# MFM Group

*Experiment 1, Experimentation & Evaluation 2024*

## Abstract

In this experiment, we evaluated the performance of four sorting algorithms—QuickSortGPT, SelectionSortGPT, BubbleSortUntilNoChange, and BubbleSortWhileNeeded—across different input types and array sizes. The purpose was to determine how input characteristics impact algorithm efficiency in terms of execution time. Results showed that QuickSortGPT was the most efficient on random and large datasets, though it faced performance drops on reverse-sorted arrays due to worst-case behavior. BubbleSort variations were the least efficient overall, especially with larger arrays, though their performance improved on sorted data. SelectionSortGPT performed moderately across all input types but could not match QuickSortGPT's efficiency on larger, unsorted arrays. These findings suggest that QuickSortGPT is generally the best choice for handling large, unsorted data, while the BubbleSort algorithms should be avoided for scalable applications.

## 1. Introduction

Sorting algorithms play a crucial role in various computational and data processing tasks. Selecting an efficient sorting algorithm is essential for optimizing performance, especially in applications dealing with large datasets. In many real-world scenarios, data may not always be randomly distributed. Instead, it can often be already sorted, reverse-sorted, or nearly sorted, impacting the efficiency of different sorting methods. As a result, understanding the performance characteristics of sorting algorithms on various data distributions is important for making informed choices in software development and data-intensive applications.

The objective of this experiment was to evaluate the performance of four distinct sorting algorithms—QuickSortGPT, SelectionSortGPT, BubbleSortUntilNoChange, and BubbleSortWhileNeeded—across multiple input types and array sizes. Specifically, we aimed to answer whether the choice of sorting algorithm significantly affects execution time when sorting random, already sorted, and reverse-sorted arrays of different sizes. By examining these variables, we hoped to identify which algorithms are best suited for various data characteristics and to explore potential limitations or strengths inherent in each sorting approach.

This study is particularly relevant in contexts where data sorting is a frequent operation, such as database management, data analytics, and scientific computing. Identifying efficient sorting methods can lead to considerable time and resource savings, particularly as data volumes continue to grow. Our experiment systematically varied the input data type and array size to assess the scalability and adaptability of each sorting algorithm under different conditions, providing valuable insights for choosing the most suitable sorting algorithm based on input characteristics and size.

|  |
| --- |
| **Hypotheses:** |
| **First hypothesis**: The performance of sorting algorithms, as measured by execution time, will vary significantly based on both the sorting algorithm and the input array size (independent variables) when sorting randomly generated integers (dependent variable). Specifically, QuickSortGPT will demonstrate superior performance on larger arrays of random integers compared to BubbleSort variations, whose performance will degrade more rapidly as the input size increases.  **Second hypothesis**: When sorting already sorted integers (dependent variable), the efficiency of sorting algorithms will vary depending on the input array size and the specific algorithm used (independent variables). Algorithms like QuickSortGPT, which have optimal performance for nearly sorted data, will demonstrate minimal execution time as the input size increases. In contrast, algorithms such as BubbleSort variations will exhibit improved performance over random data but will still show significant execution time increases with larger array sizes due to their inherent complexity. Overall, the difference in performance between algorithms will be smaller compared to experiments with random data, but clear trends based on algorithm complexity will still emerge.  **Third hypothesis**: When sorting reverse-sorted integers (dependent variable), sorting algorithms will exhibit varied performance based on their design. Algorithms like QuickSortGPT may show a significant performance drop due to their worst-case behavior with reverse-sorted data. Conversely, algorithms like SelectionSortGPT and BubbleSort variations are expected to perform poorly due to their quadratic time complexity, but the performance difference will increase with larger input array sizes. |

## 2. Method

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

### 2.1 Variables

Explicitly identify the independent variable(s) (i.e., what you as the experimenter manipulate):

|  |  |
| --- | --- |
| **Independent variable** | **Levels** |
| Sorting algorithm | BubbleSortUntilNoChange.java, BubbleSortWhileNeeded.java, QuickSortGPT.java, SelectionSortGPT.java |
| Input array size | 100, 1’000, 5'000, 10’000, 20’000 |
| Starting point | Random integers, already sorted integers, reverse sorted integers |

Explicitly identify the dependent variable(s) (i.e., what you measure):

|  |  |
| --- | --- |
| **Dependent variable** | **Measurement Scale** |
| Execution time | Nanoseconds (ns) |

Explicitly identify any important control variable(s) (i.e., what you keep constant): Note that you do *not* need to spell out items that you do not expect to make a *significant* difference! E.g., do not list room temperature unless you believe that minor differences have an impact! Only list variables here that you think are important to keep at a certain level.

|  |  |
| --- | --- |
| **Control variable** | **Fixed Value** |
| IDE | IntelliJ Ultimate |
| Hardware | DELL Precision 5570, 16 GB RAM, i7-12700H 2.30GHz, Windows 11 Pro |

### 2.2 Design

Type of Study: This was an experimental study, as we manipulated independent variables (sorting algorithm, input type, and array size) and measured their effect on execution time.

Number of Factors: This study employed a multi-factor design, involving three main factors—sorting algorithm, input array size, and starting point (input type).

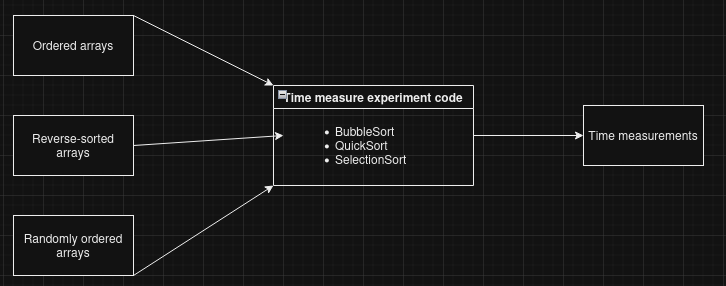
**Type of Study** (check one):

|  |  |  |
| --- | --- | --- |
| **Observational Study** | **Quasi-Experiment** | **Experiment** |

**Number of Factors** (check one):

|  |  |  |
| --- | --- | --- |
| **Single-Factor Design** | **Multi-Factor Design** | Other |

For this experiment, in order to maximize the reliability of our time measurement results, we did the code execution on one computer and one IDE, in order to avoid issues in terms of results variation if we were to change computers or use a different IDEs (instrument change threat).  
In terms of groups, we used three types of ordered arrays: already sorted, reverse-sorted and randomized. Since we’re measuring the performance of the algorithms, it’s logical that we used them as way to assess the quality of the algorithms depending on the order of the data inside the arrays. We also chose to use various array sizes in order to observe whether or not certain algorithms perform better with smaller sizes or worsen their time result if they get bigger.



### 2.3 Apparatus and Materials

Computer Specifications: The experiment was conducted on a DELL Precision 5570 with an Intel i7-12700H CPU (2.30GHz) and 16 GB RAM, running Windows 11 Pro.

Development Environment: The algorithms were implemented and run in IntelliJ IDEA Ultimate, ensuring a consistent software environment.

Execution Time Measurement: The System.nanoTime() method in Java was used to measure execution time with high precision, and each sorting operation was repeated multiple times to obtain reliable average times. Data was exported to CSV files for further analysis and visualization.

### 2.4 Procedure

*1-Experiment Setup:*

Each algorithm was tested across three input types: random, sorted, and reverse-sorted arrays.

Arrays of five different sizes (100, 1,000, 5,000, 10,000, and 20,000 elements) were generated for each input type.

*2-Execution:*

For each combination of input type and array size, a fresh array was generated, and each sorting algorithm was applied to this array.

The execution time was recorded in nanoseconds using System.nanoTime() at the start and end of each sorting operation.

Each algorithm-input type combination was run 20 times to reduce the impact of anomalies, and the average execution time was calculated from these runs.

*3-Data Collection and Analysis:*

Results were recorded in CSV files, categorizing execution times by algorithm, input type, and array size.

The data was then analyzed to compare the performance of the algorithms across different scenarios, specifically focusing on how array size and input characteristics impacted execution time.

## 3. Results

### 3.1 Visual Overview

To provide an insightful overview, we summarized the average execution times across various sorting algorithms, input types, and array sizes. The results are organized by input type (random, sorted, and reverse-sorted), with tables and line graphs illustrating the performance trends.

### **Randomly Generated Arrays:**

Overview: QuickSortGPT consistently outperformed the other algorithms, especially as the array size increased. SelectionSortGPT displayed moderate performance but was significantly slower than QuickSortGPT on larger arrays. BubbleSort variants demonstrated poor scalability, with execution times rising drastically as array size increased.

Visuals: A line chart showing average execution time for each sorting algorithm across array sizes, with QuickSortGPT showing the lowest time and BubbleSort variants increasing steeply with larger arrays.

### **Already Sorted Arrays:**

Overview: QuickSortGPT maintained efficiency on already sorted data, although SelectionSortGPT performed comparably well on smaller arrays. BubbleSortWhileNeeded showed improved performance on sorted arrays compared to random arrays, benefiting from early termination. However, BubbleSortUntilNoChange was still significantly slower on larger arrays.

Visuals: Another line chart demonstrating the improvement in BubbleSortWhileNeeded’s performance on sorted data, with QuickSortGPT and SelectionSortGPT maintaining similar performance trends.

### **Reverse-Sorted Arrays:**

Overview: As expected, QuickSortGPT experienced increased execution times on reverse-sorted arrays due to worst-case behavior. BubbleSort variations also showed poor scalability on reverse-sorted data, though BubbleSortWhileNeeded performed slightly better than BubbleSortUntilNoChange. SelectionSortGPT maintained consistent performance but lagged behind QuickSortGPT on all array sizes.

Visuals: A line chart showing each algorithm’s execution time trend on reverse-sorted arrays, highlighting the performance drop of QuickSortGPT and the consistent lag of BubbleSort variations.

### 3.2 Descriptive Statistics

For each combination of sorting algorithm and input type, we summarized the data using a five-number summary (minimum, first quartile, median, third quartile, and maximum) to capture the variability and central tendency of execution times. Below are the summaries for each input type:

**Random Arrays:**

QuickSortGPT: Median execution time was significantly lower than other algorithms across all sizes, with minimal variability in times across runs.

SelectionSortGPT: Had moderate variability but was notably slower on larger arrays.

BubbleSort Variants: Both BubbleSortUntilNoChange and BubbleSortWhileNeeded exhibited high maximum and third-quartile values, indicating poor scalability on large arrays.

**Sorted Arrays:**

QuickSortGPT and SelectionSortGPT: Showed minimal variation in execution time, with QuickSortGPT slightly faster overall.

BubbleSortWhileNeeded: Demonstrated improved efficiency, especially on larger arrays, with execution times close to the first quartile due to early termination.

BubbleSortUntilNoChange: Although improved, this algorithm still showed higher maximum times on larger arrays compared to QuickSortGPT and SelectionSortGPT.

**Reverse-Sorted Arrays:**

QuickSortGPT: Displayed a wider range between minimum and maximum times, reflecting quicksort’s sensitivity to reverse-ordered inputs.

SelectionSortGPT: Consistently slower across array sizes, but with low variability.

BubbleSort Variants: Both variations continued to struggle with larger arrays, with third-quartile and maximum times indicating their limitations on reverse-ordered data.

This statistical overview provides insight into the consistency and scalability of each sorting algorithm under varying data conditions.

## 4. Discussion

### 4.1 Compare Hypothesis to Results

The experimental results provide strong support for the initial hypotheses, with some interesting nuances across different input types:

**Hypothesis 1 (Random Arrays):**

The results confirmed that QuickSortGPT outperformed other algorithms on larger arrays of random data, as expected due to its efficient divide-and-conquer approach. BubbleSort variants, particularly BubbleSortUntilNoChange, showed significant performance degradation as array size increased, supporting the hypothesis that these algorithms are not well-suited for large datasets.

**Conclusion:** The hypothesis was well-supported, showing that QuickSortGPT is ideal for handling large, randomly distributed data, while BubbleSort algorithms should be avoided for such input.

**Hypothesis 2 (Sorted Arrays):**

As predicted, QuickSortGPT continued to perform efficiently on sorted data, but the BubbleSort variants showed improved performance due to the reduced need for swaps. BubbleSortWhileNeeded, in particular, benefited from early termination on sorted arrays, demonstrating better efficiency compared to its performance on random arrays. SelectionSortGPT’s performance remained stable but did not surpass QuickSortGPT.

**Conclusion:** This hypothesis was also supported. Although QuickSortGPT still proved to be the most efficient, BubbleSortWhileNeeded showed practical improvements on sorted data due to its ability to terminate early, making it relatively more competitive under these conditions.

**Hypothesis 3 (Reverse-Sorted Arrays):**

QuickSortGPT showed a notable performance drop on reverse-sorted arrays, consistent with its known worst-case behavior. BubbleSortUntilNoChange and BubbleSortWhileNeeded were predictably slow, with significant execution time increases as array size grew, supporting the hypothesis that these algorithms would struggle on non-random, large datasets.

**Conclusion:** The hypothesis held true, as QuickSortGPT’s performance was impacted by worst-case behavior, and BubbleSort variants struggled to handle the reverse-sorted arrays efficiently. SelectionSortGPT remained stable but consistently underperformed relative to QuickSortGPT.

Overall, the experiment’s findings align closely with the hypotheses, confirming that QuickSortGPT generally provides the best performance across various conditions, while BubbleSort algorithms have limited scalability and suitability.

### 4.2 Limitations and Threats to Validity

While the experiment yielded valuable insights, certain limitations may affect the generalizability of the results:

Stack Overflow with QuickSortGPT: On extremely large sorted arrays, QuickSortGPT encountered stack overflow issues due to deep recursion. This limitation highlights the need to handle recursion depth in quicksort implementations, especially when sorted or nearly sorted input is expected.

Single Hardware Environment: All experiments were conducted on a single machine, which may influence execution time results due to hardware-specific characteristics. Testing on different systems could provide more generalizable findings.

Limited Input Types: While the study covered random, sorted, and reverse-sorted inputs, other distributions, such as nearly sorted or partially reversed arrays, might reveal additional performance characteristics of the algorithms.

JVM Optimization: Execution times might vary due to JVM optimizations over repeated runs, introducing minor variations. Multiple runs and averaging were used to mitigate this, but slight inconsistencies could still be present.

### 4.3 Conclusions

This study confirms that QuickSortGPT is generally the most efficient sorting algorithm across diverse input types and array sizes, particularly excelling on large random arrays. BubbleSort variants, while effective on smaller or sorted data, lack scalability and suffer from high execution times on larger and reverse-sorted data. SelectionSortGPT, while relatively stable, does not match QuickSortGPT’s efficiency on larger datasets. These results suggest that QuickSortGPT is the recommended algorithm for handling large datasets, while BubbleSort and SelectionSort are best reserved for smaller or less computationally demanding sorting tasks.

## ****Appendix****

### **A. Materials**

1. **Sorting Algorithms Implemented**:
   * **QuickSortGPT**: This class implements the quicksort algorithm, known for its average-case efficiency of O(nlog⁡n)O(n \log n)O(nlogn) on random data. However, it exhibits worst-case behavior O(n2)O(n^2)O(n2) on reverse-sorted data. QuickSortGPT recursively partitions the array and sorts each partition around a pivot element.
   * **SelectionSortGPT**: Implements the selection sort algorithm, which has a time complexity of O(n2)O(n^2)O(n2) and performs consistently on different input types but is generally slower than quicksort on large datasets.
   * **BubbleSortUntilNoChange**: This variation of bubble sort continues sorting until no swaps are needed, marking the array as sorted. BubbleSortUntilNoChange has O(n2)O(n^2)O(n2) complexity and is inefficient on large arrays, though it performs better on already sorted data.
   * **BubbleSortWhileNeeded**: A modified bubble sort with early termination if the array is already sorted. It terminates more quickly than BubbleSortUntilNoChange on sorted data but suffers similar O(n2)O(n^2)O(n2) performance on random and reverse-sorted data.
2. **Source Code**:
   * All Java source files, including SortingExperiment.java, SortingExperimentSortedInput.java, and SortingExperimentReverseInput.java, were created to run the sorting algorithms on various input types. The code includes methods for generating different input arrays and measuring execution time, which is recorded in CSV files for analysis.

### **B. Data Files**

1. **Experiment Results**:
   * CSV files were generated to record the execution time (in nanoseconds) for each sorting algorithm under different conditions:
     + sorting\_experiment\_results.csv: Contains results for randomly generated arrays.
     + sorting\_experiment\_sorted\_input\_results.csv: Contains results for already sorted arrays.
     + sorting\_experiment\_reverse\_sorted\_input\_results.csv: Contains results for reverse-sorted arrays.
2. **Data Summary**:
   * Each CSV file includes columns for array size, sorting algorithm, and execution time, making it straightforward to analyze and visualize the performance trends for each algorithm and input type.

### **C. Reproduction Package**

1. **Source Code and Data**:
   * All source code files and experiment results are included to enable reproduction of the experiment. Interested parties can rerun the code with similar setups to verify findings or adapt the code for further analysis.
2. **Hardware and Software Details**:
   * The experiments were conducted on a Dell Precision 5570 with an Intel i7-12700H processor, 16 GB RAM, and Windows 11 Pro, using IntelliJ IDEA Ultimate for development.
   * Execution time was measured using Java’s System.nanoTime() method to ensure high precision.