Performance Measurement for Bubble Inc

By Lavrov, Ioffe, Biletskyi

Experiment 1, Experimentation & Evaluation 2024

Abstract

The report presents a performance measurement study conducted by Bubble Inc. The task is to evaluate which of the provided sorting algorithms works the fastest. The main criterion is execution time in milliseconds.

In the following experiment report you will find the results of testing 4 sorting algorithms:

- BubbleSortUntilNoChange
- BubbleSortWhileNeeded
- QuickSortGPT
- SelectionSortGPT

6 different data types, 3 types of structurization, 3 input dimensions, and 2 triples.

The results include:

- Comparative results of all algorithms included
- Analysis of testing environment
- Analysis of forces that can affect the results
- Situational analysis of used algorithms

1. Introduction

This experiment is targeted towards identifying the best sorting algorithm among 4 algorithms that participate in this experiment. The algorithms will be tested on simple data types that are present for each sorted object:

- Data types presented in experiment are int, long, float, double, char, string
- The data object will always hold a non-null value

[redacted]
The QuickSortGPT algorithm is expected to perform better than others, due to its average time complexity being O(nlogn) in comparison to average time complexity of O(n^2) of other presented algorithms. Experiment assumes that testing in different environments can yield highly different result times, but the difference will be due to available processing powers of environments and tendencies of algorithm performance will be the same.
The HP Omen, due to its higher raw processing power and dedicated GPU, will execute sorting algorithms faster than the MacBook Pro 14” with an M2 Pro chip. The HP Omen’s hardware specifications suggest greater computational power, especially in tasks that could leverage multi-core performance or GPU acceleration. Therefore, it is expected to outperform the MacBook Pro 14” in algorithm execution time, particularly for larger datasets with higher processing demands.

2. Method

In this experiment, the testing was performed on 2 different machines:

- HP Omen 14 - Intel Core Ultra 9 185H processor, 32 GB RAM
- Macbook pro 14 - Apple M2 processor, 16 GB RAM

It is assumed that not 100% of processing power was available for a machine as it was running a corresponding OS and it together with necessary background processes could have consumed part of the available resources (not more than 25%). The code for the experiment and IDE versions will be available in Appendix Section B.

It should also be noted, used machines are assumed to have more processing power than it is needed to perform the testing in a scale used for experiment, therefore their performance will be limited not by processing powers, but by speed with which information can travel inside a machine.

2.1 Variables

[redacted]	
Type of sorting algorithms	BubbleSortUntilNoChange, BubbleSortWhileNeeded, QuickSortGPT, SelectionSortGPT
Number of elements in tested array	100, 1 000, 10 000
Data type	Int, long, float, double, char, string
Structure	Best-case (sorted), average-case (partially sorted), worst-case (reverse sorted)

[redacted]	
Execution time	Execution time is measured in milliseconds and represents time it took for identified machine to complete sorting

[redacted]	
Processing power	HP Omen 14: 32 GB RAM, 5.1 GHz processor, 16 cores / 22 threads Macbook 14 pro: 16 GB RAM, 3.2 GHz processor, 8 cores / 8 threads
Software environment	HP Omen 14: Java JDK 21, Intellij Idea, Windows OS Macbook 14 pro: Java JDK 21, Intellij Idea, Mac OS
Execution conditions	Minimal background processes and unlimited resource usage for algorithms

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**

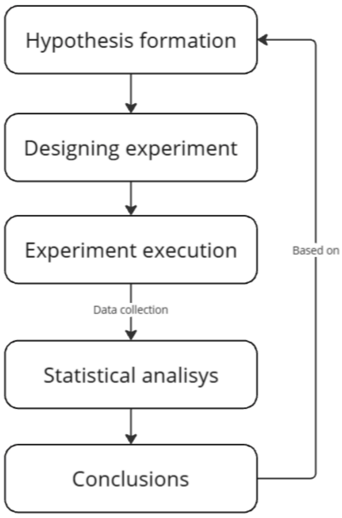
During the experiment, many factors are varied: sorting algorithms, data types, quantities, scenarios and even devices, this allows us to get a better idea of how their individual and combined influence can affect performance.

The main objective is to compare the performance of algorithms, it is necessary to show and analyze the efficiency and consistency under different conditions.

The experiment should be conducted under controlled conditions, with the same data types, input data sizes and scenarios ensuring that the results are due to the algorithms and not to external factors. Multiple trials are conducted for each combination of factors to account for variability and ensure the validity of the results.

The experiment performed is a controlled experiment in a fully controlled environment.

The process can be illustrated on the image below:

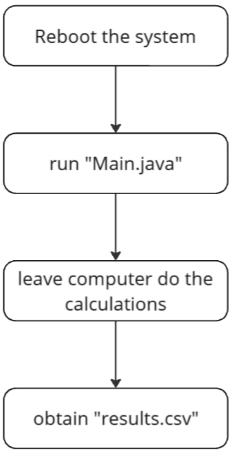


The experiment is prepared and can be re-run by both researchers and outsiders with the necessary inputs, which will be specified at the end of the report.

2.3 Apparatus and Materials

	Experiment 1 (Windows)	Experiment 2 (MacOs)
Computer and specifications	HP Omen 14, Ultra 9 185H processor and 32GB RAM	Macbook Pro 14 with an M2 processor and 16GB RAM.
Software	Java JDK 21, Intelij Idea	Java JDK 21, Intelij Idea
Time tracking	System.nanoTime()	System.nanoTime()

2.4 Procedure



The experiment was automated, with no manual restarts or system resets required.

For each experimental configuration, the procedure was as follows:

1. **Initialize Program**:
    - The program was loaded in the IDE with all required classes.
2. **Execute Main Program**:
    - The main file (`Main.java`) was executed, setting up the parameters for the current configuration.
3. **Array Generation and Sorting**:
    - An array was generated based on the configuration.
    - The sorting algorithm processed the array, and the time taken was recorded.
4. **Data Collection**:
    - Sorting times and configuration details for each trial were saved to a CSV file.
5. **Configuration Update**:
    - After each configuration, parameters were automatically updated for the next run.

**The script skips the first 10 iterations, treating them as a warm-up phase, and does not record their results.*

3. Results

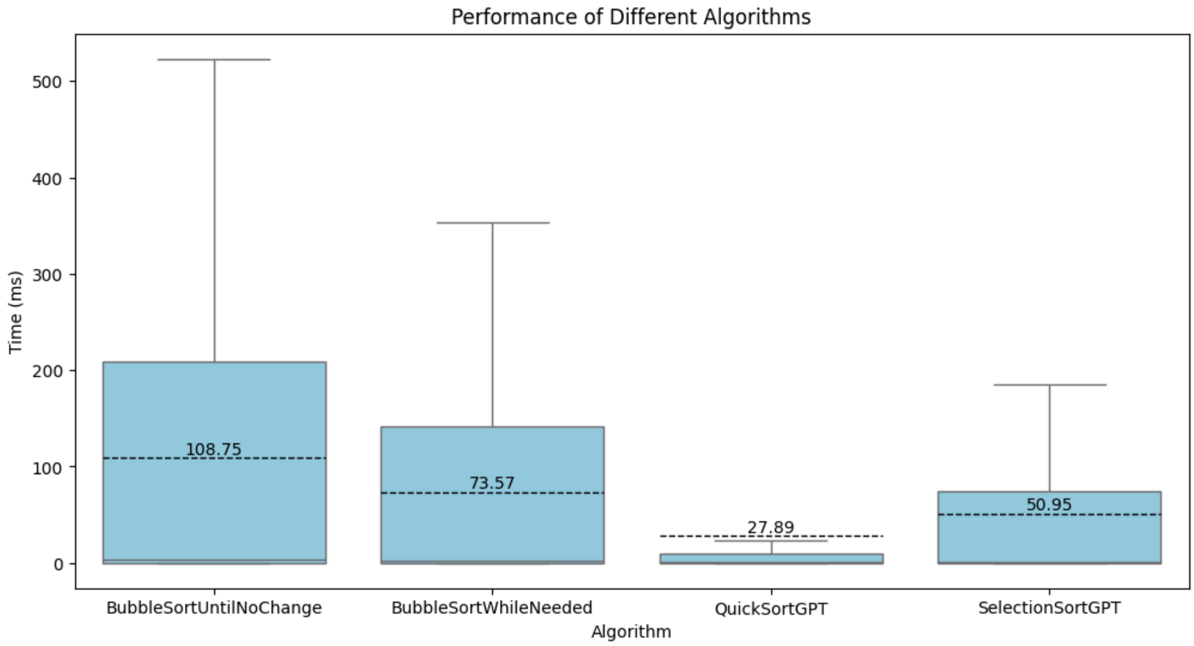
3.1 Visual Overview

The graphs obtained during the analysis of the experiment are presented here, without detailed data. For conclusions on each graph, see the file `analysis.ipynb`.

Starting from more understandable and general data, the results have been summarized and presented in the following graphs.

3.1.1 Robust comparison of algorithms

Plot of a visual representation of the execution time distribution for the sorting algorithms specified in the conditions.



Labels:

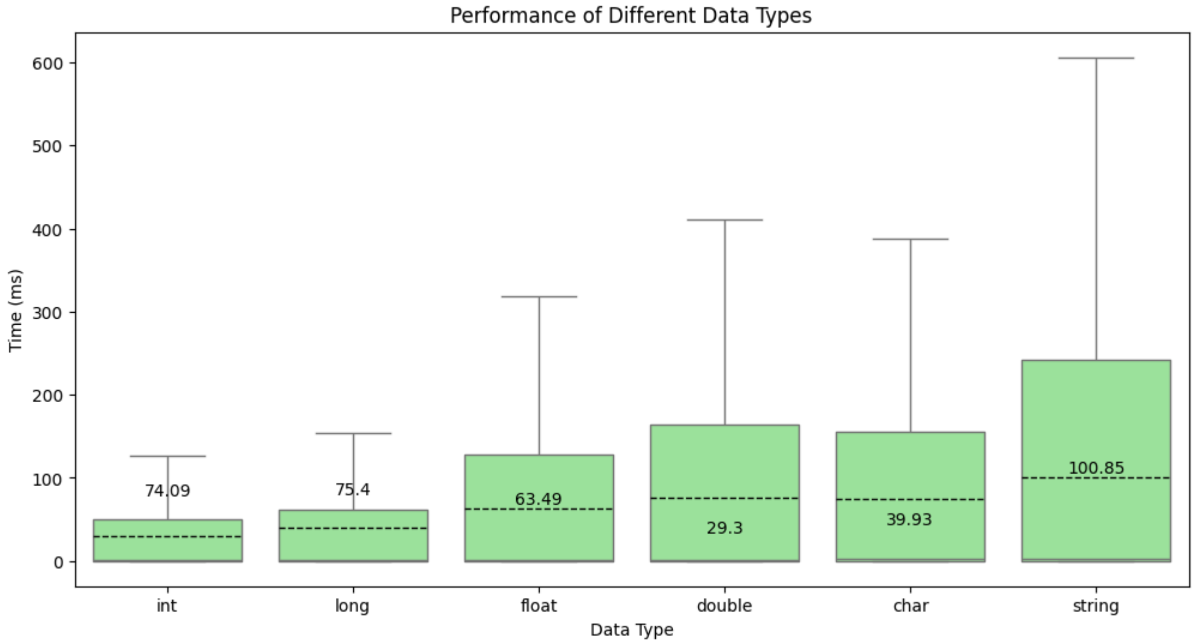
The horizontal line in each cell represents the median run time, indicating the mean of the data.

The box itself represents the interquartile range (IQR), which contains the 50th percentile of the data. The bottom and top edges of the box correspond to the 25th and 75th.

The whiskers extend from the box to the minimum and maximum values within a certain range.

The graphical diagram clearly emphasizes the superiority of QuickSortGPT and SelectionSortGPT over the BubbleSort variants.

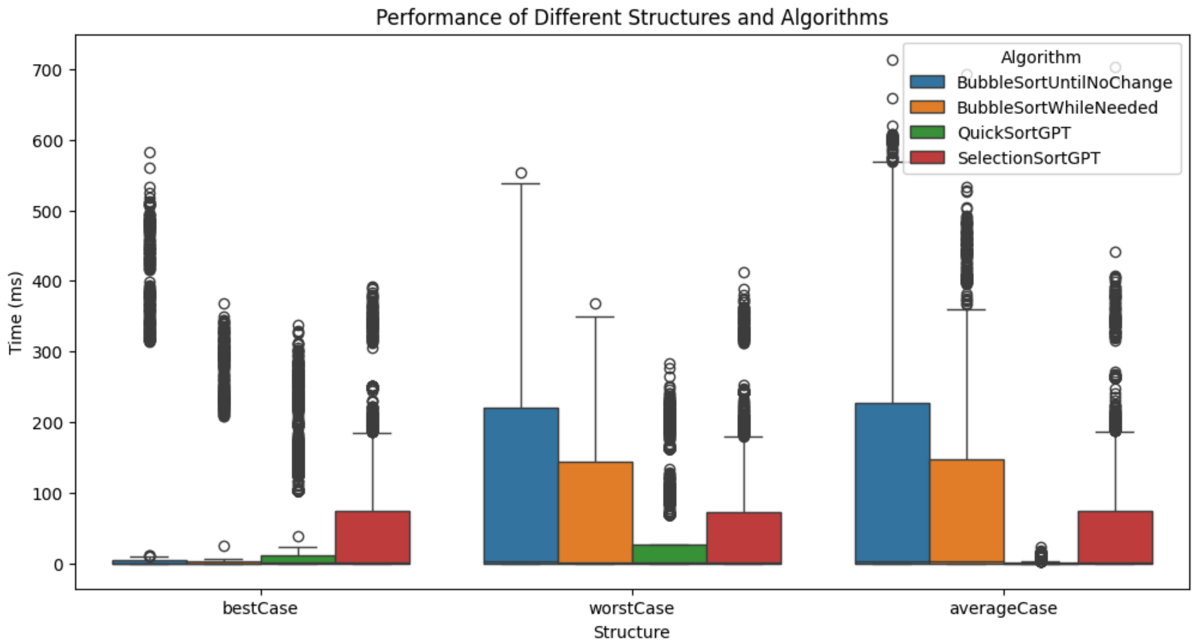
Plot of a visual representation of the execution time distribution for different data types used in the experiment: int, long, float, double, char, and string.



Labels: *See the figure above.*

From the plot, we can confirm that an important part of the experiment was to take different types of data since the performance of data types can be influenced by the complexity of the data being processed.

In the next graph, the last variable that could be generalized is the scenario.



Labels:

Data points beyond the whiskers are considered outliers and are plotted individually.

See the figure above.

Next, the performance on different input data sizes was analyzed. View file `analysis.ipynb`. The presented graphs provide an opportunity to comprehensively analyze the performance of different sorting algorithms under different input data sizes and scenarios.

In the outcomes of the experiment, QuickSortGPT (later QSgpt) and SelectionSortGPT (later SSgpt) appeared to be superior for all tested data types. However they were best performing at different data types.

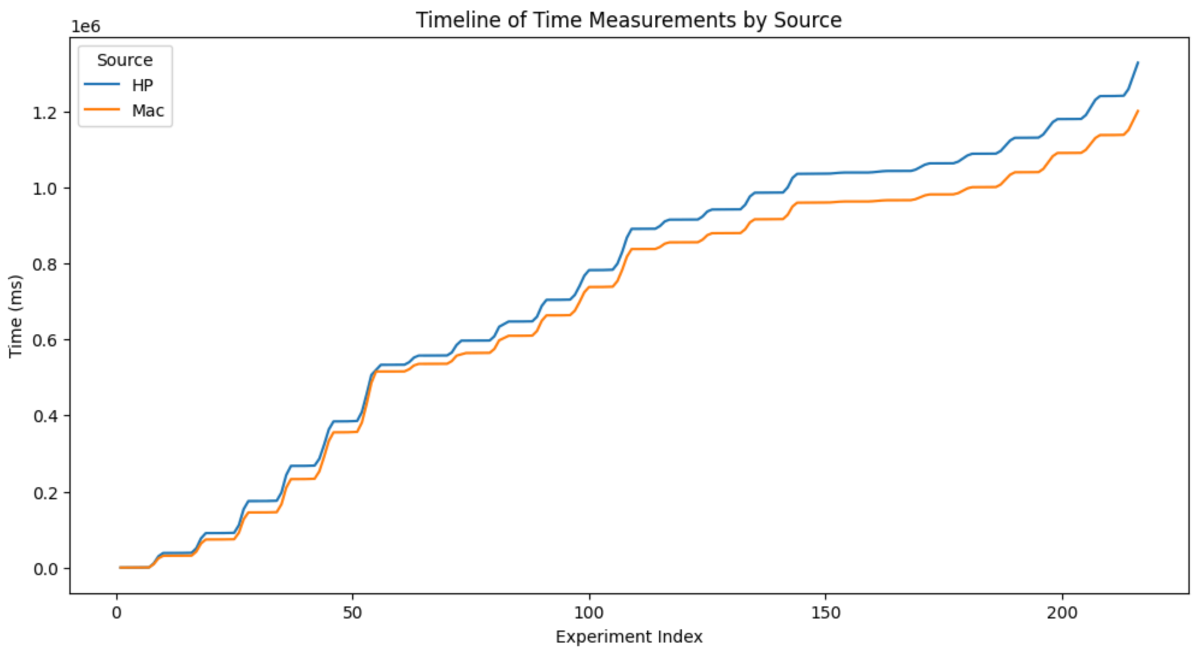
	Best data types	Analysis
QSgpt	Char, String	The QSgpt showed the fastest times to perform sorting on arrays of larger scales with non-numerical data types. While some algorithms can be faster in certain cases like low amount of elements to sort, or when the initial array is almost sorted, the QSgpt showed a great scalability with growth in time consumed severely lower than other algorithms. For arrays of extraordinary sizes, the QSgpt has absolute superiority over 3 other algorithms. It proved to be a reliable and scalable algorithm that can handle all array sizes in an efficient manner.

SSgpt	Int, Long, Foat, Double	The SSgpt showed the fastest times to perform sorting in arrays of larger scales with numerical data types. It was slightly slower than other presented algorithms for small size arrays, however at 10 000 elements it already outperforms other algorithms by a significant margin. Its growth of time consumed in relation to sorted elements is severely slower than other algorithms, which means with growth of elements in array, its efficiency will only grow in relation to other 3.

3.2 Descriptive Statistics

For comparison of actual outcomes of 4 sorting algorithms, their results were concluded in the tables in sections 3.2.1 - 3.2.4. The comparing metrics is ms - milliseconds it took the algorithm to finish sorting a given array. Each algorithm ran 100 tries of different cases for each N - number of elements in the array, after which statistics was created. All the statiscs based on results from laptop HP Omen 14*.

* - Although the results on the Macbook showed better performance (specifically a 27% speed increase), they hold the same tendency in relative comparison. Therefore HP was chosen for slightly richer information as it is using the most common OS for any settings - Windows OS.



The observed difference in execution time between the Mac M2 and HP Omen systems can be attributed in part to architectural differences in memory handling and JVM optimization.

The Mac M2's ARM-based architecture is specifically designed for high efficiency in memory and cache operations. Its unified memory architecture, which allows the CPU and

GPU to share a single pool of high-bandwidth memory, reduces latency for frequent, small read/write operations—such as those in sorting algorithms. This unified approach differs from the traditional separate RAM channels found in x86-based systems, such as the HP Omen, and allows for faster access times and lower latency, making the M2 particularly effective in memory-intensive tasks.

In addition, when running sorting algorithms in Java, macOS's Java Virtual Machine (JVM) incorporates Just-In-Time (JIT) compiler optimizations tailored to Apple Silicon's ARM architecture. These ARM-specific optimizations lead to more efficient bytecode execution, especially in repetitive operations where the JVM can dynamically optimize frequently used code paths over time. This combination of ARM's memory efficiency and the optimized JVM on macOS contributes to the M2's consistent performance in sorting algorithms.

As a result, the Mac M2 can achieve steady cumulative performance that often matches or exceeds that of the HP Omen.

3.2.1 BubbleSortUntilNoChange

For all the tables below:

- N - represents a number of elements in array.
- Quartile - represents the value below which and including it, the given % of data falls into.

The table is measured in milliseconds (ms):

Int

N \ quartile	Min	25%	50%	75%	Max
100	0.01	0.01	0.02	0.06	0.65
1000	0	0	1.5	1.79	4.29
10000	0.01	0.01	169.9	199.0575	247.58

Long

N \ quartile	Min	25%	50%	75%	Max
100	0.01	0.01	0.06	0.06	0.35
1000	0	0	1.99	2.24	5.43
10000	0.01	0.03	235.95	270.5025	389.94

Float

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.06	0.08	0.15
1000	0	0.01	3.08	3.61	7.18
10000	0.03	0.05	396.685	425.3	530

Double

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.04	0.04	0.07
1000	0	0	3.72	4.07	8.28
10000	0.03	0.03	425.51	479.08	536.39

Char

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.03	0.03	0.03	0.08
1000	2.93	3.17	3.36	3.8025	7.87
10000	313.7	354.4575	381.405	415.565	511.81

String

N \ quartile	Min	25%	50%	75%	Max
100	0.03	0.03	0.04	0.05	3.01
1000	4.21	4.52	4.65	5.085	14.39
10000	416.07	451.14	482.105	526.4675	713.84

3.2.2 BubbleSortWhileNeeded

Int

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.03	0.04	0.11
1000	0	0	0.99	1.3	15.93
10000	0.01	0.01	104.21	129.16	160.31

Long

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.02	0.02	0.22
1000	0	0	1.24	1.4225	3.12
10000	0.01	0.01	156.81	210.8	306.92*

* - In the experiment, there was a single entity of 236478.5 ms. It is unsure how a single sample reached such a significant time, but it was removed from statistics, however it should be taken into account that there is a possible flow of an algorithm that caused it.

Float

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.03	0.03	0.18
1000	0	0	1.89	2.0525	6.34
10000	0.02	0.03	218.805	267.68	348.25

Double

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.02	0.03	0.08
1000	0	0	2.13	2.34	5.77
10000	0.03	0.03	240.24	302.49	418.82

Char

N \ quartile	Min	25%	50%	75%	Max

100	0.02	0.02	0.02	0.02	2.05
1000	1.98	2.05	2.2	2.35	6.85
10000	207.18	229.8575	246.74	283.6675	359.41

String

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.02	0.03	0.04	0.05
1000	2.61	2.85	3.12	3.7825	25.83
10000	272.28	297.005	318.775	439.3325	693.48

3.2.3 QuickSortGPT

Int

N \ quartile	Min	25%	50%	75%	Max
100	0.01	0.02	0.03	0.17	0.63
1000	0.05	0.12	0.96	1.27	3.76
10000	0.71	0.78	93.905	133.4075	218.36

Long

N \ quartile	Min	25%	50%	75%	Max
100	0.01	0.01	0.02	0.03	0.48
1000	0.06	0.07	0.92	1.46	3.03
10000	0.82	0.9	99.485	148.65	217.64

Float

N \ quartile	Min	25%	50%	75%	Max
100	0.01	0.01	0.03	0.03	0.06
1000	0.07	0.18	1.77	2.16	9.51
10000	1.04	1.2175	193.89	227.1675	277.98

Double

N \ quartile	Min	25%	50%	75%	Max
100	0	0.01	0.02	0.02	0.06
1000	0.08	0.09	1.94	2.44	4.1
10000	1.08	1.795	210.45	264.19	338.25

Char

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0	0.01	0.01
1000	0.13	0.13	0.14	0.15	2.77
10000	9.67	10.02	10.3	11.14	21.59

String

N \ quartile	Min	25%	50%	75%	Max
100	0	0	0.01	0.01	0.03
1000	0.15	0.16	0.165	0.18	0.61
10000	11.26	11.9175	12.39	13.565	38.63

3.2.4 SelectionSortGPT

Int

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.02	0.02	0.02	0.26
1000	0.5	0.54	0.58	0.59	1.62
10000	57.87	60.41	64.535	69.91	97.4

Long

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.02	0.02	0.02	1.63
1000	0.69	0.7	0.71	0.73	2.15
10000	66.19	75.7575	81.28	88.6825	141.24

Float

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.02	0.02	0.02	0.07
1000	1.19	1.23	1.26	1.31	3.17
10000	120.95	128.305	134.38	145.205	203.51

Double

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.02	0.02	0.02	0.18
1000	1.44	1.49	1.515	1.56	5.87
10000	142.26	152.2775	160.64	168.2225	209.75

Char

N \ quartile	Min	25%	50%	75%	Max
100	0.02	0.02	0.02	0.02	0.02
1000	1.73	1.74	1.77	1.8525	6.47
10000	178.02	189.23	196.855	207.9175	247.41

String

N \ quartile	Min	25%	50%	75%	Max
100	0.03	0.03	0.03	0.03	0.07
1000	2.83	3	3.12	4.0525	21.29
10000	305.41	329.35	343.15	359.18	703.09

4. Discussion

4.1 Compare Hypothesis to Results

- The hypothesis was partially proven by the outcomes of the experiment. It was proven to be true that QuickSortGPT will perform best for all non-numerical data types, and proven to be untrue for numerical data types. It is unsure how the SelectionSortGPT algorithm with average time complexity of O(n^2) may surpass the QuickSortGPT algorithm with average time complexity of O(nlogn). However, it is not an exception that occurred once, as it is a tendency for all the numerical data types. It is possible due to the SelectionSorGPTt algorithm going through an array to find the smallest automatically, which is easy to do for numerical data types. Other possible reason is imperfect coding for given algorithms.

- The results show that architectural efficiency and memory management play a significant role in sorting algorithm performance. The MacBook Pro 14” M2’s unified memory architecture and ARM-optimized JVM allowed it to surpass the HP Omen, despite the Omen’s presumed advantage in processing power. This indicates that, for algorithmic tasks, system architecture and memory design can sometimes outweigh raw processing capabilities.

4.2 Limitations and Threats to Validity

- It is possible that algorithms presented in the experiment may have flaws, as the results didn’t line up with hypotheses based on the mathematical research.
- The * case of BubbleSortedWhenNeeded with long data types (a single element had a significant execution time) may represent the undetected flaw.
- The data used for analysis may be not full which will damage or invalid the results of this experiment

4.3 Conclusions

For low-scale arrays, it is a low difference in which algorithm to use, because O(nlogn) O(n) O(n^2) time complexities are very close to each other, and end result differences are insignificant taking into consideration average processing powers of modern PCs and professional working stations. However, with increasing in scale, only QuickSortGPT for non-numerical data types and SelectionSortGPT for numerical data types are viable, because their growth in average consumed time is massively less exponential than other algorithms.

Appendix

A. Materials

This document uses algorithms provided in attached “Algorithms” folders.
Hand-out is in “experiment-01.pdf”.

B. Reproduction Package (or: Raw Data)

This document supports and uses the data in the attached

- “Experiment-1-Performance-Measurement.zip” file.

To start experiment file “Main.java” should be executed. It uses “ArrayGenerator.java” and files found in “Algorithms” folder.

- An example of the output of the testing session is in “results.csv” file.
- Results for hpo14 is in “results-hpo14.csv”.
- Results for m2pro is in “results_m2pro.csv”.
- Merged resources is in “final_data.csv”

“Results hpo14 PROCESSED.xlsm” has the same content as Results hpo14, additionally processed for ease of representation in report. Was processed using macros found in file “MacrosForProcessing.txt”

Various graphs and results of data analysis, among with explanations can be found in Jupiter notebook file “analysis.ipynb”. It has useful insights, but not everything made it to the report due to lack of relevancy in context of the experiment.