Experimenting on BubbleSort variations and their performance

Experiment 1, Experimentation & Evaluation 2022

Abstract

In this experiment we tried to find the fastest algorithm of sorting among BubbleSortPassPerItem, BubbleSortUntilNoChange and BubbleSortWhileNeeded. To do it we have chosen 3 independent variables in addition to the algorithms and they are the size of the list, type of the sorter and status of the list.
Each one has 3 values. For example, the status of the list can be already sorted (AS), randomly generated (RG) and sorted backwards (SB).
We have implemented a java program that returns all time measurements and saves them on a json file that we then examined to extract information about the performance of the algorithms.
For each combination of independent variables, we did 500 iterations to have more data to compare in the plots, the main findings were that the faster algorithm was BubbleSortWhileNeeded.

1. Introduction

The experiment was conducted with the objective of finding the fastest of the three given bubble sorting algorithms, BubbleSortPassPerItem, BubbleSortUntilNoChange and BubbleSortWhileNeeded.
All of the algorithms are guaranteed to be functional and correct, the only difference between them from a practical viewpoint is the time that they take to complete their execution.
The source code for the algorithms was available to be viewed and, as they are all implemented in the Java programming language, we performed the measurements by writing a Java program that would call the algorithms with different input arrays and automatically measure the time it took to sort them.

From reading the source code, we reached two hypotheses that relate to the speed of execution of these algorithms.

Hypotheses:
Our main hypothesis is that the `BubbleSortPassPerItem` will be the worst performing algorithm, since the others have a condition for which they can terminate early if the list is sorted at a certain point during the execution. However, in the cases where the array is arranged in the worst possible way before being passed to the algorithm (sorted backwards), the difference wouldn't be as noticeable.
The most noticable difference between the `BubbleSortUntilNoChange` algorithm and

the `BubbleSortWhileNeeded` algorithm is that the former always iterates over the array in full, while the latter will decrease the range of the array it iterates over as it sorts the array.
Therefore, we hypothesize that the `BubbleSortWhileNeeded` algorithm will generally perform faster than the `BubbleSortUntilNoChange` algorithm especially for large input sizes.

The independent and dependent variables are listed in section 2.1.

2. Method

In the following subsections we describe everything that a reader would need to replicate the experiment in all important details.

2.1 Variables

Independent variable	Levels
Size of List	The size of the list is categorized on three levels, SMALL, MEDIUM and LARGE. SMALL lists have 10 elements. MEDIUM lists have 100 elements. LARGE lists have 1000 elements.

Algorithm to use	The three algorithms provided. From here on referred to as:
	BSPPI (BubbleSortPassPerItem);
	BSUNC (BubbleSortUntilNoChange);
	BSWN (BubbleSortWhileNeeded);

Type of the Sorter	Three types of lists were sorted for a list of length n: INTEGER lists: containing integers from 1 to n; STRING lists: containing string representations of the integers from 1 to n of length 4, with leading zeros so as to be sorted the same way as the integers. DUMMY lists: containing dummy objects, being integer wrappers from 1 to n, while having had the `compareTo()` method modified to run slower for the first 100 times it's called on an object. This allows us to emulate large objects that take a long time to compare to each other.
Status of the List	Represents the way the array is given to the algorithm. Can be one of: Randomly generated (RG) (The elements are shuffled randomly with a fixed seed); Already sorted (AS); Sorted backwards (SB)

Dependent variable	Measurement Scale
Total Time	Nanoseconds, as measured by the `System.nanoTime()` method in Java.
Time per Item	This is calculated in nanoseconds after the experiment from the Total Time, depending on the size of the list.

Control variable	Fixed Value
Java version	Everything was run on as a single execution, using Java 14.
Machine Running the Code	The Experiment was run on a MacBook Pro 2019 with a 2.6 GHz Intel Core i7 processor, running macOS 10.14.6.

2.2 Design

Type of Study:

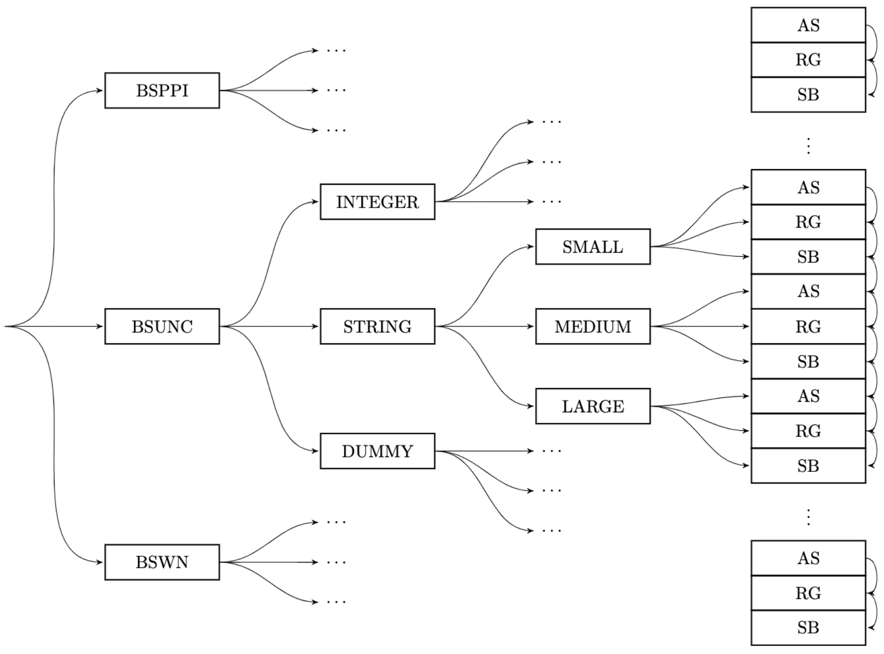
☐ **Observational Study**	☐ **Quasi-Experiment**	x **Experiment**

Number of Factors:

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

We designed the experiment as a Java program that would run once and gather all necessary data. The program iterates over all the independent variables in a nested for loop in order to generate instances of every possible combination of independent variables.

Below we can see a diagram of how the program generates the experiment instances:
For each algorithm, we iterate over the possible types, then over the array sizes and finally over the array statuses.



This means that the experiment follows the “Full factorial design” principle, where every possible combination of independent variables is tested. At the end, we have $3 \times 3 \times 3 \times 3 = 3^4 = 81$ total instances.

The final instances on the right of the diagram are executed top-to-bottom, one at a time and without breaks.
Each instance runs its algorithm 500 times and the measured times are stored in a json file.

In terms of experiment design, we can say that we have 81 sorters and we randomly assign a different treatment (random variation of the independent variables) to each, performing 500 single measurements on each one afterwards.

In the case where the arrays are randomly sorted, the shuffling uses a Random object created with the same seed (the seed being 0) for every instance. This means that two instances with RG arrays and the same array size will have the same array “order” if their algorithms are run the same number of times, even if the types are different.

2.3 Apparatus and Materials

The experiment was designed to be run from a single Java method. As such, it utilizes Java’s integrated `System.nanoTime()` method to measure the time it takes to run the algorithms under different inputs. The program used can be viewed and downloaded from its GitHub page. It is designed to be expanded easily, such as by having enums for each independent variable that can be easily expanded by adding more values.
The experiment was compiled and ran using the Java JDK 17.0.2 on a MacBook Pro 2019 with a 2.6 GHz Intel Core i7 processor, running macOS 10.14.6.

2.4 Procedure

The program was run a single time using Java version 17.0.2 with no settings modified, and by running the `Main.main()` method as the main method.
Details for the requirements can be found in the README file.

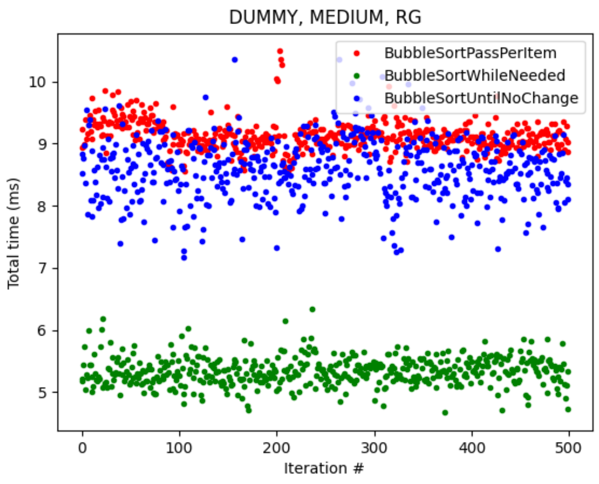
The procedure was completed in 474.833 seconds, according to the time measurement at the beginning and at the end of the program.

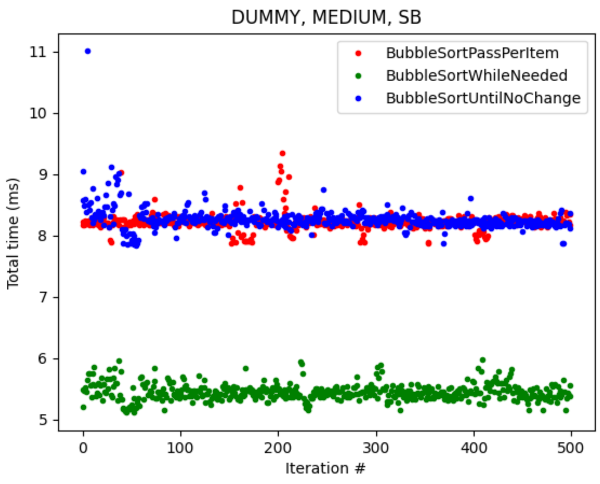
At the end, the raw data results can be found in the output.json file in the same directory. The graphs that can be seen in the 3.1 visual overview were produced from a separate python program. See the Reproduction Package (Appending B) for more information.

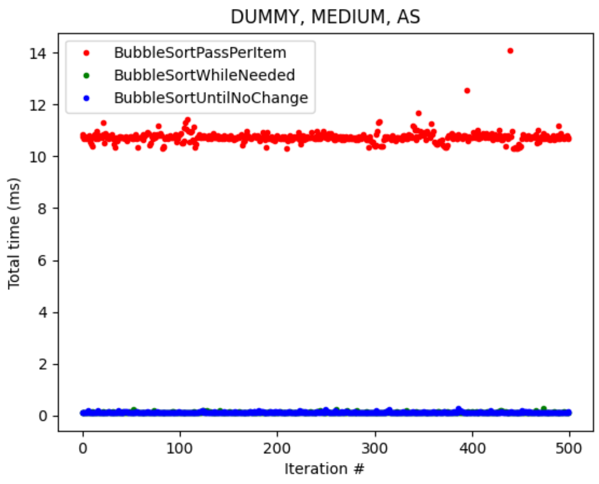
3. Results

In this section, we show only some of the plots that we generated from the data gathered. The rest can be found in the GitHub repository by running the python script or in the figures.zip archive.

3.1 Visual Overview



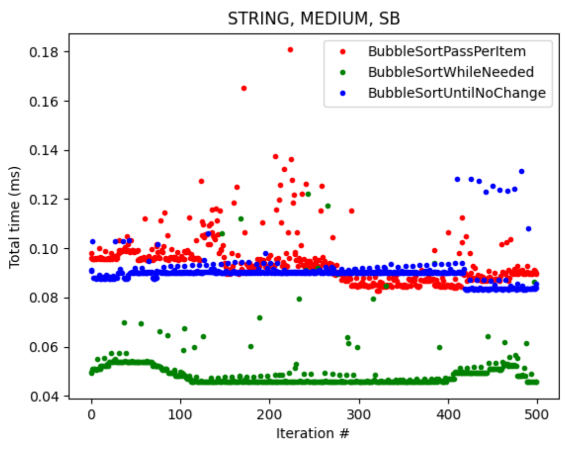


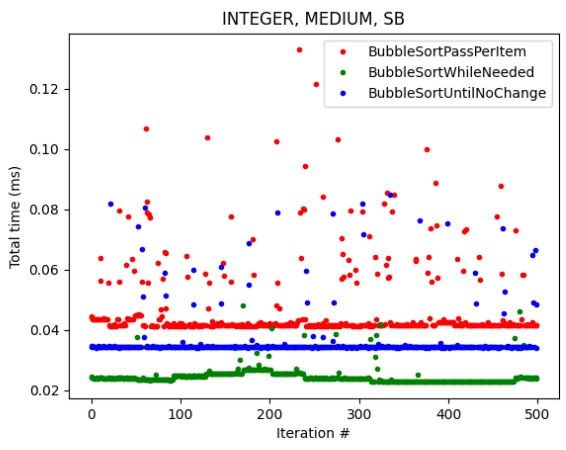


(1)

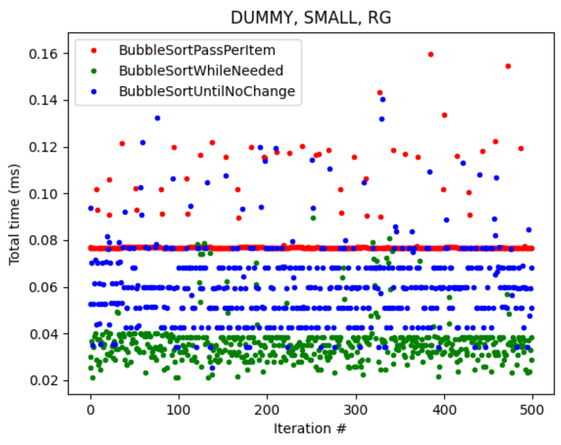
Here we can see the comparison of 3 plots, each plot has as common variables the type of the sorter (dummy) and the size of the list (medium), the variable that changes is the status of the list.

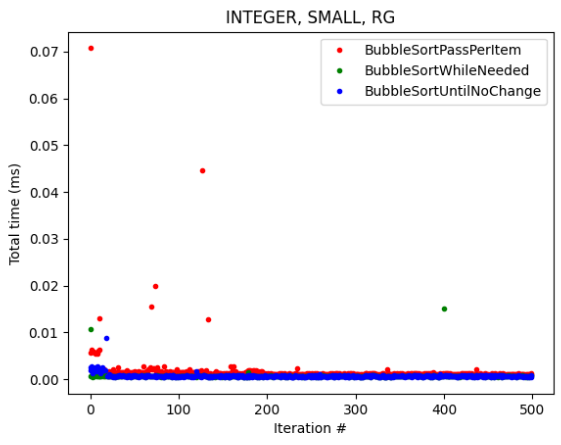
In the third plot we can see that if the list is already sorted, the BubbleSortPassPerItem algorithm does not end after the first loop and therefore does a lot of comparisons, while the other 2 recognize it and therefore take little time. The blue and the green are overlapped.



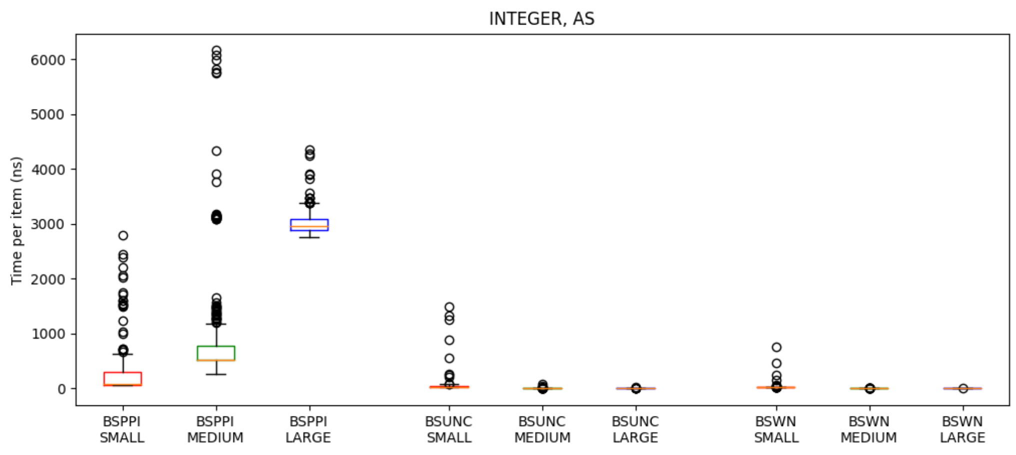


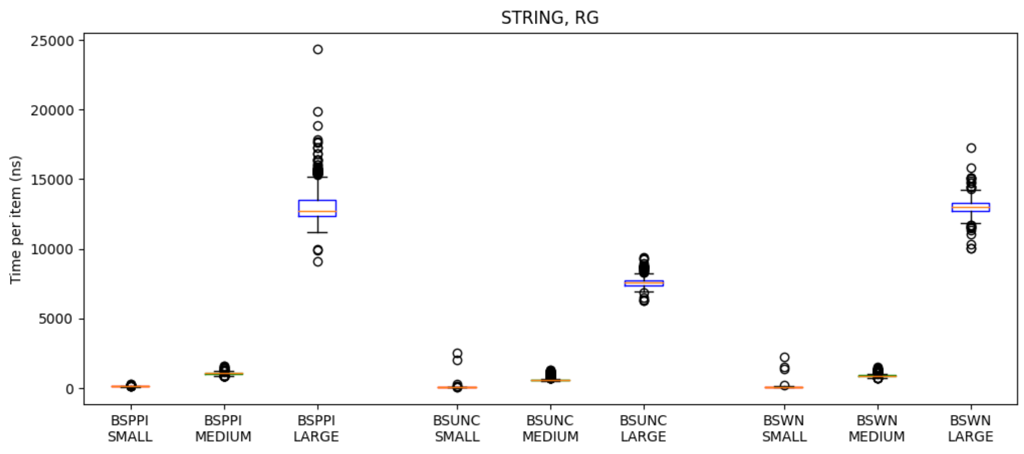
(2)
Here we can see the comparison of 2 plots, the common variables are the size of the list (medium) and the status of the list (SB)
We can notice that the plots are similar but when we have a list of strings, all the algorithms take longer than an integer list.



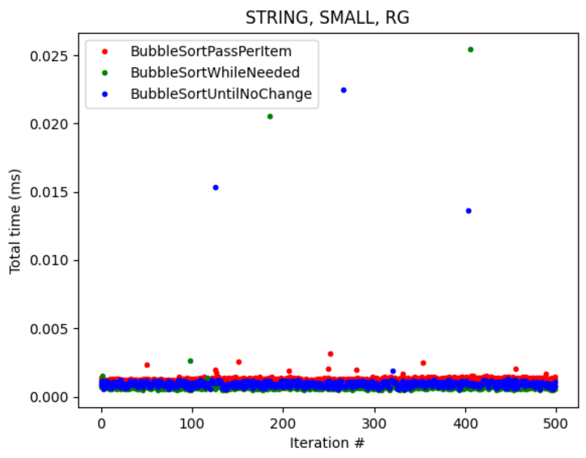


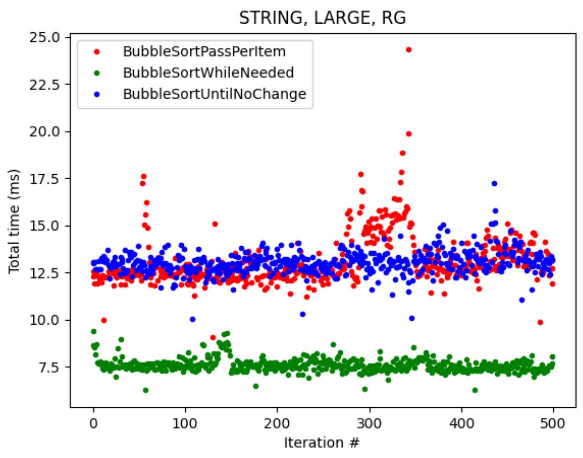
(3)
Here we can see the comparison of 2 plots, the common variables are the size of the list (small) and the status of the list (RG).
One of the quirks of the design of the Dummy object is that in the left plot we can clearly see the time it takes for a single `compareTo()` operation on a Dummy object as a “step” in the plot.





(4)
In these 2 plots we can see clearly the comparison between the algorithms behavior with the different sizes of the lists.
In the first plot we have as fixed variables the type of the data (integer) and the state of the list (AS).
In the second plot we have as fixed variables the type of the data (string) and the state of the list (RG).
We notice that even after accounting for the size of the array by dividing the total time by the input size the larger arrays still take more time per item. This can be easily explained by the fact that bubble sort is an algorithm that runs in $O(n^2)$ time.





(5)
Here we see the comparison between the instance with small array size and the one with large array size in the case of sorting arrays of randomly shuffled strings.
Notice that while in the SMALL case the times are mostly overlapped, in the LARGE case the green algorithm manages to gain an advantage over the other two in terms of speed.

3.2 Descriptive Statistics

The provided Python script, apart from generating the images used in the Visual Overview section, also calculates the "five-number summary" for each of the experiment instances of the Total Time dependent variable. Below is a table of some of the instances together with their summary.

Instance variables	Variables	First quartile	Median	Third quartile	Maximum
INTEGER RG SMALL BSPPI	939	1116.0	1204.0	1475.5	70746
INTEGER RG SMALL BSUNC	381	592.0	680.0	766.25	8901
INTEGER RG SMALL BSWN	433	610.25	674.0	746.25	15135
INTEGER RG MEDIUM BSPPI	38679	40995.25	43220.5	56088.0	498737
INTEGER RG MEDIUM BSUNC	30011	35339.25	37756.0	41050.5	403556
INTEGER RG MEDIUM BSWN	19896	22594.5	23970.0	199902.25	328893
INTEGER RG LARGE BSPPI	3431946	3585207.0	3637280.0	3742359.25	5270942
INTEGER RG LARGE BSUNC	2621759	2803288.0	2869574.5	2985084.0	3382663
INTEGER RG LARGE BSWN	1494419	1602900.25	1661884.5	1749886.75	2356845

The rest of this data can be viewed by running the Python script provided in the GitHub repository.

In the cases, like above, where the array is Randomly Generated we can see the difference between the median and the first and third quartile is sometimes significant, meaning that there is a lot of variation between iterations. This is because each iteration may require a different number of comparisons to make for the same algorithm than the previous.

Instance variables	Variables	First quartile	Median	Third quartile	Maximum
STRING AS SMALL BSPPI	907	957.75	1015.0	1364.25	44723
STRING AS SMALL BSUNC	165	169.0	179.0	363.0	30209
STRING AS SMALL BSWN	156	167.0	330.5	350.0	34378
STRING AS MEDIUM BSPPI	73258	117656.0	129849.0	134066.5	212335
STRING AS MEDIUM BSUNC	857	886.0	901.0	923.0	37851
STRING AS MEDIUM BSWN	864	892.0	910.0	949.0	15304
STRING AS LARGE BSPPI	7100465	9156089.25	9410721.5	9911476.75	16347559
STRING AS LARGE BSUNC	9334	22890.25	23620.0	24350.5	91187
STRING AS LARGE BSWN	11533	22876.0	23939.0	27238.5	96536

As we can see from these instances (where the array was already sorted before it was processed) we have that, outside of the notable BSPPI algorithm, a similar minimum, first quartile, median and third quartile.

The maximum value, however, hints us to the fact that there might be issues with the way the experiment was conducted (see section 4.2) as a result of the fact that in a perfect environment the time that the experiment measures for each instance would be nearly identical in cases where the array is sorted the same way for every iteration (that is, when the array status is not Randomly Generated).

4. Discussion

4.1 Compare Hypothesis to Results

The first hypothesis is supported by the data gathered by the experiment. We can see especially clearly in comparison (1) in section 3.1 that the red points are consistently the worst performing, even if in the case where the array is sorted backwards the difference is less noticeable.

The second hypothesis is also supported by the data, as in most graphs the BubbleSortWhileNeeded is the fastest of the three.
Particularly we can see in the comparison (5) that the BubbleSortWhileNeeded algorithm is faster than the competition especially when the input sizes are large.

4.2 Limitations and Threats to Validity

The experiment could have been improved by monitoring more accurately the status of the machine it was running on. For example, no measures of the temperature of the internals were taken during the experiment, which could have outlined a period of automatic throttling that sometimes occurs to keep the machine cool during periods of intense work.

From the descriptive information in section 3.2 (and the visualization of it provided by the box plots in section 3.1) that in most of the cases there are outliers that spike the maximum, making it much higher than the median. Since all of the iterations have the same parameters and variables, this is caused by some other process that slows down execution. Such process could be the garbage collector that periodically flushes the memory of the Java virtual machine, interrupting execution for a time in some of the instances. A more thorough test would have run the instances as separate processes.

4.3 Conclusions

From the experiment we can conclude that there is a clear hierarchy of the three algorithms:

(1) The BubbleSortWhileNeeded algorithm is usually the best performing algorithm, especially when the array is sorted backwards, or when the array is large and randomly sorted.
(2) The BubbleSortUntilNoChange falls between the other two in terms of performance when we want to sort a randomly shuffled string. However, it lies closer to the BubbleSortWhileNeeded algorithm when the array is already sorted and closer to the BubbleSortPassPerItem algorithm when the array is sorted backwards.
(3) The BubbleSortPassPerItem is often the worst performing of the three, especially when the array is already sorted.

Appendix

A. Materials

All of the materials used in the experiment are accessible from the GitHub repository.
To run the experiment, simply compile

The following resources helped in the making of this report:

- The bubble sorting library which provided implementation for the sorting algorithms.
- The information document about the assignment.
- The org.json Java library, which is included in the repository.
- This StackOverflow post, which was used as reference for the python script.
- The numpy and matplotlib python libraries, as well as this tutorial for finding the quartiles.
- The python documentation in regards to string formatting.

B. Reproduction Package (or: Raw Data)

The raw data can be found in the GitHub repository in the output.json file.
To generate the plots seen in this report, follow the instructions in the README file to run the python script, or unzip the archive provided (figures.zip).