Homework Assignment

Experimentation & Evaluation 2022

Experiment 1

Alessandro Gobbetti
Bojan Lazarevski

Contents

Università della Svizzera italiana
Faculty of Informatics
Switzerland

Abstract

In this report the execution time of three sorting algorithm is measured, compared, visualized and analyzed. As input, arrays of length $10, 10^2, 10^3, 10^4$ and types `int, double, string, small object, large object` are used. The execution time, in nanoseconds, is measured for each algorithm and the final output is stored in a `csv` file. After running the algorithms and observing the data and plots, we came to the conclusion that the algorithm `BubbleSortWhileNeeded` has the smallest execution time given our measuring variables. In the following sections the entire procedure for understanding and replicating the experiment can be found alongside with used references and materials. It usually takes a few hours to generate and analyze the entire data and plots.

1 Introduction

Bubble Inc. is a company providing utilities to Java developers. One new feature they wish to include is the implementation of a fast sorting algorithm. The company **has to** decide which one out of three possible algorithms to include in the final version of the library. Only the fastest algorithm, with smallest execution time, will be taken into consideration.
In this experiment, three candidate algorithms will be measured and their running time will be compared and analyzed. Plots and charts alongside a written description will be included to provide a complete analysis of the algorithms. The algorithm that performs the best, given our measurements, will be recommended to the company for deployment.

Hypothesis:

> *The execution time of all algorithms is the same. The independent variables, which we manipulate, are the algorithms, type of data stored in the array, the size of the array and initial ordering of the array. Considering these independent variables, the algorithms will be measured only by one dependent variable: execution time measured in* ns *(nanoseconds).*

2 Method

In the following subsections, all independent, dependent and control variables are listed, as well as a procedure on how to replicate the experiment. A complete procedure on how to run the experiment is present in subsection 2.4. Taking into consideration our variables, it generally takes around 5 hours to run all algorithms and generate the `csv` file of all measurements and draw the plots. For a faster execution of the experiment, the size of the array or the number of data types could be lowered, but precision is lost. The main benchmark for executing the code is found in class `BubbleSortComparator.java`. All plots can be generated by executing the python file `plot.py` and are stored in location `/plotting/plots`, see the README on GitHub for more info.

2.1 Variables

In our experiment we include independent variables (Table 1), dependent variables (Table 2) and control variables (Table 3). Each one of these variables is used in the experiment in order to capture more combinations of the expected output and try to get as precise as possible to the actual solution.

The independent variables are variables that we as experimenters manipulate. We are given three algorithms to test, we choose to perform those algorithms on arrays with 5 different data-types with 4 different lengths of 3 different orderings. This gives us $3 * 5 * 4 * 3 = 180$ different output scenarios that we need to further analyze and compare.

Independent variable	**Levels**
Algorithms	BubbleSortPassPerItem, BubbleSortUntilNoChange, BubbleSortWhileNeeded
Type of data	int, string, double, small object, large object
Size of array	$10, 10^2, 10^3, 10^4$
Ordering of array	randomized, ordered, reverse-ordered

Table 1: Independent variables: what we manipulate.

The dependent variable describes what we measure and in what scale we represent our measurements. All execution times of the algorithms is measured in nanosecond SI time unit.

Dependent variable	**Measurement Scale**
Execution time	ns (nanoseconds)

Table 2: Dependent variables: what we measure.

The control variables are the constant variables that ensure that every run of the experiment is executed in a stable environment to prevent bias in the results. We execute 1000 times the algorithms with 100 cycles as warm-up to measure the execution time on a specific PC model and Java version with approximate CPU activity. All arrays of data-type string have a fixed length on the string.

Control variable	**Fixed Value**
Number of executions	1000
Warm-up executions	100
Type of PC	DELL Precision 5540, with CPU Intel i7-9850H (12) @ 4.600GHz
Java Version	openjdk 17.0.5 2022-10-18
CPU Activity	Constant activity $< 5\%$
Length of string	10 characters

Table 3: Control variables: what we keep constant.

2.2 Design

Characteristics of the experimental design:
Type of Study: **Experiment**
Number of Factors: **Multi Factorial Design**

In this experiment there is use of 4 independent variables, therefore it is a multi factorial design that allows us to analyze how they interact with each other. There are also control variables and a dependent variable. The experiment is performed in an artificial environment where elements to the arrays are assigned randomly. Every control group undergoes a run into 3 different algorithms and the execution time of each run is measured.

2.3 Apparatus and Materials

For this experiment the following materials were used:

1. PC: Dell Precision 5540 - model 2020, with CPU Intel i7-9850H (12) @ 4.600GHz running Ubuntu 22.04.1 LTS x86_64
2. terminal: gnome-terminal
3. Java Version: openjdk 17.0.5 2022-10-18

The final experiment has been run after a minute from the computer being turned on. For the hole experiment the computer was not used for any other task and the battery was fully charged.
All arrays are randomly generated with the functions `create[type]Arrays()` where `type` is the data type of the items in the array. To keep track of the time, the method `System.nanoTime()` is used to keep the starting

and ending time, measured in nanoseconds. To find the execution time, the ending time is subtracted from the starting time.

2.4 Procedure

As mentioned in subsection 2.1, in this experiment, we measured the 3 sorting algorithms using lists of several data-types:

1. `Integer`: each integer is randomly generated with `(int)(Math.random()*100);`
2. `Double`: each double is randomly generated with `Math.random()*100;`
3. `String`: each alphanumeric string in the list is generated randomly given the length (we used 10);
4. `DummmySmallObject`: a small java object containing just an integer value used for the sorting;
5. `DummyLargeObject`: a big java object containing an integer value used for sorting and a big list of large strings. (In this experiment we used lists of 100 identical string of length 100.

For each data-type we generate 3 arrays: one generated with random elements, then we have a sorted version of the first array and finally a reversed sorted one. The idea behind this is that sorting an already sorted array might be faster, while sorting a reversed sorted array should in theory be the worst case. Below we show an example for for the integers:

```
private List<Integer[]> createIntArrays(final int length) {
    Integer[] intArray = new Integer[length];
    // initialize intArray with random numbers
    for (int i = 0; i < length; i++) {
        intArray[i] = (int) (Math.random() * 100);
    }

    Integer[] intSortedArray = Arrays.copyOf(intArray, intArray.length);
    Arrays.sort(intSortedArray);

    Integer[] intReverseSortedArray = Arrays.copyOf(intSortedArray, intSortedArray.length);
    Collections.reverse(Arrays.asList(intReverseSortedArray));

    // return a list of all arrays
    return Arrays.asList(intArray, intSortedArray, intReverseSortedArray);
}
```

Now that we have the data we are ready to run the benchmark and measure the time it take to sort the array. To do so, as already mentioned in subsection 2.1 we first run the sorting algorithm 100 times to warm-up the java virtual machine, then we start the measurements and we sort the array another 1000 times to be more precise.

```
...
// skip the first few runs
for (int i = 0; i < warmUp; i++) {
    sorter.sort(Arrays.copyOf(items, items.length));
}
// measure the start time
long startTime = System.nanoTime();

// run the sort for the specified number of times
for (int i = 0; i < numberOfTimes; i++) {
    sorter.sort(Arrays.copyOf(items, items.length));
}
// measure the end time
long endTime = System.nanoTime();
...
```

The sorting algorithms overwrite the input array with the sorted one, so every time we sort it we have to provide a copy of the original array. Otherwise from the second time we sort the array it will be already sorted and our results will be falsified. But in doing so we are spending time in coping the array, so, to avoid measuring something else that it is not the sorting we measure the time it takes to copy the input array and then we adjust the final time by subtracting the coping time.

```
...
long startCopyTime = System.nanoTime();
for (int i = 0; i < numberOfTimes; i++) {
    Arrays.copyOf(items, items.length);
}
long endCopyTime = System.nanoTime();


long sortTotalTime = endTime - startTime - (endCopyTime - startCopyTime);

return (double) (sortTotalTime) / numberOfTimes;
...
```

Finally we are ready to compute the final result time: the mean time it took for each sorting.
The results are written into a csv file organized as following:

```
Algorithm,Type,Array,Size,Time
BubbleSortPassPerItem,Integer,random,10,1413.215
BubbleSortPassPerItem,Integer,sorted,10,1580.353
BubbleSortPassPerItem,Integer,reverse,10,2493.837
...
```

We are now ready to use this data to create plots. To do so, we switch to python which has several useful functions for plotting. We use `pandas` to read the csv file and `matplotlib.pyplot` for plotting. During the plotting, some metrics are also computed, printed to the console in a table and also saved in a csv file.

3 Results

In this section we present the results of the experiment. We first start by presenting a visual representation of the results in subsection 3.1, then we show some statistics in subsection 3.2.

3.1 Visual Overview

In this section we present a visual overview of the results. This is done by plotting the execution time of the sorting algorithms for each data-type and array type. The execution time, measured in nanoseconds, is plotted against the size of the array.
Note that every plot is plotted using logarithmic scale for the x and y axis, so that the plots are more readable. In each of the following figures we present 3 plots, one for each array ordering type: random, sorted and reversed sorted. The 3 sorting algorithms are plotted in different colors as also shown in the graph legends: BubbleSortPassPerItem is plotted in blue, BubbleSortUntilNoChange in orange and BubbleSortWhileNeeded in green. The points in the plots are connected by a line to make it easier to read the plots.

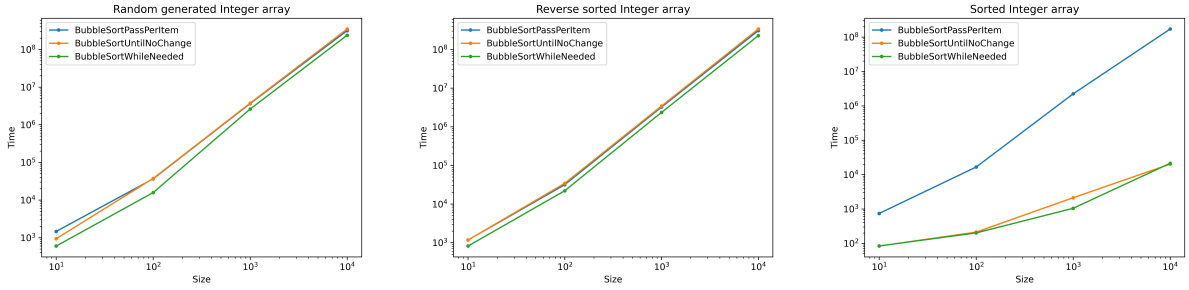

Figure 1: Data type: Integer

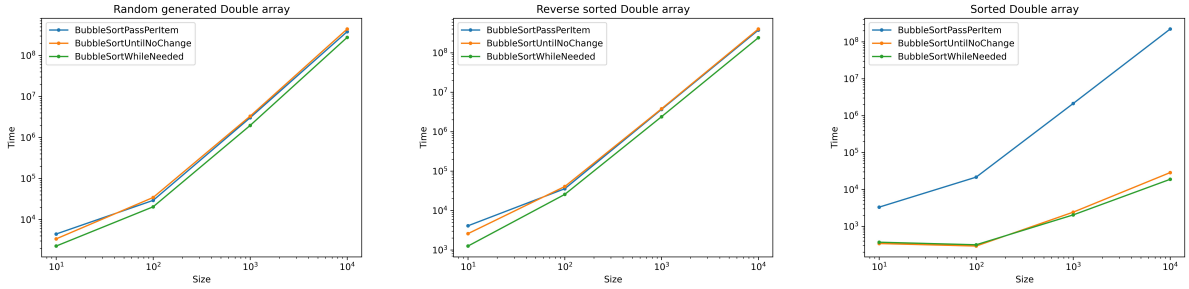

Figure 2: Data type: Double

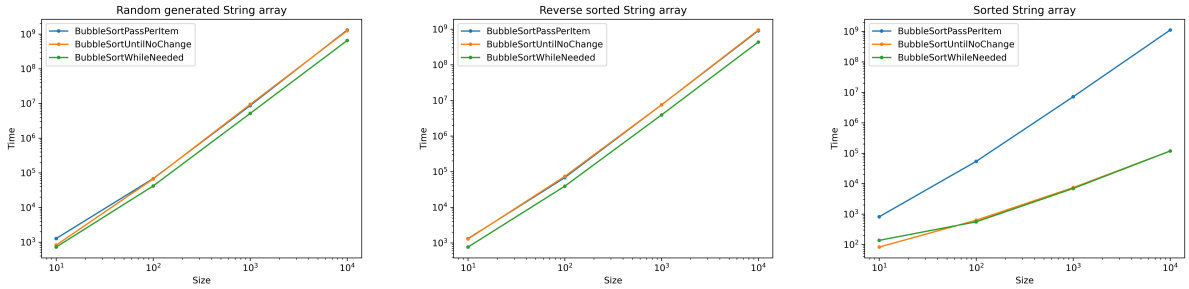

Figure 3: Data type: String

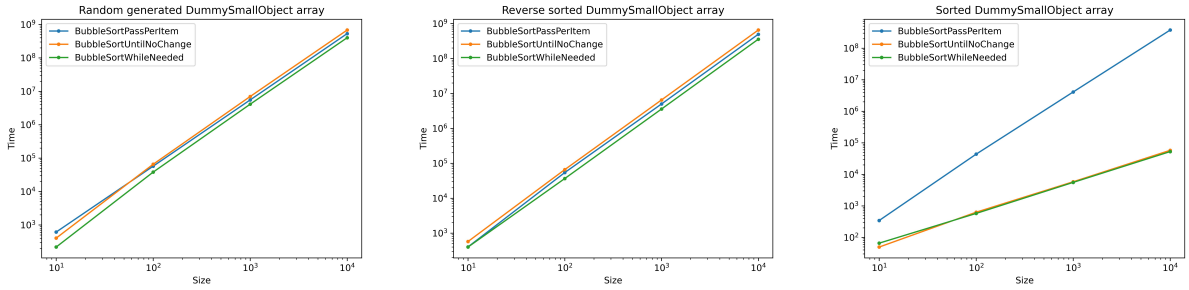

Figure 4: Data type: DummySmallObject

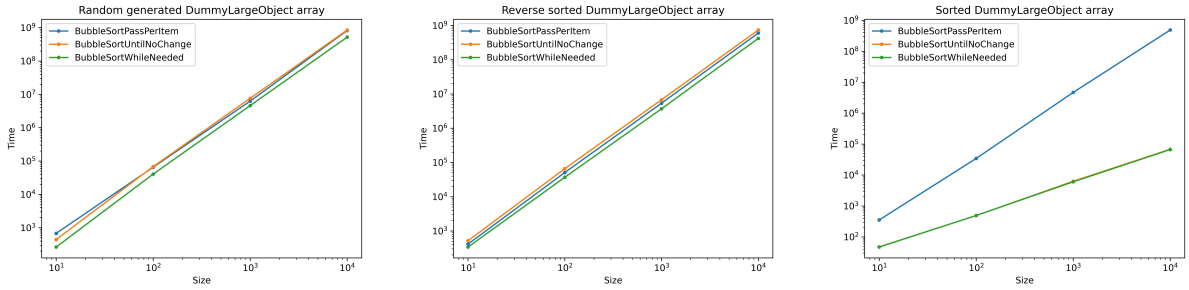

Figure 5: Data type: DummyBigObject

3.2 Descriptive Statistics

In this section we present some descriptive statistics of the results. For each group or condition we summarize the set of measured values with a "five-number summary": minimum, first quartile, median, third quartile, and maximum.

The minimum and the maximum values are are the smallest and the largest time in nanoseconds it took to sort the array using the given algorithm. The first quartile is the value that divides the first 25% of the data from the remaining 75%. The median is the value that divides the data in half. The third quartile is the value that divides the last 25% of the data from the remaining 75%.

In Table 4 we present the statistics results for the random generated arrays, while in Table 5 and Table 6 we present the statistics results for the same array sorted and reverse sorted respectively.

Type	Algorithm	Min	Max	Q1	Median	Q3
Integer	BubbleSortPassPerItem	$1.47 \cdot 10^3$	$3.16 \cdot 10^8$	$2.78 \cdot 10^4$	$1.85 \cdot 10^6$	$8.18 \cdot 10^7$
Integer	BubbleSortUntilNoChange	$9.39 \cdot 10^2$	$3.48 \cdot 10^8$	$2.82 \cdot 10^4$	$1.89 \cdot 10^6$	$8.99 \cdot 10^7$
Integer	BubbleSortWhileNeeded	$5.99 \cdot 10^2$	$2.39 \cdot 10^8$	$1.20 \cdot 10^4$	$1.31 \cdot 10^6$	$6.17 \cdot 10^7$
Double	BubbleSortPassPerItem	$4.44 \cdot 10^3$	$3.82 \cdot 10^8$	$2.31 \cdot 10^4$	$1.54 \cdot 10^6$	$9.77 \cdot 10^7$
Double	BubbleSortUntilNoChange	$3.38 \cdot 10^3$	$4.42 \cdot 10^8$	$2.68 \cdot 10^4$	$1.68 \cdot 10^6$	$1.13 \cdot 10^8$
Double	BubbleSortWhileNeeded	$2.26 \cdot 10^3$	$2.77 \cdot 10^8$	$1.59 \cdot 10^4$	$9.97 \cdot 10^5$	$7.07 \cdot 10^7$
String	BubbleSortPassPerItem	$1.28 \cdot 10^3$	$1.31 \cdot 10^9$	$5.14 \cdot 10^4$	$4.38 \cdot 10^6$	$3.34 \cdot 10^8$
String	BubbleSortUntilNoChange	$8.25 \cdot 10^2$	$1.25 \cdot 10^9$	$5.02 \cdot 10^4$	$4.72 \cdot 10^6$	$3.20 \cdot 10^8$
String	BubbleSortWhileNeeded	$7.24 \cdot 10^2$	$6.54 \cdot 10^8$	$3.16 \cdot 10^4$	$2.60 \cdot 10^6$	$1.67 \cdot 10^8$
SmallObject	BubbleSortPassPerItem	$6.11 \cdot 10^2$	$5.34 \cdot 10^8$	$4.31 \cdot 10^4$	$2.79 \cdot 10^6$	$1.38 \cdot 10^8$
SmallObject	BubbleSortUntilNoChange	$4.02 \cdot 10^2$	$6.83 \cdot 10^8$	$4.92 \cdot 10^4$	$3.52 \cdot 10^6$	$1.76 \cdot 10^8$
SmallObject	BubbleSortWhileNeeded	$2.17 \cdot 10^2$	$4.02 \cdot 10^8$	$2.86 \cdot 10^4$	$2.07 \cdot 10^6$	$1.04 \cdot 10^8$
LargeObject	BubbleSortPassPerItem	$6.77 \cdot 10^2$	$8.18 \cdot 10^8$	$4.91 \cdot 10^4$	$3.16 \cdot 10^6$	$2.09 \cdot 10^8$
LargeObject	BubbleSortUntilNoChange	$4.42 \cdot 10^2$	$8.47 \cdot 10^8$	$5.12 \cdot 10^4$	$3.83 \cdot 10^6$	$2.17 \cdot 10^8$
LargeObject	BubbleSortWhileNeeded	$2.63 \cdot 10^2$	$5.18 \cdot 10^8$	$3.05 \cdot 10^4$	$2.32 \cdot 10^6$	$1.33 \cdot 10^8$

Table 4: Random generated arrays

Type	Algorithm	Min	Max	Q1	Median	Q3
Integer	BubbleSortPassPerItem	$7.39 \cdot 10^2$	$1.72 \cdot 10^8$	$1.26 \cdot 10^4$	$1.13 \cdot 10^6$	$4.46 \cdot 10^7$
Integer	BubbleSortUntilNoChange	$8.33 \cdot 10^1$	$2.01 \cdot 10^4$	$1.80 \cdot 10^2$	$1.17 \cdot 10^3$	$6.62 \cdot 10^3$
Integer	BubbleSortWhileNeeded	$8.38 \cdot 10^1$	$2.13 \cdot 10^4$	$1.72 \cdot 10^2$	$6.23 \cdot 10^2$	$6.10 \cdot 10^3$
Double	BubbleSortPassPerItem	$3.31 \cdot 10^3$	$2.22 \cdot 10^8$	$1.70 \cdot 10^4$	$1.08 \cdot 10^6$	$5.72 \cdot 10^7$
Double	BubbleSortUntilNoChange	$2.92 \cdot 10^2$	$2.86 \cdot 10^4$	$3.29 \cdot 10^2$	$1.38 \cdot 10^3$	$8.97 \cdot 10^3$
Double	BubbleSortWhileNeeded	$3.16 \cdot 10^2$	$1.88 \cdot 10^4$	$3.57 \cdot 10^2$	$1.20 \cdot 10^3$	$6.23 \cdot 10^3$
String	BubbleSortPassPerItem	$8.17 \cdot 10^2$	$1.11 \cdot 10^9$	$4.06 \cdot 10^4$	$3.60 \cdot 10^6$	$2.83 \cdot 10^8$
String	BubbleSortUntilNoChange	$8.27 \cdot 10^1$	$1.18 \cdot 10^5$	$4.92 \cdot 10^2$	$4.03 \cdot 10^3$	$3.51 \cdot 10^4$
String	BubbleSortWhileNeeded	$1.37 \cdot 10^2$	$1.18 \cdot 10^5$	$4.53 \cdot 10^2$	$3.75 \cdot 10^3$	$3.48 \cdot 10^4$
SmallObject	BubbleSortPassPerItem	$3.43 \cdot 10^2$	$3.77 \cdot 10^8$	$3.28 \cdot 10^4$	$2.07 \cdot 10^6$	$9.73 \cdot 10^7$
SmallObject	BubbleSortUntilNoChange	$4.94 \cdot 10^1$	$5.77 \cdot 10^4$	$4.84 \cdot 10^2$	$3.22 \cdot 10^3$	$1.88 \cdot 10^4$
SmallObject	BubbleSortWhileNeeded	$6.56 \cdot 10^1$	$5.26 \cdot 10^4$	$4.48 \cdot 10^2$	$3.06 \cdot 10^3$	$1.73 \cdot 10^4$
LargeObject	BubbleSortPassPerItem	$3.52 \cdot 10^2$	$4.84 \cdot 10^8$	$2.58 \cdot 10^4$	$2.34 \cdot 10^6$	$1.24 \cdot 10^8$
LargeObject	BubbleSortUntilNoChange	$4.76 \cdot 10^1$	$6.77 \cdot 10^4$	$3.78 \cdot 10^2$	$3.38 \cdot 10^3$	$2.16 \cdot 10^4$
LargeObject	BubbleSortWhileNeeded	$4.70 \cdot 10^1$	$6.64 \cdot 10^4$	$3.80 \cdot 10^2$	$3.26 \cdot 10^3$	$2.11 \cdot 10^4$

Table 5: Sorted arrays

Type	Algorithm	Min	Max	Q1	Median	Q3
Integer	BubbleSortPassPerItem	$1.18 \cdot 10^3$	$3.07 \cdot 10^8$	$2.41 \cdot 10^4$	$1.63 \cdot 10^6$	$7.92 \cdot 10^7$
Integer	BubbleSortUntilNoChange	$1.17 \cdot 10^3$	$3.38 \cdot 10^8$	$2.61 \cdot 10^4$	$1.74 \cdot 10^6$	$8.70 \cdot 10^7$
Integer	BubbleSortWhileNeeded	$8.20 \cdot 10^2$	$2.27 \cdot 10^8$	$1.68 \cdot 10^4$	$1.19 \cdot 10^6$	$5.86 \cdot 10^7$
Double	BubbleSortPassPerItem	$4.10 \cdot 10^3$	$3.76 \cdot 10^8$	$2.78 \cdot 10^4$	$1.85 \cdot 10^6$	$9.67 \cdot 10^7$
Double	BubbleSortUntilNoChange	$2.59 \cdot 10^3$	$4.00 \cdot 10^8$	$3.11 \cdot 10^4$	$1.92 \cdot 10^6$	$1.03 \cdot 10^8$
Double	BubbleSortWhileNeeded	$1.26 \cdot 10^3$	$2.41 \cdot 10^8$	$1.96 \cdot 10^4$	$1.21 \cdot 10^6$	$6.20 \cdot 10^7$
String	BubbleSortPassPerItem	$1.32 \cdot 10^3$	$9.03 \cdot 10^8$	$5.14 \cdot 10^4$	$3.80 \cdot 10^6$	$2.31 \cdot 10^8$
String	BubbleSortUntilNoChange	$1.29 \cdot 10^3$	$9.45 \cdot 10^8$	$5.55 \cdot 10^4$	$3.80 \cdot 10^6$	$2.42 \cdot 10^8$
String	BubbleSortWhileNeeded	$7.65 \cdot 10^2$	$4.35 \cdot 10^8$	$2.95 \cdot 10^4$	$1.98 \cdot 10^6$	$1.12 \cdot 10^8$
SmallObject	BubbleSortPassPerItem	$4.03 \cdot 10^2$	$4.95 \cdot 10^8$	$4.08 \cdot 10^4$	$2.52 \cdot 10^6$	$1.27 \cdot 10^8$
SmallObject	BubbleSortUntilNoChange	$5.74 \cdot 10^2$	$6.53 \cdot 10^8$	$4.95 \cdot 10^4$	$3.29 \cdot 10^6$	$1.68 \cdot 10^8$
SmallObject	BubbleSortWhileNeeded	$3.99 \cdot 10^2$	$3.57 \cdot 10^8$	$2.75 \cdot 10^4$	$1.81 \cdot 10^6$	$9.19 \cdot 10^7$
LargeObject	BubbleSortPassPerItem	$4.13 \cdot 10^2$	$6.00 \cdot 10^8$	$3.82 \cdot 10^4$	$2.72 \cdot 10^6$	$1.54 \cdot 10^8$
LargeObject	BubbleSortUntilNoChange	$5.15 \cdot 10^2$	$7.35 \cdot 10^8$	$4.95 \cdot 10^4$	$3.43 \cdot 10^6$	$1.89 \cdot 10^8$
LargeObject	BubbleSortWhileNeeded	$3.38 \cdot 10^2$	$4.20 \cdot 10^8$	$2.77 \cdot 10^4$	$1.86 \cdot 10^6$	$1.08 \cdot 10^8$

Table 6: Reverse sorted arrays

4 Discussion

4.1 Compare Hypothesis to Results

In the previous section we have shown the results obtained from the experiment. In this section, we will discuss the results and compare them to the hypothesis.
Looking at the plots in subsection 3.1, we can see that for the randomly generated and reverse sorted arrays. All the algorithms performs slightly the same, with the `BubbleSortWhileNeeded` being just a little faster than the other two every time for every data-type. The `BubbleSortPassPerItem` and the `BubbleSortUntilNoChange` are almost the same, with the `BubbleSortUntilNoChange` being slightly faster when it comes to small lists and a little slower when it comes to very large lists.
When it comes to the already sorted arrays, the `BubbleSortPassPerItem` is clearly the worst: while the other algorithms are able to understand that the array is already sorted, the `BubbleSortPassPerItem` is not able to do that and it has to re-sort the array as it wasn't sorted. The other two algorithms performs almost the same, with the `BubbleSortUntilNoChange` being slightly faster than the `BubbleSortWhileNeeded` for small lists.
We run the algorithms on different array sizes of 10 to the power of up to 4. Looking at the plots we noticed that the latest deviations in performance occur between sizes 10^2 and 10^3. However, after size 10^3 we do not see any algorithm outperforming another algorithm and the ranking of the performance of algorithms at point 10^3 is the same as at point 10^4. With this we can draw a conclusion that every algorithm executed on array bigger than 10^3 in size will have a constant growth compared to the other algorithms and no changes in ranking will be present. Moreover, this can be considered just as another hypothesis and basis for an another experiment. What we claimed here is based on our results and is not proven.
Looking at the Tables 4, 5 and 6 the data type of the array doesn't seem to have any effect on the relative performance of the algorithms. They clearly spend more time sorting the `String` arrays, but the relative performance of the algorithms is always very similar. This means that the algorithms just move pointers to the objects, and they don't have to copy the objects themselves. Obviously the only things that can impact the performance is the comparison to determine if one element is greater than the other, which is handled java itself (comparing strings is notoriously slower then integers).
The initial H_0 hypothesis was that all the algorithms will perform the same, but we have seen that this is not the case. We can now reject the hypothesis and give a precise ranking of the algorithms in terms of performance given our variables and measurements.

4.2 Limitations and Threats to Validity

- Time threat with measurement of 5 hours: machine can get better or worse depending on battery level, CPU activity, etc.
- Limiting experiment to one machine, operating system, terminal or IDE: some algorithms might perform better on different machines, operating systems, terminals or IDEs due to different hardware, software, etc.
- Length of strings: Some algorithms might perform better or worse depending the length of the string.
- Warm-up iteration: We assume that 100 cycles of warm-up are enough for the machine to instantiate and reduce the bias as much as possible. In reality, we are not sure if that is the case.

4.3 Conclusions

After carrying out all the calculations and analyzing the numbers and plots, we can see that the algorithm `BubbleSortWhileNeeded` performs the best given our measurements. The algorithms `BubbleSortPassPerItem` has clearly the worst performance and `BubbleSortUntilNoChange` is in the middle. The results obtained gave us a clear picture of the whole experiment and we were able to carry out the entire procedure in an organized and successful manner.

4.4 Looking back at the sorting algorithms

The three sorting algorithms that need to be analyzed have different implementations. It is now interesting to see why we get this results in the experiment.
A short description and a code snippet of each one of them is present below as a short introduction in order to be able to understand and reason about the results better at the end of the experiment.
All the three algorithms are variations of the well-known `BubbleSort` algorithm.
The first one, `BubbleSortPassPerItem`, is the most basic implementation of the algorithm. It is a simple algorithm that iterates over the array and compares each element with all the next ones. If an element is bigger than the next one, the two elements are swapped. This process is repeated until the array is sorted.

```
1  public final class BubbleSortPassPerItem<T extends Comparable<T>> implements Sorter<T> {
2      public void sort(final T[] items) {
3          for (int pass = 0; pass < items.length; pass++) {
4              for (int i = 0; i < items.length - 1; i++) {
5                  if (items[i].compareTo(items[i + 1]) > 0) {
6                      final T item = items[i];
7                      items[i] = items[i + 1];
8                      items[i + 1] = item;
9                  }
10             }
11         }
12     }
13 }
```

The second algorithm, `BubbleSortWhileNeeded`, is a variation of the first one. The array to be sorted is iterated several times and elements are swapped if needed. At every iteration we keep track of how many elements have been swapped. If no elements have been swapped, the array is sorted and the algorithm stops.

```
1  public final class BubbleSortWhileNeeded<T extends Comparable<T>> implements Sorter<T> {
2      public void sort(final T[] items) {
3          int n = items.length;
4          do {
5              int maxIndex = 0;
6              for (int i = 1; i < n; i++) {
7                  if (items[i - 1].compareTo(items[i]) > 0) {
8                      final T item = items[i - 1];
```

```
                    items[i - 1] = items[i];
                    items[i] = item;
                    maxIndex = i;
                }
            }
            n = maxIndex;
        } while (n > 0);
    }
}
```

The third algorithm, `BubbleSortUntilNoChange` is another variation of the bubble sort algorithm. It is very similar to the second one, but instead of keeping track of the number of swaps, it keeps track with a boolean value if some elements have been swapped or not. If no elements have been swapped, the array is sorted and the algorithm stops.

```
public final class BubbleSortUntilNoChange<T extends Comparable<T>> implements Sorter<T> {
    public void sort(final T[] items) {
        boolean changed;
        do {
            changed = false;
            for (int i = 0; i < items.length - 1; i++) {
                if (items[i].compareTo(items[i + 1]) > 0) {
                        final T item = items[i];
                        items[i] = items[i + 1];
                        items[i + 1] = item;
                        changed = true;
                    }
                }
        } while (changed);
    }
}
```

Appendix

4.5 Materials

[1] How to Design and Report Experiments: Andy; Hole, Graham J. Field
[2] Slides on iCorsi
[3] Plotly python graphing library
[4] JDK 18 Documentation

4.6 Reproduction Package (or: Raw Data)

The reproduction package is included and can be found in the following GitHub repository:
https://github.com/Alessandro-Gobbetti/Experimentation-and-Evaluation
In the repository, there is also the `csv` file that contains all raw data of the measurements named `results.csv`.