse-sorted values.

Hypotheses:
According to the general functionality of the algorithms, we can formulate the following hypothesis: 1. **Null Hypothesis**: The sorting algorithms taken into consideration demonstrate comparable performance in terms of sorting time across varying array sizes and object types. 2. **Alternative Hypothesis**: The sorting algorithms taken into consideration demonstrate significant differences in sorting time based on array size and object types.

2. Method

2.1 Variables

Independent variable	**Levels**
Algorithm Length of the array to sort Type of object in the array	PassPerItem, UntilNoChange, WhileNeeded 100, 1000, 10000 String, Integer, Byte

Dependent variable	**Measurement Scale**
Time required to sort the array	Seconds

Control variable	**Fixed Value**
Elements of the array	Random values, Sorted values and inverted-sorted values

2.2 Design

Type of Study (check one):

☐ **Observational Study**	☐ **Quasi-Experiment**	X **Experiment**

The experimental design involves manipulating independent variables to observe their effects. Through randomization of groups, we aim to control extraneous variables and establish causal relationships between the independent variables and the observed outcomes.

Number of Factors (check one):

☐ **Single-Factor Design**	X **Multi-Factor Design**	☐ Other

The study incorporates multiple independent variables, forming a multi-factor design. This approach allows us to explore the interactions and combined effects of these variables, providing a more comprehensive understanding of the phenomena under investigation.

For the experiment, all the possible combinations of independent variables have been taken into consideration for all the control variable, obtaining so 81 valid results, testing each algorithm with three different sizes and each size tested with three different type of elements. Everything has been repeated three times for each type of ordination of the elements in the array.

2.3 Apparatus and Materials

- Computer: model Dell Precision 5550
- Java: version openjdk 17.0.8.1 2023-08-24 (Used for creating the test)
- Python: version 3.8.10 (Used for creating the graphs)
- Libreoffice: version 6.4.7.2 40 (Used to save the datas)

2.4 Procedure

Test Execution Overview:

The experiment involves evaluating the performance of three sorting algorithms across different data types (Integer, String, Byte) and array sizes (100, 1000, 10000). The testing process is performed in the Test.java class.

Algorithm Initialization:

Before conducting tests, each sorting algorithm is initialized for every data type. To enhance performance, a warming-up process (warmup() method) takes place, utilizing an arbitrary sorter to ensure more consistent CPU scheduling for subsequent tasks.

Benchmarking Process:

For each array size under consideration, a suite of tests is executed using the benchmark() method. This function takes in the sorters of a specific type, a function to generate an array of that same type, and the array size. The benchmark initializes three arrays to record execution times for each algorithm and performs sorting operations N times. The generated arrays remain consistent across implementations, eliminating randomness. To ensure consistency a copy of the array is passed, not the array itself.

Detailed Steps:

1. Initialization: Sorters are initialized for Integer, String, and Byte data types.
2. Warm-up: The warmup() method ensures consistent CPU space occupation for subsequent scheduling.
3. Benchmark Execution: For each array size and data type, the benchmark() method records execution times for each algorithm across multiple repetitions.
4. Result Output: The mean execution times are printed to the terminal while the single executions per array are written to separate text files for further analysis.

Experimental Controls:

- Array sizes: 100, 1000, 10000
- Data types: Integer, String, Byte
- Repetitions: Each test is performed 100 times
- Warming-up cycles: 25

The comprehensive experimental setup, including warm-up cycles and repeated tests, ensured a solid and valid evaluation of the sorting algorithms' performance under varying conditions. The use of consistent arrays and detailed recording of execution times contributes to the reliability and reproducibility of the experiment.

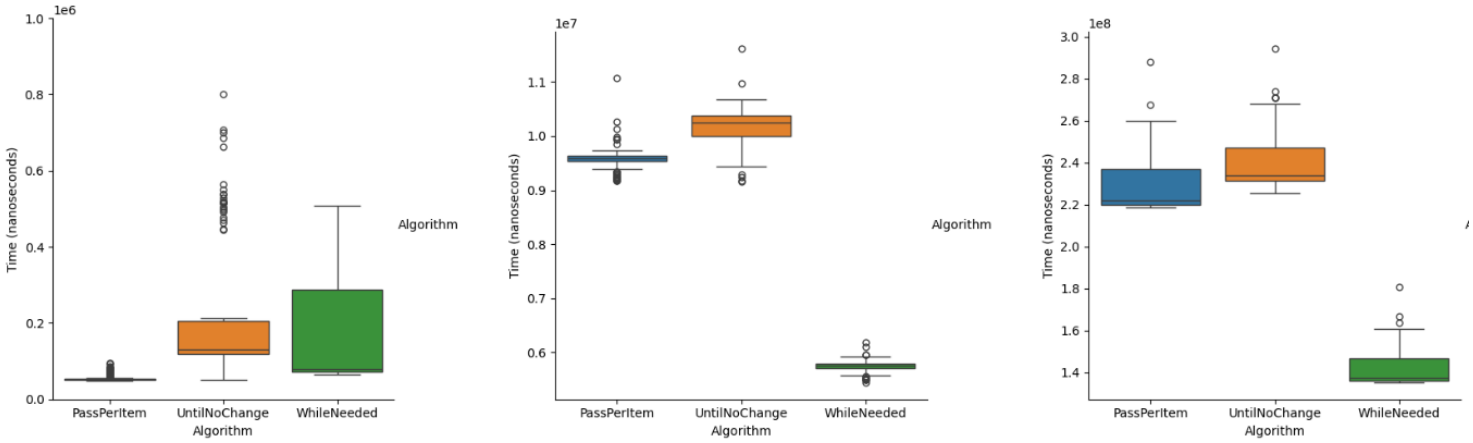
3. Results

The graphs and the data shown will represent only the array of random elements. The data for the arrays of sorted elements and inverse-sorted elements can be found in the Appendix. The interest results for them will be exterminated in Section 4, but in our opinion the array of random elements represents a more realistic and interesting case of study.

3.1 Visual Overview

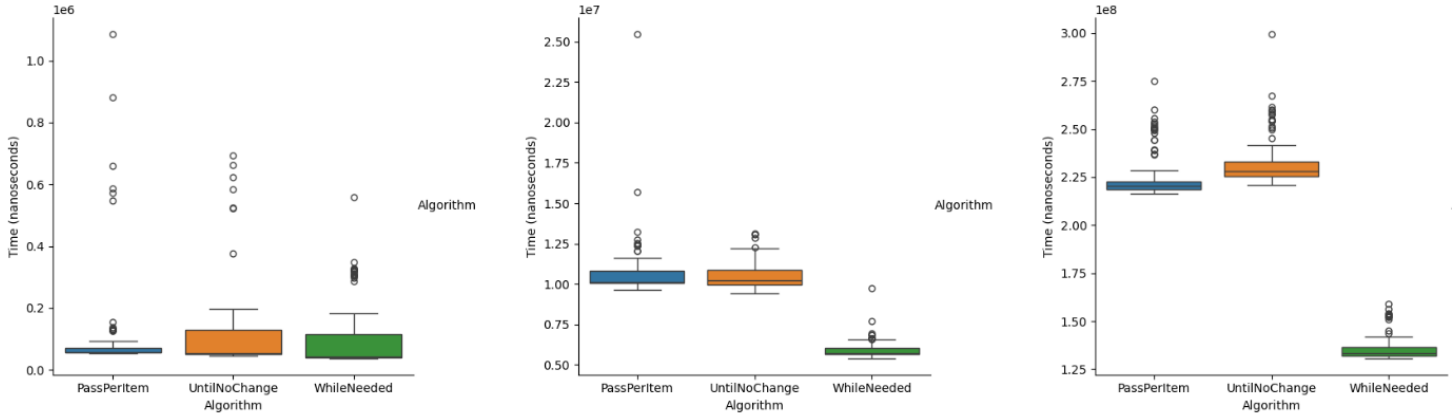
The graphs represent the time needed for the various algorithms to sort an array of different sizes of different types.

On the X axis are written the algorithm used.
On the Y axis is written the time in nanoseconds multiplied for the value on top of the axis (to improve visibility).
The scale of the graphs are all set to improve the visibility from the reader.



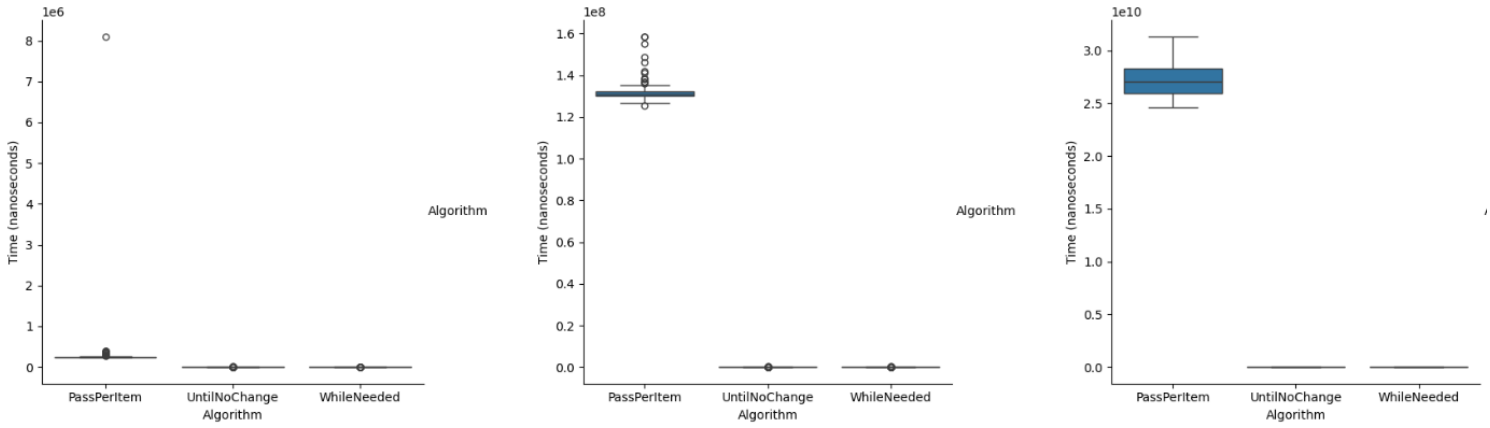
(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 1: Time to sort array of random Integers



(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 2: Time to sort random array of Bytes



(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 3: Time to sort random array of String

3.2 Descriptive Statistics

To provide a comprehensive overview of the time taken by each algorithm in various scenarios, we will present five key measurements: the minimum, first quartile, median, third quartile, and maximum.
The minimum and maximum values offer insights into the lower and upper bounds of each algorithm's performance in a specific scenario.
The first quartile signifies the value below which 25% of the data points reside, while the third quartile indicates the value below which 75% of the data points fall.
Lastly, the median provides an understanding of the central value within the distribution of data points.

FIRST TABLE: measurements for arrays of random Integers. **All values in microseconds (µs)**

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	49.03	51.52	52.74	53.95	95.51
100	UntilNoChange	50.31	118.71	130.34	205.79	800.69
100	WhileNeeded	65.17	71.95	77.95	288.50	507.59
1000	PassPerItem	9169.74	9541.96	9589.33	9643.04	11081.43
1000	UntilNoChange	9154.04	10001.68	10244.72	10379.44	11621.55
1000	WhileNeeded	5438.5	5697.70	5744.58	5789.28	6182.90
5000	PassPerItem	218450.07	219676.20	222031.12	236642.46	287952.51
5000	UntilNoChange	225384.25	231465.62	233995.79	246977.60	294343.50
5000	WhileNeeded	135107.66	135914.39	137273.98	146697.47	180580.24

SECOND TABLE: measurements fro arrays of random Byte. **All values in microseconds (µs)**

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	54.44	56.21	56.98	71.53	1086.22
100	UntilNoChange	46.38	51.74	54.85	128.87	692.78
100	WhileNeeded	37.32	40.00	41.83	114.51	559.82
1000	PassPerItem	9618.65	10046.31	10111.73	10839.853	25477.71
1000	UntilNoChange	9404.28	9945.08	10211.40	10860.93	13120.64
1000	WhileNeeded	5389.33	5637.88	5697.99	6019.42	9713.91
5000	PassPerItem	216418.49	218440.15	220268.62	222861.40	275246.85
5000	UntilNoChange	221121.31	225391.06	228321.18	233253.96	299524.33
5000	WhileNeeded	130591.06	132012.10	133435.01	136269.87	159040.65

THIRD TABLE: measurements fro arrays of random Strings of five letters. **All values in microseconds (µs)**

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	235.13	237.77	239.54	247.01	8101.433
100	UntilNoChange	2.71	2.76	2.82	2.98	14.13
100	WhileNeeded	2.66	2.73	2.78	2.99	11.05
1000	PassPerItem	125416.45	129899.18	130942.68	132303.21	158488.91
1000	UntilNoChange	124.92	131.20	131.93	133.24	427.18
1000	WhileNeeded	124.95	124.95	131.71	132.70	506.47
5000	PassPerItem	24635275.49	25924622.61	27040183.64	28272501.57	31318334.49
5000	UntilNoChange	4715.37	5218.96	5354.17	5616.20	5616.20
5000	WhileNeeded	4845.61	5207.87	5370.26	5640.08	6098.39

4. Discussion

4.1 Compare Hypothesis to Results

The data reveals important variations in the performance of the three algorithms, particularly when adjusting the array size and element types.

The observed increase in third quartiles and maximum values aligns with expectations, indicating that sorting time consistently grows with array size for each algorithm.

Sorting Strings, especially in lengthy arrays, demands substantial time, exemplified by the PassPerItem algorithm taking a maximum of 31 seconds to sort a 5000-element String array. This pattern persists even when sorting already ordered or inversely ordered arrays (see Appendix A [Figure 6 - Figure 9]).

Across all data, the PassPerItem algorithm consistently exhibits higher sorting times, except for small arrays of random Integers and Bytes.

For the UntilNoChange algorithm, the trend varies. In Section 3.1 and Appendix A, graphs show mixed behavior, with similar or higher performance than PassPerItem when sorting Integers or Bytes but significantly lower performance for Strings (except when sorting an inversely sorted array of Strings, Appendix A [Figure 6]).

The WhileNeeded algorithm consistently maintains high performance across changes in data type and array length, often surpassing or matching the performance of the UntilNoChange algorithm, with the exception of sorting 100 random Integers.

The collective data supports the validity of the Alternative Hypothesis, indicating substantial differences in sorting algorithms based on element types and array sizes.

4.2 Limitations and Threats to Validity

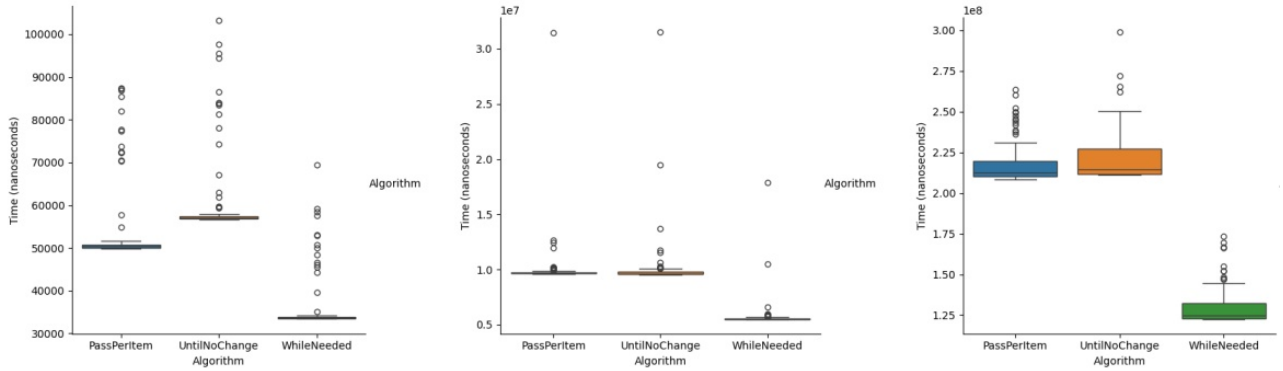
Despite our experimentation taking into consideration various cases, there might be some limitations in the study.
For example, the warm up processes used in this experiment could be enhanced, considering an higher number of warm up cycles or a warm up for the sorting of all the different data types instead of focusing only on one. This could effect the result of the time taken by each algorithm to sort arrays of Bytes or Strings.
Other factor that must be taken into consideration is that the experiment could be limited by the system specifics of Dell Precision 5550. To obtain more reliable results, the test should be conducted on the machines used by Bubble Inc, in order to choose the best possible algorithm given the client’s system specifics.

4.3 Conclusions

In summary, after conducting a series of tests, it's clear that the BubbleSortWhileNeeded algorithm consistently performs better in terms of time efficiency compared to the other two algorithms in the majority of cases.
Specifically, in 26 out of the 27 tests, BubbleSortWhileNeeded showed superior performance. This indicates that this algorithm is a reliable and efficient choice across different scenarios with varying data types and array sizes in the context of this experiment.

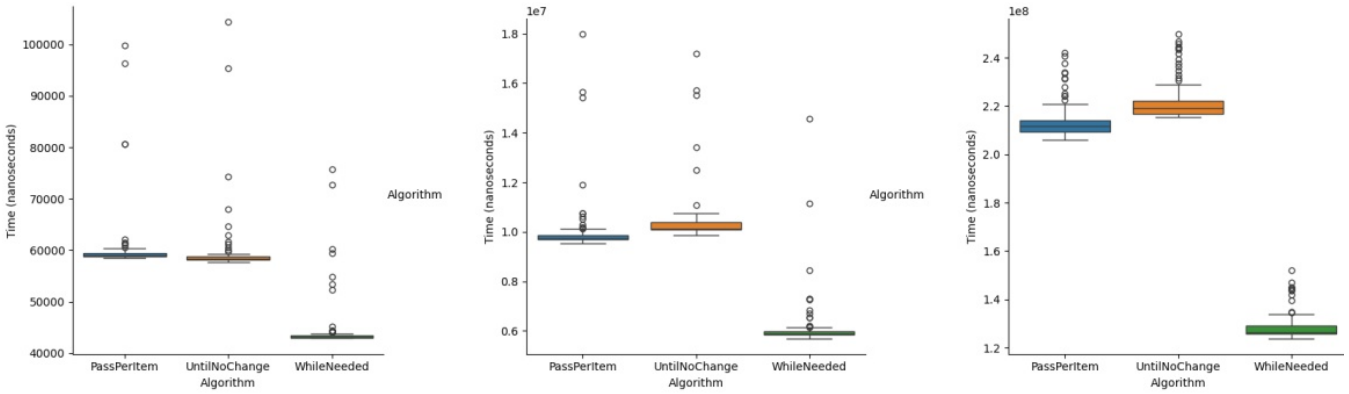
Appendix

A. Materials



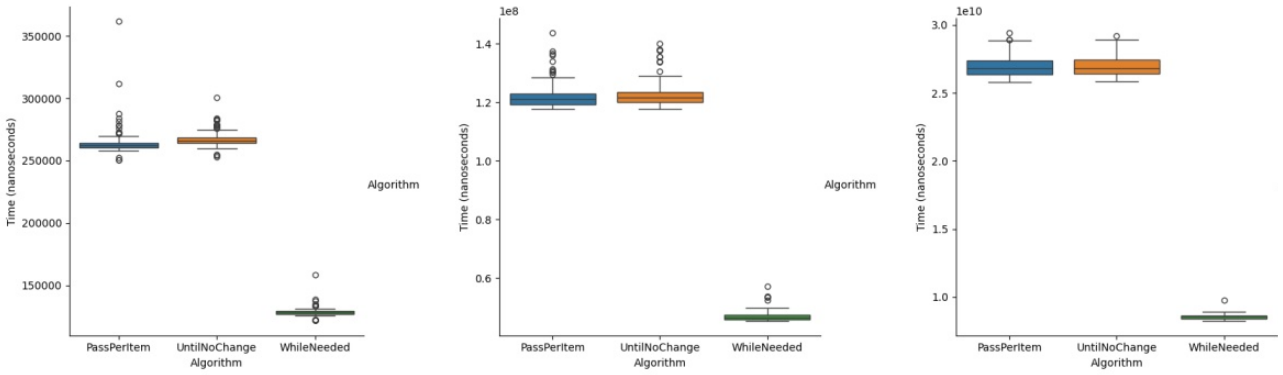
(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 4: Time to sort descending arrays of Integers



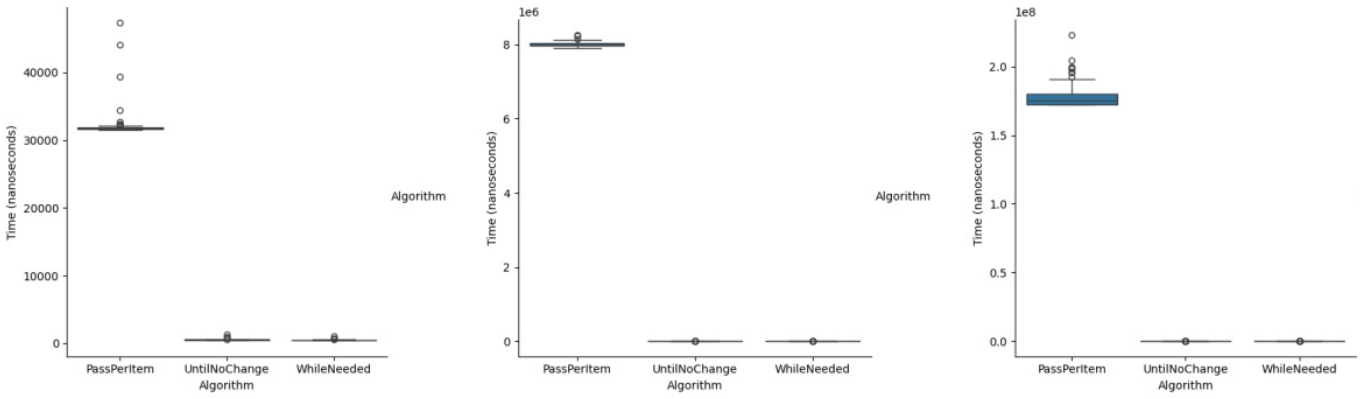
(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 5: Time to sort descending array of Bytes



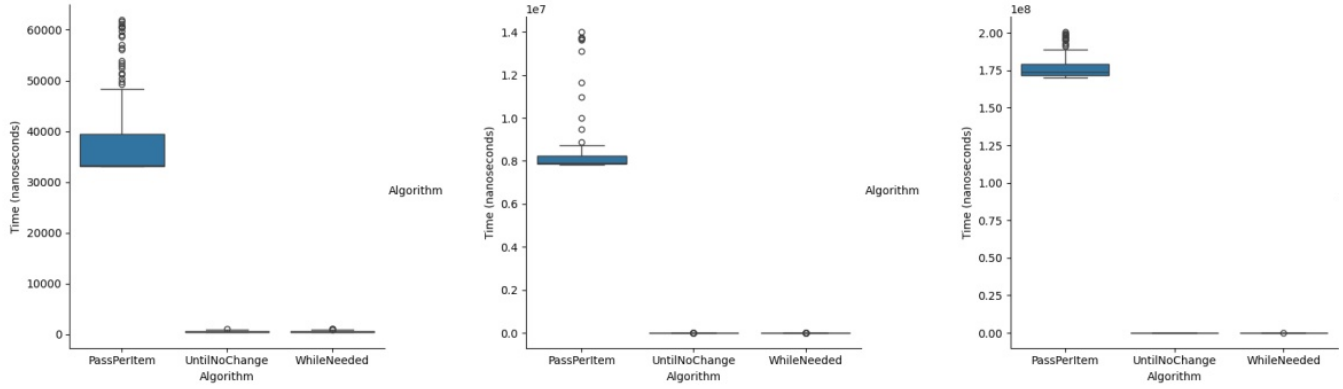
(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 6: Time to sort descending array of String



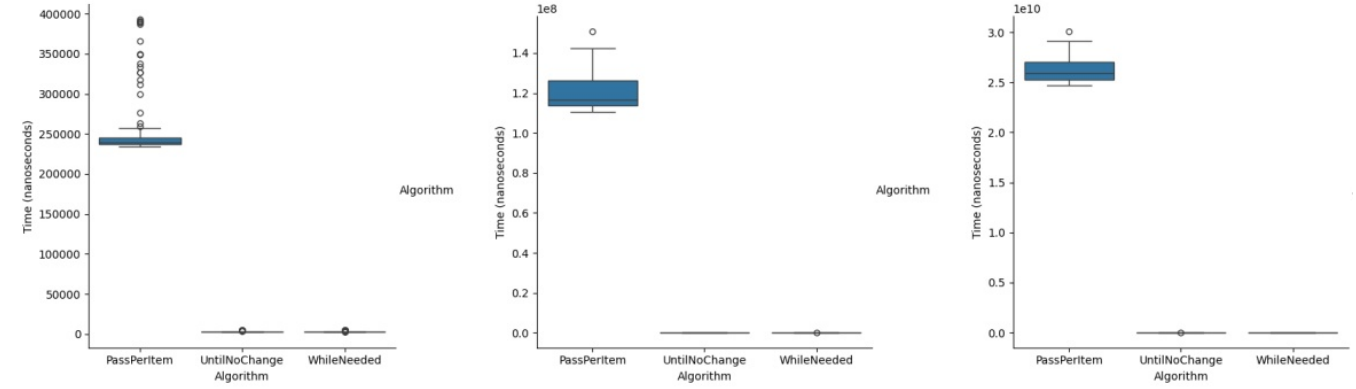
(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 7: Time to sort ascending of random Integers



(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 8: Time to sort ascending array of Bytes



(a) Size: 100 (b) Size 1000 (c) Size 5000

Figure 9: Time to sort ascending array of String

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	49.85	50.07	50.31	50.82	87.38
100	UntilNoChange	56.74	56.92	57.06	57.45	103.16
100	WhileNeeded	33.44	33.50	33.59	33.78	69.36
1000	PassPerItem	9606.17	9640.39	9665.68	9751.96	31493.50
1000	UntilNoChange	9548.54	9562.60	9575.98	9774.66	31540.57
1000	WhileNeeded	5481.91	5497.36	5507.04	5569.73	17908.92
5000	PassPerItem	208339.31	210349.57	212377.40	219466.91	263363.72
5000	UntilNoChange	210984.56	211619.51	214544.31	227371.03	298996.13
5000	WhileNeeded	122377.71	122787.50	124621.14	132163.16	173597.92

Table 1: Performance Metrics for arrays of Descending Integers (in microseconds)

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	58.54	58.77	58.96	59.42	99.76
100	UntilNoChange	57.71	58.16	58.32	58.76	104.34
100	WhileNeeded	42.91	43.02	43.13	43.43	75.81
1000	PassPerItem	9548.90	9708.79	9747.38	9877.97	17988.70
1000	UntilNoChange	9859.16	10082.39	10124.00	10387.20	17207.29
1000	WhileNeeded	5687.05	5857.50	5881.01	5975.96	14572.42
5000	PassPerItem	205850.43	209231.70	211795.54	214074.12	241904.37
5000	UntilNoChange	215364.05	216803.60	219137.71	222237.53	249848.90
5000	WhileNeeded	123721.80	125714.37	126542.95	129139.10	151971.81

Table 2: Performance Metrics for arrays of Descending Bytes(in microseconds)

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	250.51	260.69	262.30	264.39	361.71
100	UntilNoChange	252.84	264.19	265.84	268.57	300.53
100	WhileNeeded	121.71	126.99	127.92	129.11	158.24
1000	PassPerItem	117837.06	119357.54	121274.97	123099.46	143828.36
1000	UntilNoChange	117809.62	119980.49	121655.67	123569.40	140236.91
1000	WhileNeeded	45292.69	45868.37	46430.09	47424.17	57093.29
5000	PassPerItem	25781857.71	26383544.20	26828103.50	27380123.07	29426932.85
5000	UntilNoChange	25842208.87	26413720.27	26818579.97	27416407.67	29207443.63
5000	WhileNeeded	8228578.19	8406592.12	8543826.54	8635735.72	9775953.45

Table 3: Performance Metrics for arrays of Descending Strings(in microseconds)

Size	Algorithm	Minimum	First Quartile	Median	Third Quartile	Maximum
100	PassPerItem	31.47	31.64	31.71	31.84	47.30
100	UntilNoChange	0.46	0.51	0.53	0.55	1.28
100	WhileNeeded	0.42	0.47	0.49	0.50	1.05
1000	PassPerItem	7913.74	7981.70	8007.96	8041.87	8261.34
1000	UntilNoChange	8.72	8.83	8.95	9.11	13.34
1000	WhileNeeded	8.60	8.72	8.78	8.97	10.79
5000	PassPerItem	172160.11	172462.95	175633.39	179902.05	223133.42
5000	UntilNoChange	37.61	37.76	37.99	39.33	46.91
5000	WhileNeeded	37.06	37.18	37.39	38.58	69.91

Table 4: Performance Metrics for array of Ascending Integers (in microseconds)

B. Reproduction Package (or: Raw Data)

All the data and the, the graphics and the programs used are available in the following online Git-Hub repository: https://github.com/Ap0calypse2017/EEUSI