Sorting algorithm evaluation

Experiment 1, Experimentation & Evaluation 2020

Abstract

Bubble Inc. wants to test three implementations of the BubbleSort Algorithm in order to find the most performing with different kinds of inputs. I choose to test them with integer and string arrays with four different sizes to ensure scalability. I included a “worst-case scenario” to check for consistency on the various scenarios the company can experience in daily operations. The tests were performed ten times to include a warmup and mitigate inconsistencies related to the compiler and the machine I used. Some inputs are much slower to compare but this is not directly dependent from the algorithms under testing here, the results are still consistent between the two kinds of inputs. The results clearly show that BubbleSortWhileNeeded is faster in every scenario tested.

1. Introduction

Bubble Inc. is implementing a class library and has to include a sorting algorithm. We are given three implementations of the Bubble sort algorithm and we have to decide which one to use. We have to select the best algorithm based on its speed. The algorithms are:

- BubbleSortUntilNoChange
- BubbleSortPassPerItem
- BubbleSortWhileNeeded

This experiment will try to give an answer based on the measurement of the performance of the algorithm in different scenarios with different kinds of inputs.

Hypotheses:
The algorithm BubbleSortPassPerItem doesn’t stop when the array is sorted, for this reason I expect this algorithm to be the worst performing. I also expect the String type to be slower in comparison with Integer, and the reversed order Integer array to be the most time consuming in comparison with random order Integer arrays.

2. Method

2.1 Variables

Independent variable	Levels
BubbleSort Algorithm	BubbleSortUntilNoChange BubbleSortPassPerItem BubbleSortWhileNeeded
Input size	50 500 5000 50000
Input type	Random Integer array Random String array Reversed Order Integer array (worst case scenario)

Dependent variable	Measurement Scale
Time	Parametric Ratio scale

Control variable	Fixed Value
Power source	Plugged-in

Blocking variable	Levels
Background Processes	Difficult to measure and control

2.2 Design

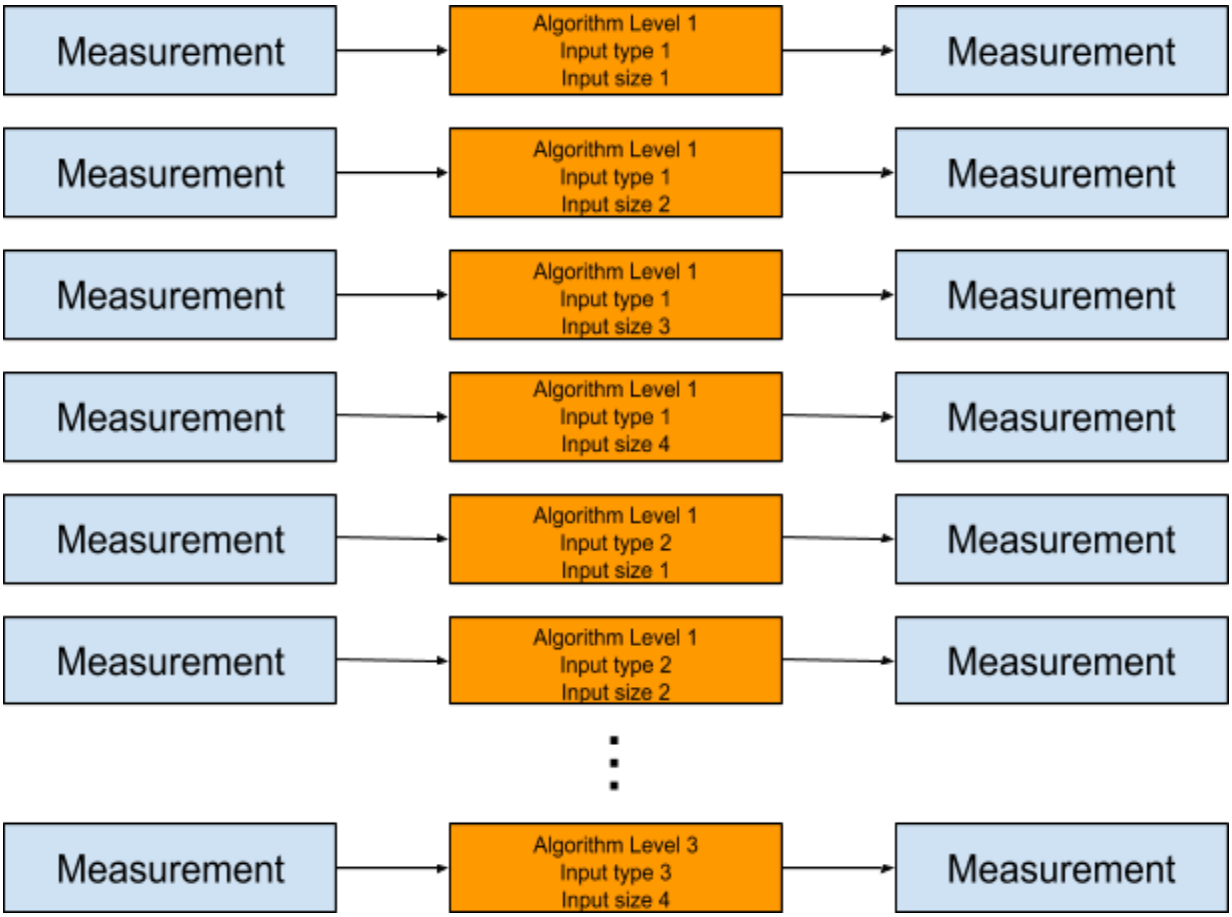
Type of Study (check one):

☐ **Observational Study**	☐ **Quasi-Experiment**	☒ **Experiment**

Number of Factors (check one):

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

This experiment is a Multi-Factor Design with the independent variables being the algorithm, the input type and input size. Due to the fact that I am not testing every kind of possible inputs and input sizes this is a Fractional factorial experimental design.
I have chosen this method, which takes into consideration different types of inputs and sizes, to make sure that the most common use-cases are covered. The randomization of the arrays ensures generalization and eliminates bias. I added a reverse order array to evaluate the performance of the algorithms in a worst case scenario that the end user could experience (even though very unlikely). The selected algorithm will be the one performing the best.



2.4 Apparatus and Materials

To perform the tests, I compiled and executed the code using Eclipse IDE for Java Developers 2020-09 (https://www.eclipse.org/downloads/). The computer used is a x64-based PC Dell XPS 13 9350 with Intel Core i7-6560U CPU @ 2.20GHz, 2208 Mhz, 2 Cores and 16 GB of Ram. The operating system used is Microsoft Windows 10 Home (10.0.18363 Build 18363).

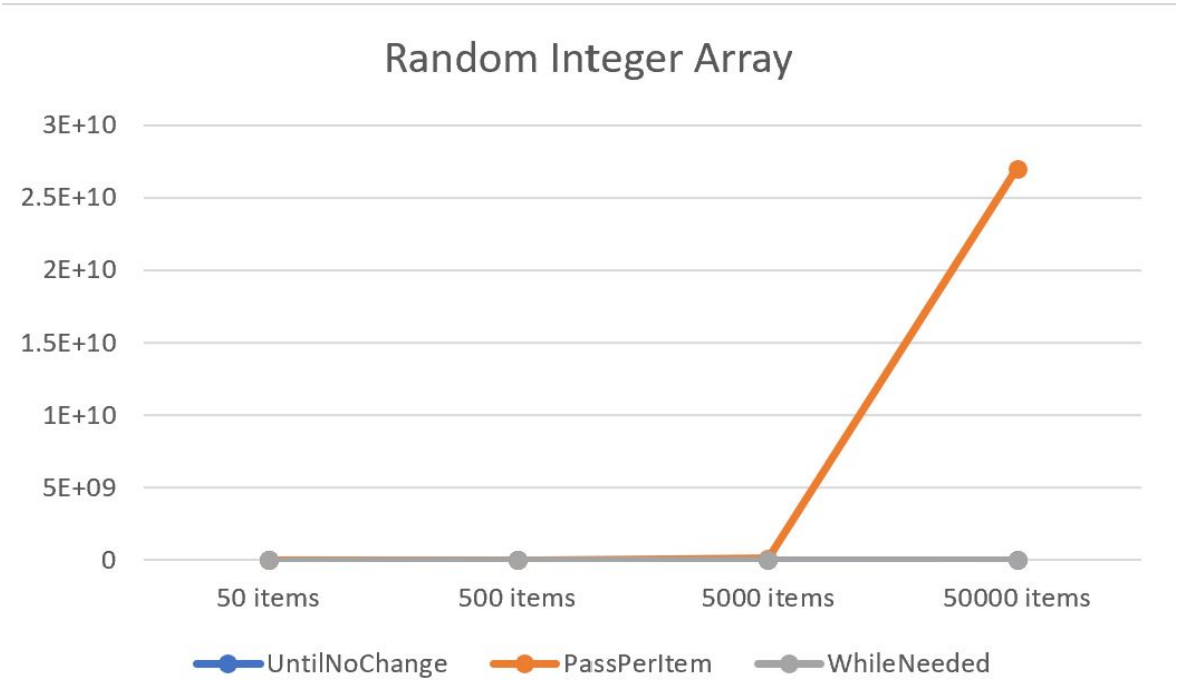
The time measurements were taken in nanoseconds with the Java.lang.System.nanoTime() Method before and after sorting the array.

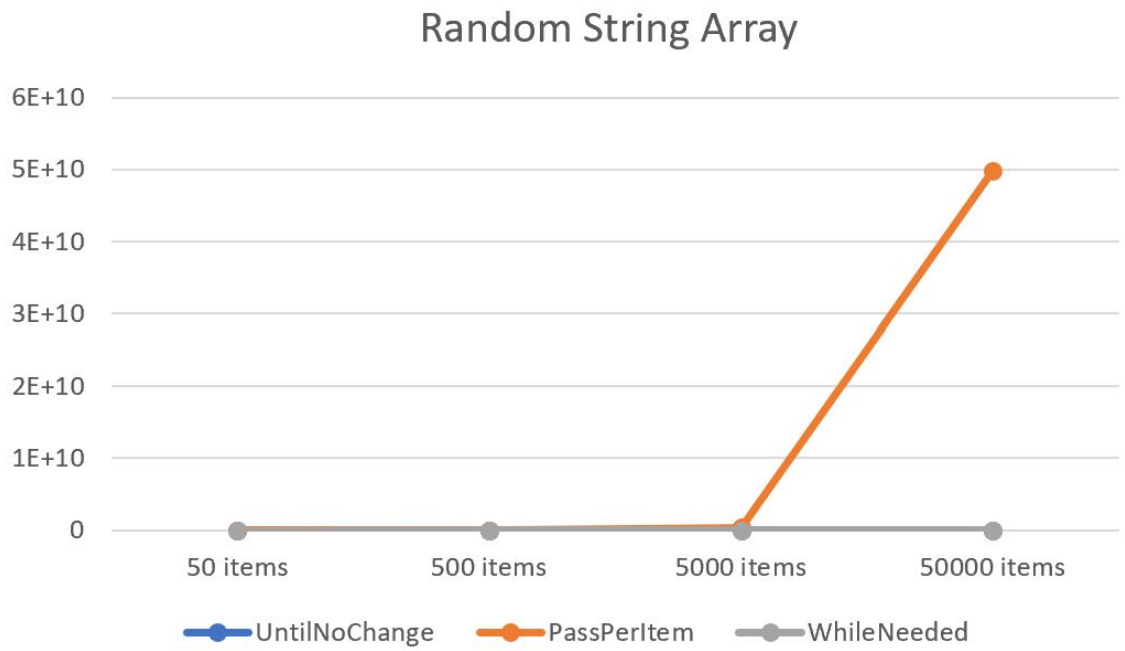
2.5 Procedure

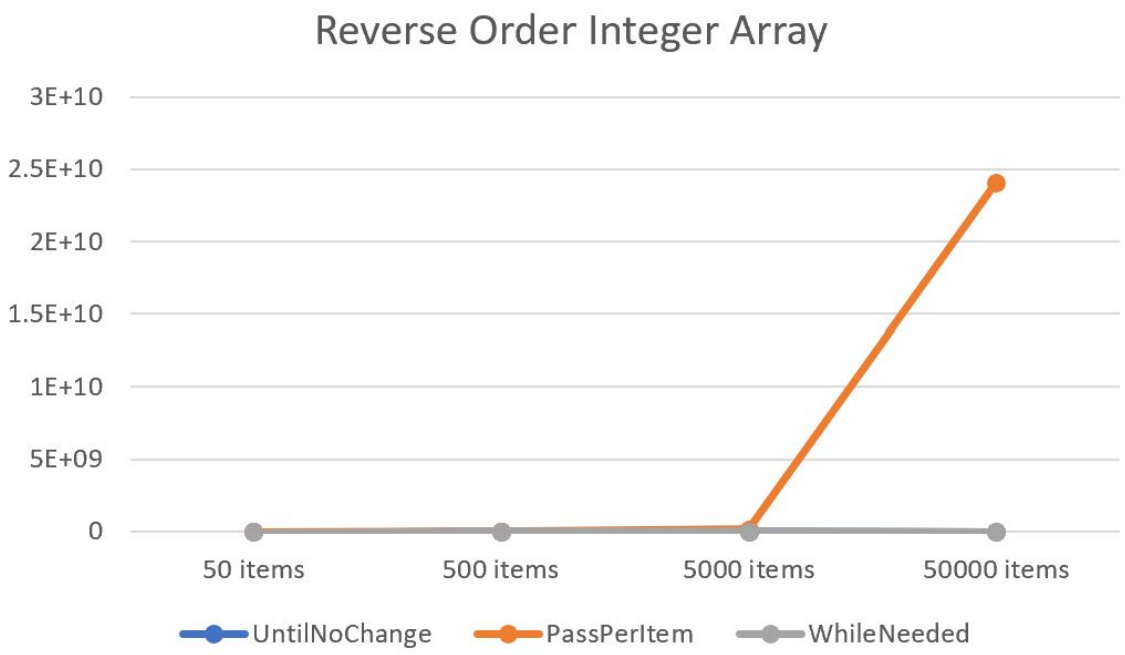
I started Eclipse to run the test and Excel to collect the data. I created a class Main.java that is the entry point of our tests. I first created a function to generate a random array of integer numbers for a given size, one for strings, and one for a random integer array sorted in reverse order. As the program is started, it generates the array and it passes it to the algorithm. I wanted to have some warm up for the compiler to optimize the program, for this reason I used a loop to perform 10 times the sorting. The time is recorded using the function nanoTime(). After programming the test I runned all 36 cases one after the other. Other open programs were Chrome, Notepad++, Acrobat Reader DC and Opera. The tests were carried out in the same session in order to keep room temperature constant and background processes as similar as possible. Out of the 10 runs I excluded the first one which always had the worst time. The final timing is a median of the last 9 runs.

3. Results

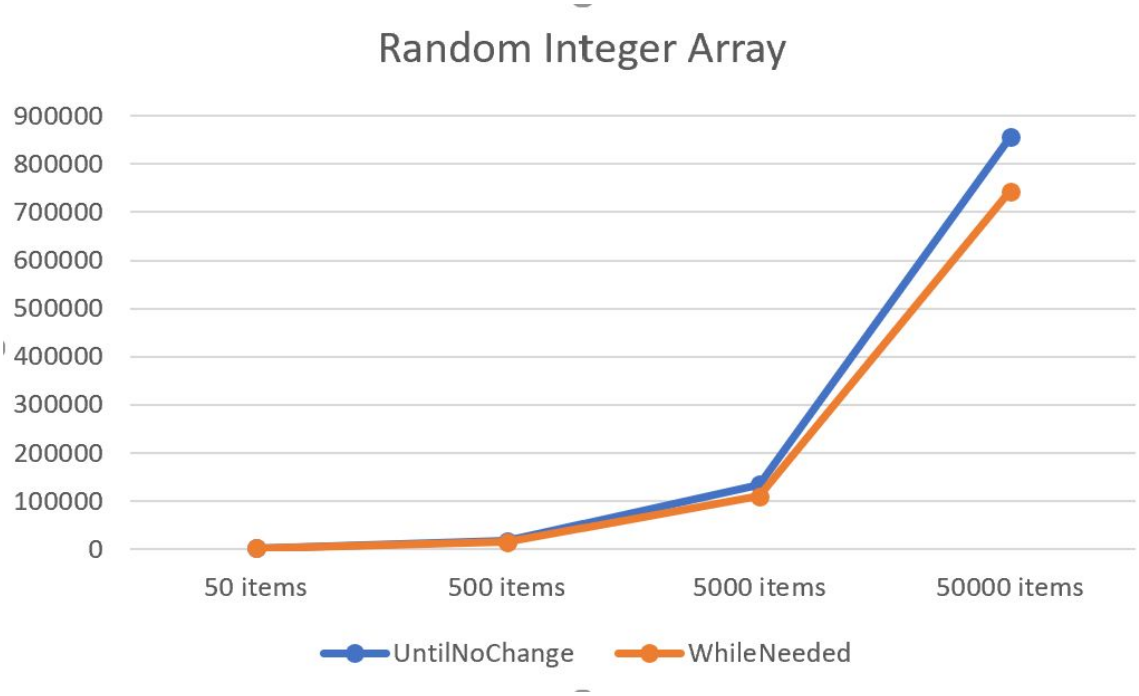
3.1 Visual Overview

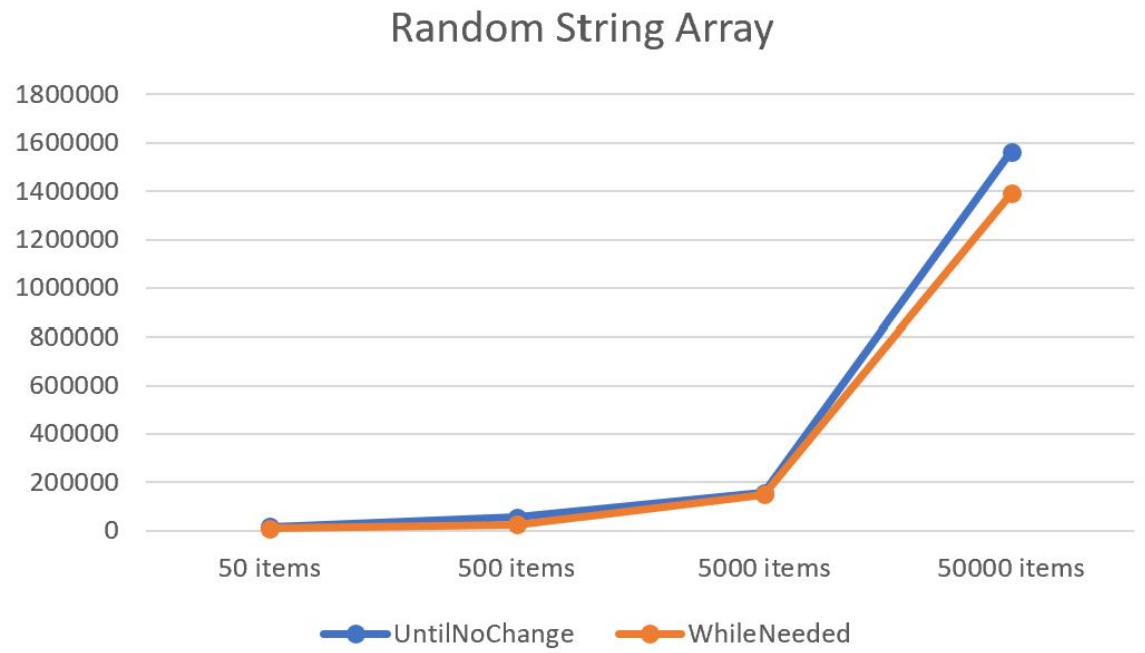


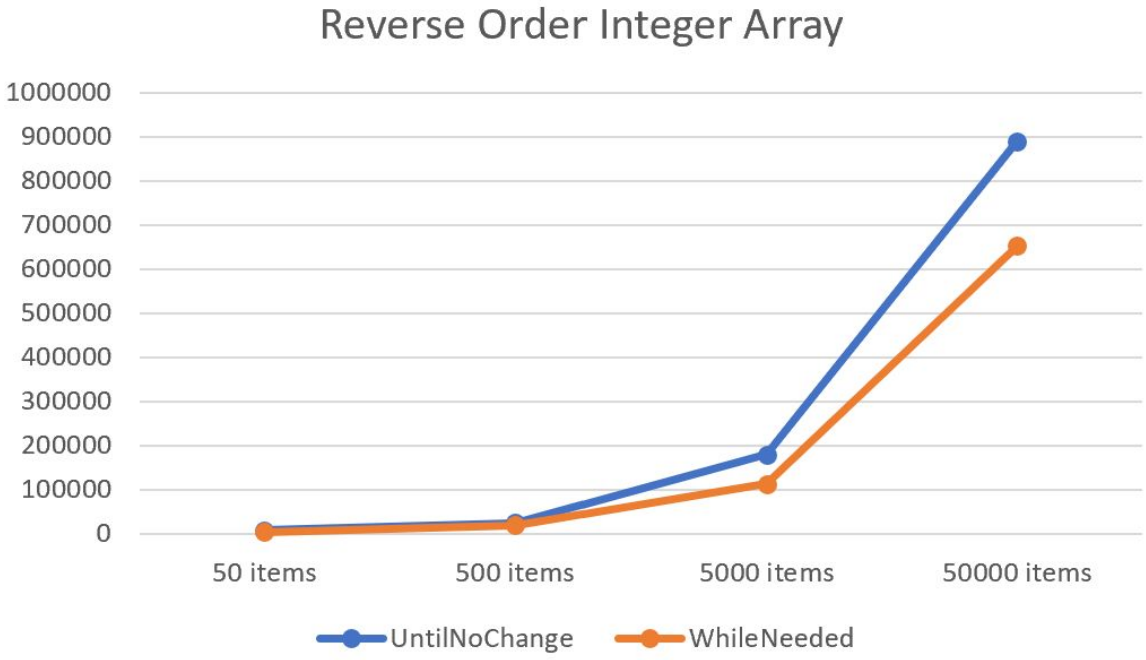




Detail of the algorithms UntilNoChange and WhileNeeded:




Random String Array
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
50 items
500 items
5000 items
50000 items
UntilNoChange
WhileNeeded


Reverse Order Integer Array
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
50 items
500 items
5000 items
50000 items
UntilNoChange
WhileNeeded

3.2 Descriptive Statistics

Experiment Results (ns)

BubbleSort Alg	Input type	50 items	500 items	5000 items	50000 items
UntilNoChange	Rnd Int	2911	17600	135067	855889
	Rnd String	14444	53556	155944	1562467
	RevSorted Int	8189	24733	179111	890000
PassPerItem	Rnd Int	276556	3455678	132880333	27026406811
	Rnd String	424211	5717244	346834178	49879290500
	RevSorted Int	296578	4725633	173644200	24081339022
WhileNeeded	Rnd Int	2622	15433	110567	743656
	Rnd String	8756	25533	148444	1389956
	RevSorted Int	3956	19600	113244	652233

For this experiment I report only the median result. The performance of each algorithm can vary for each execution due to different compiler optimizations, CPU scheduling, and available resources. For this reason I will not report other measurements, the average time mitigates the fluctuations of each run. The best time for every input size is highlighted in green.

4. Discussion

4.1 Compare Hypothesis to Results

The results show that the algorithms BubbleSortUntilNoChange and BubbleSortWhileNeeded are the fastest sorting algorithms. We can clearly see that BubbleSortPassPerItem is, as expected, by far the slowest algorithm, as it does not consider the fact that the array is already sorted, and continues to run.
Another observation we can make is that sorting type string is slower than type integer. This should be dependent from the implementation of the *compareTo* method. We can conclude that this difference in performance is not affected by our algorithms and thus the WhileNeeded and

UntilNoChange algorithms are faster than PassPerItem in all cases independently from the cost of the *compareTo* method.
The last result we can see is the fact that the worst case scenario (reverse order integer array) is almost always slower than just random order, as expected, but it is interesting to note that for PassPerItem and WhileNeeded, in the case of 50000 items, this is not the case. This difference should be explained by different CPU scheduling or different resources available (less cache available).
My hypothesis has been proved by this experiment.

4.2 Limitations and Threats to Validity

As seen in the results, the outcome of this experiment has some limitations due to compiler optimizations, CPU scheduling policies and different resources available. I tried to mitigate some of it by performing multiple runnings in the same session with the same conditions (that I can influence). Some conditions are very difficult to control and for this reason I had the result of faster times when sorting the reverse order array compared to random order. To overcome some of these limitations, one could perform these tests on a dedicated machine that is not performing all kinds of processes in the background, ensuring a better CPU scheduling time and more resources available (for example space availability in RAM and caches). One could also run more instances of the test in order to mitigate even more these fluctuations, for example 100 times.

4.3 Conclusions

We can conclude that BubbleSortWhileNeeded is the fastest implementation of the sort algorithm. The result clearly shows that with various types of input and sizes, even for the worst possible case, WhileNeeded beats the other algorithms. I advise Bubble Inc. to include BubbleSortWhileNeeded in their class library.