Authors: Michele Dalle Rive, Loredana De Filippo

Bubble Inc. Benchmarking

Experiment 1, Experimentation & Evaluation 2023

Abstract

In this comparative study, we examined three variations of bubble sort algorithms - *BubbleSortPassPerItem*, *BubbleSortWhileNeeded*, and *BubbleSortUntilNoChange* - across different input parameters such as sizes, types, and orders. Our goal was to determine the most effective algorithm across different scenarios. The experiment was carried out using a comprehensive multi-factor design with a full factorial configuration. The findings consistently demonstrated that *BubbleSortWhileNeeded* outperformed the other algorithms. Our hypotheses regarding execution time in specific contexts, the independence of list type, and the time complexity of bubble sort ($O(n^2)$) were confirmed. Despite constraints related to array size and system limitations, this analysis yields valuable insights into the performance of the bubble sort algorithm and its variations.

1. Introduction

Sorting algorithms play a fundamental role in the software we use on a daily basis: from operating system kernels to Google search results ranking.
Starting from naive approaches, computer scientists have spent decades analysing and comparing sorting algorithms (Mishra & Garg, 2008), trying to discover faster and more efficient ones.
The research is still widely open [1], focusing on finding the fastest possible algorithm.
When it comes to particular circumstances, however, the fastest is not necessarily better: in systems with extremely low memory availability, using algorithms like merge sort or quicksort might not be possible at all, and the choice of in-place algorithms is forced. In the extreme case of memory size of at most two elements, bubble sort might be the only option.
For this reason, optimizations and improvements are being developed to this day [2, 3]

This experiment aims at comparing and analysing three different implementations of bubble sort: *BubbleSortPassPerItem*, *BubbleSortWhileNeeded,* and *BubbleSortUntilNoChange*. The goal of this analysis is to find out which algorithm performs the best overall.

The hypotheses of this experiment are formulated based on the source code provided for each algorithm.

Hypotheses:
In the case of a sorted list, no matter the size or type of element in the list, BubbleSortPassPerItem will result in a longer execution time compared to the other algorithms. With a fixed list size, the execution time of the different algorithms is not affected by the

type and size of the elements contained in the list.

Independently of the settings, *BubbleSortWhileNeeded* will always result in a lower average execution time compared to the other algorithms.

The time complexity of any bubble sort algorithm is always $O(n^2)$, i.e. if the input size increases of a factor n, the sorting time increases of a factor n^2.

2. Method

2.1 Variables

Independent variable	Levels
Algorithm	BubbleSortWhileNeeded; BubbleSortUntilNoChange; BubbleSortPassPerItem
Input size	500, 1000, 5000, 10000, 50000, 100000
Input type	“Primitive” (Integer); Object (String)
Input order	Random order; Already sorted; Reversed sorted order

When the *String* input type is used, all the strings have a fixed width of 4 characters, in order to keep the comparison and swap operations constant in cost. However, this does not actually happen: string comparison is linear in the length of the shortest string, but stops as soon as a different character is found; for this reason, as the array approaches being sorted, the comparison of the strings will be cheaper and cheaper, while the swap operation stays constant in time.

Dependent variable	Measurement Scale
Execution time ratio	Nanoseconds per element

Control variable	Fixed Value
Machine workload	No user space threads in the experiment core

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

We designed a multi-factor experiment with a full factorial design, i.e. measuring the outcome of each combination of independent variables; it's not a truly full factorial design, because we measured the String with every other combination of IV, but we excluded sizes 50,000 and 100,000 because of time constraints.

Benchmarking is a complex task, since it is not possible to keep all the confounding variables completely under control: operating system periodic tasks, memory-induced delays, and JVM garbage collection are just some of the difficulties that can be encountered while measuring performances on the millisecond scale.
For this reason, we preferred having no assumption and measuring how the performance is affected by each change in the IVs we chose. Although we suspect the *type* independent variable, for example, has no effect on the final result.

Our design includes repeated measurements within the same run, in order to balance out the delay entities explained earlier. The specifications of the measurements will be provided in the next sections.
An important detail is that our measurements are repeated a different number of times based on the size of the array. Although such an approach could act as a confounding factor, by creating data much more precise for smaller arrays than bigger ones, it is dictated by the constraints and by the environment: big arrays should be taken into account in this analysis, but it is not possible to sort them hundreds of times; small arrays, on the other hand, can be highly susceptible to the machine's confounding factors, and should be measured more, in order to reduce the impact of such delays. Having an "adaptive" number of measurements should actually make more precise results than having a fixed number.
However, it must be said that we don't sample the big arrays (>= 50000 elements) enough to be statistically accurate: around 30 measurements are required [4], but we only compute 20.

2.3 Apparatus and Materials

Hardware: Dell Precision 5550 (CPU: Intel(R) Core(TM) i7-10850H CPU @ 2.70GHz). All measurements were executed on a single isolated core [5].
JVM: openjdk 17.0.8.1 2023-08-24. No java parameters or
Time measurement: Computed as an interval using System.nanoTime (https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/System.html#nanoTime())

2.4 Procedure

A warm-up phase, consisting of the sorting of 10000 elements, is first executed to get the JVM started for more stable results.

After that, the independent variables are iterated as shown in the diagram in section 2.2, in the order specified in section 2.1. For each combination of array size and array initial ordering, a run is executed.

Each run consists of multiple measurements with the same settings.
Each measurement consists of:

- Array generation; the array is re-generated every time, in order to always have a different one in the random array case. Note on the random array: it does not contain *random* values, but it consists of a sorted array that is *randomly* shuffled.
- Measurement of the start time in nanoseconds.
- Execution of the sorting algorithm on the array.
- Measurement of the end time in nanoseconds.
- Computation of the *delta*, i.e. the time taken by the algorithm to execute.

The delta of each measurement is stored in order to then be analysed.

The number of measurements in each run, because of time constraints, is different: 2000 measurements if the size is less or equal to 1000, 200 measurements if the size is less or equal to 10000, and 20 measurements for bigger arrays.

A new experiment is then executed, using String instead of Integer.

3. Results

All the data visualised, unless specified, refers to the sorting of arrays containing Integer values.

3.1 Visual Overview

Average execution time

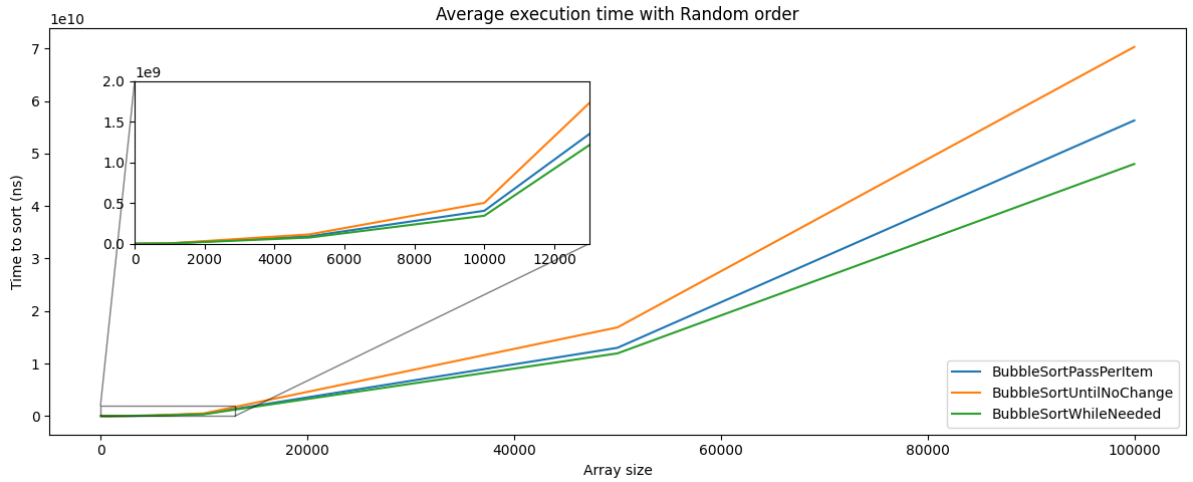

Figure 1: Average execution time.

At a quick glance, the *BubbleSortWhileNeeded* algorithm looks the best overall, with it being constantly below every other sorting algorithm. In contrast, *BubbleSortUntilNoChange* is constantly above the others, implying that its sorting time, on average, is the slowest for a random array.

Average sorting time per order

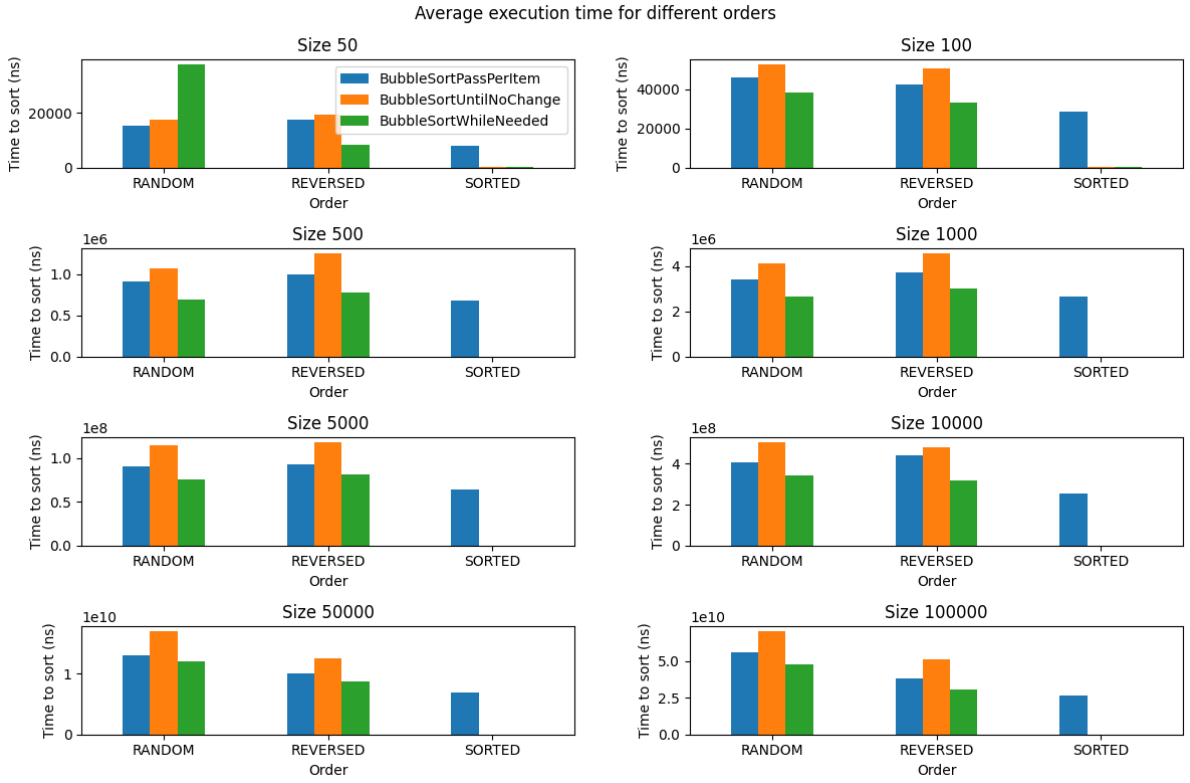

Figure 2: Average sorting time per order with Integer type.

The chart displays the array sizes tested for each algorithm based on their order. An obvious observation is that BubbleSortPassPerItem is the sole bar visible in the sorted section. Aside from the case of array size 50, it is evident that BubbleSortUntilNoChange consistently proves to be the longest average sorting time in both the random and reversed orders.
We noticed that some runs have huge outliers (up to 10x the average), which are particularly present in measurements with small array sizes (< 1000). We are not sure about the reason, but we think it might be linked to TLB/cache misses or delays added by garbage collection. The result is more visible for small arrays because they are faster to sort, and such a delay has a bigger impact.

Locality times

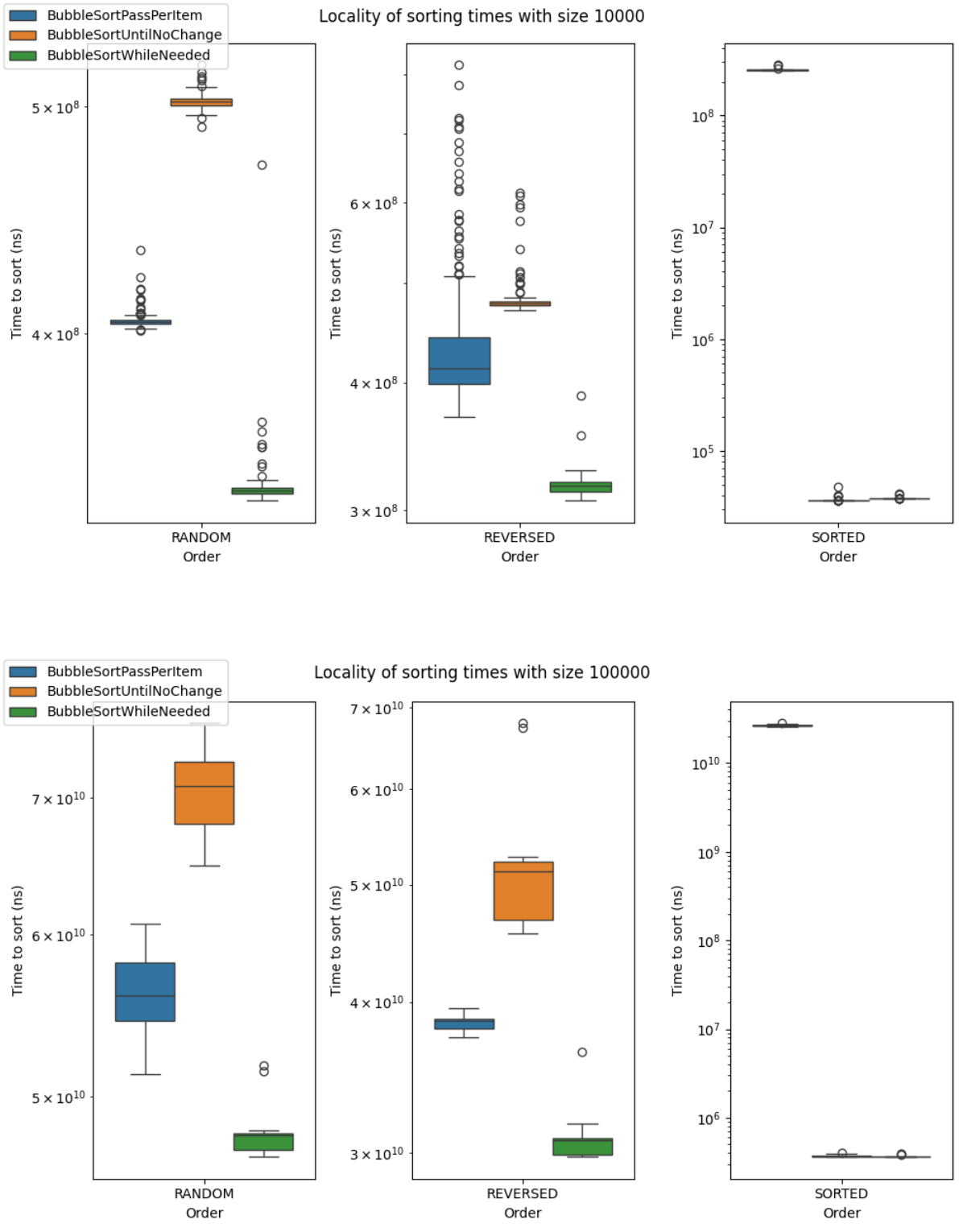

Figure 3: Locality times.

This box-and-whisker plot illustrates that regardless of size, the distribution positions of data for each algorithm remain consistent. The sorted order displays almost identical patterns across the sizes.

Cumulative distribution

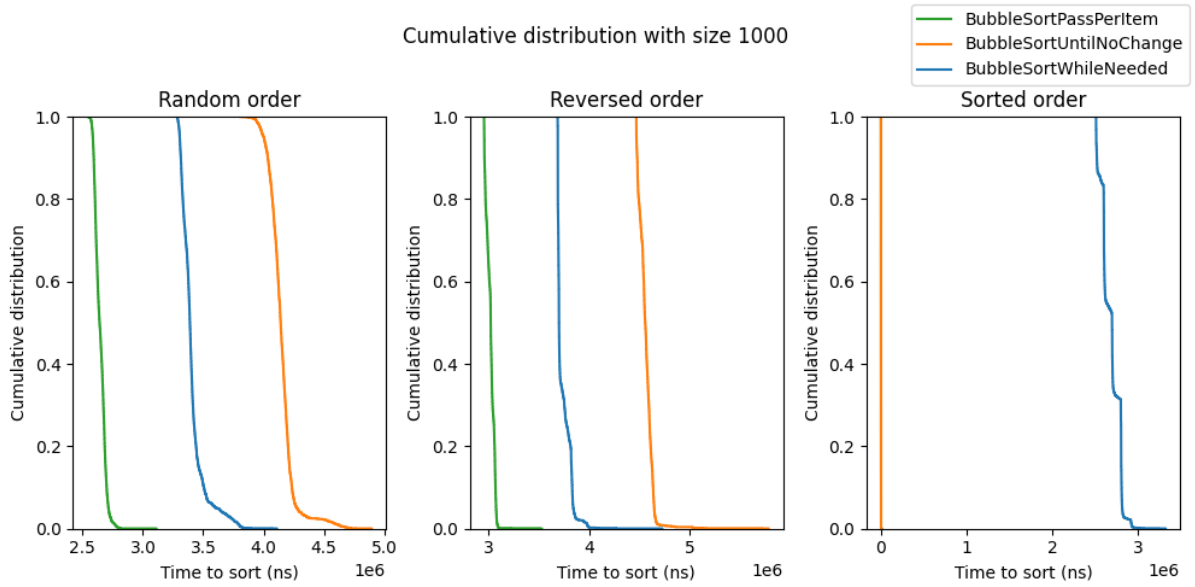

Figure 4: Cumulative distribution.

This cumulative distribution emphasises how similar the random and reversed order in the spacing of each algorithm displayed on the graph compared to the sorted order: *BubbleSortUntilNoChange* and *BubbleSortWhileNeeded* overlap, while *BubbleSortPassPerItem* stands out noticeably.
Also, it can be seen that every sorting algorithm has, generally speaking, some outliers.

Type of element

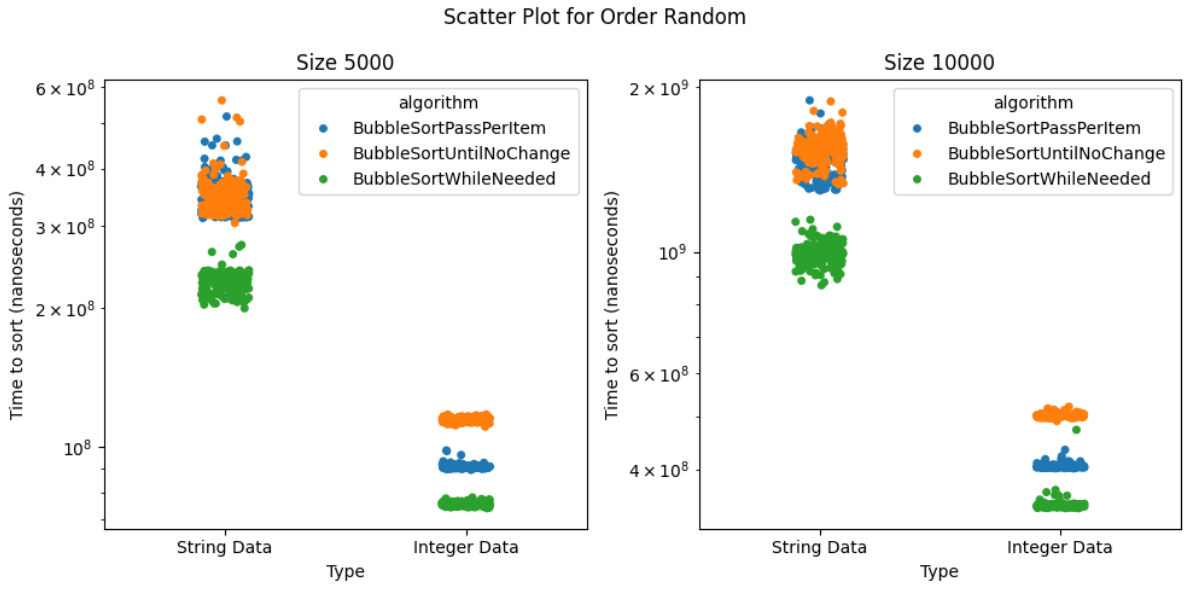

Figure 5: Type of element.

The scatter plot displays the contrast in sorting between string and integer elements. All algorithms take more time to sort random string elements due to their complexity with comparison. Additionally, BubbleSortPassPerItem and BubbleSortUntilNoChange largely overlap in time when sorting strings, whereas, in the case of integers, there's a distinct difference, with BubbleSortUntilNoChange taking longer.
A plot similar to the one in Figure 2 with type string is available on Google Colab (see the link at the end). We noticed a slight difference in the ranking with Strings: *BubbleSortWhileNeeded* is by far the best, while the other ones are very close. The reason could be because of the comparison and swap functions of bigger objects, which are far more complex than a simple Integer comparison and swap. However, we couldn't completely explain this behaviour.

3.2 Descriptive Statistics

Because of the excessive number of combinations, not all results can be displayed in a table. The following table shows statistics about the most relevant runs with Integer type and random order since it best describes the actual sort time of the algorithms.

Size	Algorithm	Minimum	Q1	Median	Q3	Maximum
1K	BSPassPerItem	3.2769e+06	3.3413e+06	3.3895e+06	3.4251e+06	4.1105e+06
1K	BSUntilNoChange	3.7995e+06	4.0855e+06	4.1408e+06	4.1907e+06	4.8953e+06
1K	BSWhileNeeded	2.5383e+06	2.6087e+06	2.6418e+06	2.6795e+06	3.1125e+06
10K	BSPassPerItem	4.0143e+08	4.0397e+08	4.0458e+08	4.0538e+08	4.3426e+08
10K	BSUntilNoChange	4.9010e+08	5.0038e+08	5.0230e+08	5.0418e+08	5.2082e+08
10K	BSWhileNeeded	3.3956e+08	3.4197e+08	3.4276e+08	3.4390e+08	4.7240e+08
100K	BSPassPerItem	5.1297e+10	5.4468e+10	5.6018e+10	5.8144e+10	6.0748e+10
100K	BSUntilNoChange	6.4889e+10	6.7981e+10	7.0944e+10	7.2871e+10	7.6134e+10
100K	BSWhileNeeded	4.6723e+10	4.7089e+10	4.7880e+10	4.7996e+10	5.1814e+10

The minimum and the maximum represent the full range of the measured data, and they are usually outliers.
The first and third quartiles (Q1 and Q3) describe where 50% of data is contained: it gives an overview of how to spread the most dense part of the sample.
The median, in the end, represents the middle point: it's the measurement higher and lower than 50% of the sample.
This table, which is also partially visualised in Figure **add number**, can give interesting insights about how spread out the data is, and around which values it is focused.

4. Discussion

4.1 Compare Hypothesis to Results

Hypothesis 1

“In the case of a sorted list, no matter the size or type of element in the list, BubbleSortPassPerItem will result in a longer execution time compared to the other algorithms.”

Our findings show that when dealing with a sorted list, BubbleSortPassPerItem consistently took much more time compared to the other sorting methods, regardless of any other factors. This surely originates from the difference in the “stop conditions” of the three algorithms: *BubbleSortPassPerItem* always iterates over each element *n* times, where *n* is the size of the input array; on the other hand, both the other algorithms stop earlier, making them much faster.
This, therefore, supports the hypothesis that BubbleSortPassPerItem is indeed slower in such scenarios.

Hypothesis 2

“With a fixed list size, the execution time of the different algorithms is not affected by the type and size of the elements contained in the list.”

help

Hypothesis 3

“Independently of the settings, *BubbleSortWhileNeeded* will always result in a lower average execution time compared to the other algorithms.”

Our benchmarks revealed that *BubbleSortWhileNeeded* is often the best performer, sometimes by a large factor. Although this is not true, as is always the case for very small arrays (< 500), we blame all the system confounding factors for this result: it can be noted that, independently of the type, length, and order, *BubbleSortWhileNeeded* can be considered the best overall performer.
A reason that could explain the better performance of *BubbleSortUntilNoChange* is the fact that the boolean flag used in the latter can be more efficient than the index used by *BubbleSortWhileNeeded* if the array is very small and already sorted.
Based on our data, however, this hypothesis is to be considered false as, in particular cases of small arrays already sorted, *BubbleSortUntilNoChange* performs better.

Hypothesis 4

“The time complexity of bubble sort is always $O(n^2)$, i.e. if the input size increases of a factor n, the sorting time increases of a factor n^2.”

As shown in Figure 1 and Chapter 3.2, the sorting time increases of a quadratic factor w.r.t the input size. For example, in the table, it can be seen that, as the size increases by a factor of 10, the execution time increases by a factor of ~100. The result is not precise, since bigger arrays have an intrinsic bigger computational overhead, but the order of magnitude of the change is the expected one. Since this happens for all the algorithms, in all cases as

shown in Figure 2, we can consider our hypothesis verified.

4.2 Limitations and Threats to Validity

A relevant limitation of our experiment is the reduced size of the array we considered; because of the time complexity of the algorithms, running a benchmark for an array of 1M elements with our current settings (3 different algorithms, 3 different array orders, 20 measurements per run) would take ~15 days to complete. However, we don't consider this limitation a threat to the validity of our study, since it is unlikely for a user of bubble sort to try to sort an array of such size.

Another limitation is the system used. As already noted in Section 3.1, there are a few big outliers that we are not completely able to explain. Using a system with a bigger page table, and probably tweaking the configuration of the JVM, could help reduce them.

The biggest limitation, which has a relevant impact on the final results, is probably the choice of the language; a job like benchmarking, where the difference is in the nanoseconds/milliseconds scale, working in a system that you have full control over can make a huge difference.
Improvements to this analysis can be made by using a language that gives full control to the user, like C, C++, or Rust. Doing so:

- Eliminates any garbage-collector-induced delay
- Allows for lower-level execution (i.e. delays can be computed with CPU intrinsics)
- Allows for better control over possible system-level delays: the program can be profiled and accurate results about TLB misses and cache misses can be retrieved.
- Provides overall better control over the machine instructions that are executed; for example, optimizations can be completely removed to have the compiler produce exactly the code wanted.

4.3 Conclusions

After benchmarking three comparable sorting algorithms, we found out that 3 out of our 4 hypotheses are true. In particular, the results highlight that, even if the time complexity for all the algorithms is the same, *BubbleSortWhileNeeded* has the lowest execution time in all array sizes, array orders, and array element types, making it the top overall performer.

Appendix

A. Materials

1. Mankowitz, D.J., Michi, A., Zhernov, A. et al. Faster sorting algorithms discovered using deep reinforcement learning. Nature 618, 257–263 (2023).
2. T. Paul, "Enhancement of Bubble and Insertion Sort Algorithm Using Block Partitioning," 2022 25th International Conference on Computer and Information Technology (ICCIT), Cox's Bazar, Bangladesh, 2022, pp. 412-417, doi: 10.1109/ICCIT57492.2022.10055404.

3. N. Faujdar and S. P. Ghrera, "A practical approach of GPU bubble sort with CUDA hardware," 2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence, Noida, India, 2017, pp. 7-12, doi: 10.1109/CONFLUENCE.2017.7943115.
4. Kwak SG, Kim JH. Central limit theorem: the cornerstone of modern statistics. Korean J Anesthesiol. 2017;70(2):144-156. doi:10.4097/kjae.2017.70.2.144
5. Linuxtopia. Linux Kernel Configuration - isolcpus. https://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/re46.html.

- Andy Field, Graham Hole (2003). *How to Design and Report Experiments*. SAGE Publications Ltd

B. Reproduction Package (or: Raw Data)

CSV data with the benchmark data is attached to the report.

Google Colab we used to analyse the data: EE1 - Analysis (https://colab.research.google.com/drive/15_UbEZs9ibwpsJ5rDVgTHD8VlSuFqfSD?usp=sharing)

Repository with the code we used to benchmark the algorithms: https://github.com/micheledallerive/sorting-algos-benchmark