MFM GROUP

Experimentation & Evaluation

UNIVERSITÀ DELLA SVIZZERA ITALIANA
Authors: Mehmet Fatih Tekin, Mike Fiorita, Mustafa Özyürek

Abstract

This experiment evaluated the performance of four sorting algorithms labeled as QuickSortGPT, SelectionSortGPT, BubbleSortUntilNoChange and BubbleSortWhileNeeded for different types of inputs as well as array sizes. The aim was to find out the influence of input factors on the performance of the algorithm in terms of execution time. Among others, the research results indicated that QuickSortGPT was the most efficient algorithm on random and large datasets; however, due to its worst-case characteristics, declines in speed were noted for this algorithm on reverse-sorted array. Yet while variants of BubbleSort performed relatively well on already sorted dataset, they less frequently ranked at the top especially when applied on bigger problems. On larger unsorted arrays, SelectionSortGPT was not as efficient as QuickSortGPT, however, it was satisfactory across various input types. Based on these findings, BubbleSort type algorithms cannot be relied on for systems that require scalability, in which case QuickSort is often the preferred alternative for handling large amounts of data which is unsorted.

1. Introduction

There are several tasks that involve computation and data processing where sorting is an important activity. Thus, an appropriate sorting algorithm must be in place so as to enhance the workload efficiency, and sorting problem performance in particular. Nevertheless, in several cases encountered in practice, the data is not purely random. It could be either sorted, reversed, or almost sorted which therefore affects sorting techniques in one way or the other. For this reason, it is crucial to understand how different sorting algorithms perform on different kinds of input distributions so as to make sound decisions regarding the implementation of sorting within applications that deal with large amounts of data.

The purpose of this experiment was to test four sorting algorithms (QuickSortGPT, SelectionSortGPT, BubbleSortUntilNoChange and BubbleSortWhileNeeded) on various types of input and sizes of arrays. More specifically, we wanted to find out if the sorting algorithm chosen has any effect on the execution time of the process when random, sorted and reversed-sorted arrays of different dimensions are being sorted. In doing so, we expected to determine which are the most appropriate algorithms for given data types and check some advantages or disadvantages that each sorting method has.

This particular analysis is most significant where sorting is static and there are many databases, data analysis, or scientific calculations. Efficiency in sorting techniques can save a lot of time and costs more so with the increase in the amount of data being processed. In this experiment, we applied a systematic approach towards manipulating the type of input data as well as the size of the array in order to evaluate the scalability and flexibility of the performance of each sorting algorithm under different scenarios thereby aiding in determining the best possible options for sorting their data.

Hypotheses:

Null hypothesis: There's no discernible difference in performance between all algorithms, the time measurements that we will obtain through this experiment are going to be similar with each other.
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

This section provides a comprehensive guide for replicating the experiment on sorting algorithm performance. It covers the manipulation of independent variables (sorting algorithm, input array size, input type) and the measurement of the dependent variable (execution time). We detail the experimental setup, hardware and software configurations, and procedures used, ensuring the accuracy of time measurements and consistency across tests. The results were recorded in CSV files and analyzed for insights into each algorithm's efficiency and scalability under different conditions.

2.1 Variables

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

Dependent variable	Measurement Scale
Execution time	Nanoseconds (ns)

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

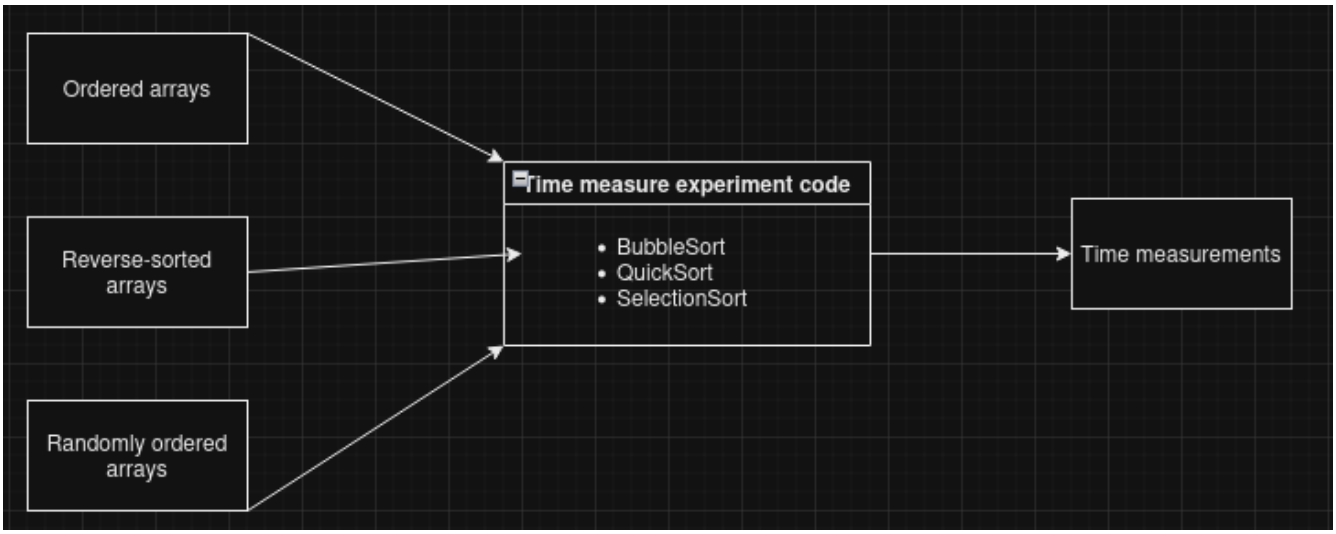
2.2 Design

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.

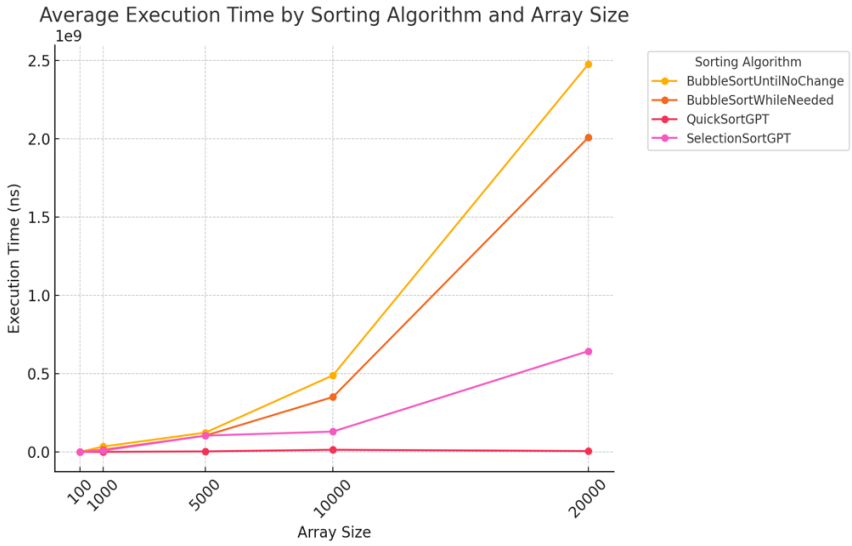
3. Results

3.1 Visual Overview

Through the results that we have obtained, we can say that the null hypothesis has been disproved, since they show substantial difference in execution time and performance in various situations. 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:

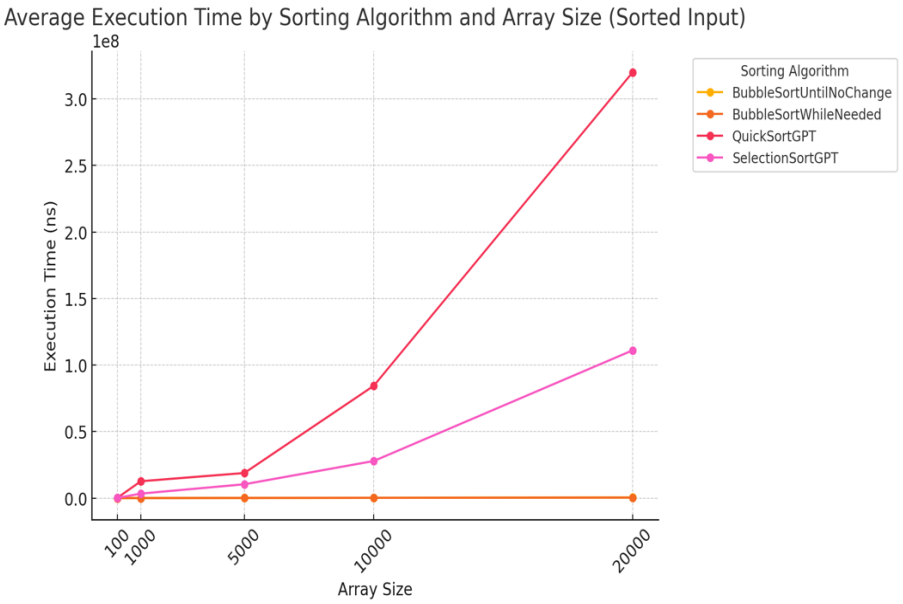
Summary: QuickSortGPT is by far the best-performing algorithm in this survey, and its superiority only increases with the size of the array. SelectionSortGPT showed moderate performance but was outclassed by QuickSortGPT, especially on large arrays. The BubbleSort variants were not scalable as correlating with the increase in the array size, the execution time increased tremendously.



Graph: A line graph plotting the average execution time for every algorithm against the size of the array, where QuickSortGPT takes the least time and the time for Bubble sort variants increases steeply with the increase in the size of the array.

Already Sorted Arrays:

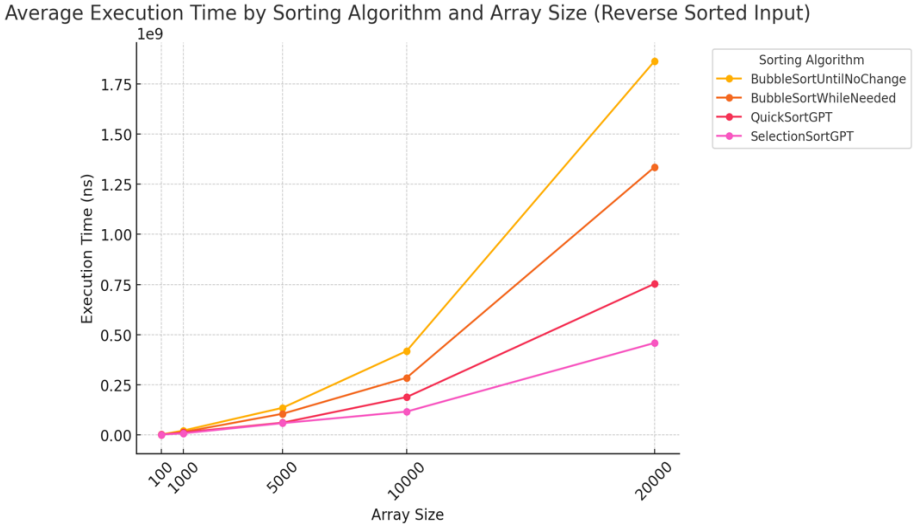
Summary: QuickSortGPT remained efficient on sorted sequences, however, SelectionSortGPT fared well with small-sized datasets. Effective groups suited decreasing order of execution for every array of degenerative random cylindrical perforated patterns within the g-structures. Graham's scan based hierarchical clustering of points space enabling their active organization is also significantly less performant on aggregates of data of greater size.



Graph: A line graph showing the enhancement of performance of BubbleSortWhileNeeded on sorted input data with the performance trends of QuickSortGPT and SelectionSortGPT remaining on similar levels.

Reverse-Sorted Arrays:

Summary: In line with expectations, QuickSortGPT exhibited slower execution times on reverse-sorted input arrays owing to its worst-case behavior. Similarly, variations of BubbleSort also portrayed poor scalability on reverse sorted data but BubbleSortWhileNeeded performed slightly better than BubbleSortUntilNoChange. On the other hand, SelectionSortGPT delivered a steady performance, despite falling behind QuickSortGPT in every array size.



Graph: A line graph plotting the changes in execution time of the different algorithms tested on reverse-sorted arrays, where the fluctuations of the performance of QuickSortGPT and the low performance of the BubbleSort variations are shown.

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:

Null Hypothesis:
Primarily, the results of the experiment reveal substantial variation in the amount of time taken by different algorithms to execute, more so with regard to the sizes and types of input data. Such outcomes give no credence to the null hypothesis that there would be no statistically significant performance differences of the algorithms. On the contrary, QuickSortGPT registered better performance when tested with random and large data sets while various versions of BubbleSort were found to be the slowest even without sorted arrays. Results from using the SelectionSortGPT were average across the board, although QuickSortGPT always outperformed it when the data size was increased.

Hypothesis 1 (Random Arrays):
Research outcomes revealed that, as anticipated, QuickSortGPT proved itself superior to all other algorithms on the random data larger arrays since it employs the divide-and-conquer strategy efficiently. However, BubbleSort types especially BubbleSortUntilNoChange showed great performance drop when the arrays grew in size, affirming the claim that such algorithms are unsuitable for big data.
Conclusion: The theory was proven correct, demonstrating that QuickSortGPT is suitable for massive verifiable random data distributions, while the use of BubbleSorts in such an instance should be discouraged.

Hypothesis 2 (Sorted Arrays):
As anticipated, QuickSortGPT maintained a high level of performance on ordered data while the modifications of BubbleSort were able to improve their performances, with reduced amount of required swaps being the main contributing factor. In particular, BubbleSortWhileNeeded was able to utilize early-exit on sorted arrays effectively, hence showing more efficiency than when used on random arrays. The performance of SelectionSortGPT also remained within similar levels, yet it did not surpass QuickSortGPT's efficiency.
Conclusion: This hypothesis was also supported. While QuickSortGPT still remained the most efficient one, a practical benefit was observed for BubbleSortWhileNeeded on sorted data, which made it more competitive under such circumstances thanks to the early exit possibility.

Hypothesis 3 (Reverse-Sorted Arrays):
QuickSortGPT performed as expected in that it experienced a performance drop during the tests with the arrays that were in reverse order which was in line with the theoretical worst-case situation known for the algorithm. BubbleSortUntilNoChange and BubbleSortWhileNeeded were unsurprisingly slow, with their execution times increasing significantly as the size of the array increased, thereby supporting the assumptions that these algorithms would have difficulties with large datasets that are non-random in nature. Inference: As predicted, performance of QuickSortGPT was low due to its worst-case behavior, while BubbleSort versions performed

badly on reverse-sorted datasets, but fared better than QuickSortGPT. SelectionSortGPT fared consistently poor when compared to QuickSortGPT, yet exhibited rather stable performance levels.

Conclusion: The results of the experiment were consistent with the initial propositions, particularly with regard to the performance superiority of QuickSortGPT in most of the tests and the inferior performance of BubbleSort algorithms in terms of performance scalability.

4.2 Limitations and Threats to Validity

Although the experiment conducted was informative, there are some factors that can limit the extent of applicability of the findings:

QuickSortGPT and Stack Overflows: While sorting very large ordered arrays with QuickSortGPT, the quick sorter reached stack overflow due to excessive use of recursion. This underlines the need for proper management of recursion depth while implementing quicksort algorithm because such inputs, especially where the data is sorted or almost sorted, tend to be commonplace.

Single Hardware Environment – All tests were conducted on a single processor and this may impact the time performance measurements because of the dependent on the particular hardware. Testing on diverse systems would add to more conclusive results.

Input type Restriction: The experiments focused on random, sorted, and reverse-sorted input types, but other types such as almost sorted or varying degrees of reversal may expose other variables affecting performance of the algorithms.

JVM Tuning: It is possible that some of the changes in execution times on different iterations of the test were simply due to some optimizations that the JVM performed. To address this, averaging was done after several trials however, some variations may still be the case.

4.3 Conclusions

This research shows that QuickSortGPT is unmatched as a sorting algorithm across different types of inputs and sizes of arrays, most notably on massive random arrays. Variants of BubbleSort are helpful to an extent for smaller or already sorted data but do not scale well and have high runtime costs for larger or reversed sorted data. On the other hand, SelectionSortGPT which seems relatively consistent does not outperform the efficiency of QuickSortGPT even on medium databases. Based on these findings, it can be concluded that the most preferred algorithm for large data handling is the QuickSortGPT algorithm, while algorithms like BubbleSort and SelectionSort are better suited for sorting that is smaller in scale and less resource intensive.

Appendix

A. Materials

1. **Sorting Algorithms Implemented**:
    - **QuickSortGPT**: This class implements the quicksort algorithm, known for its average-case efficiency of $O(n\log_n)$ on random data. However, it exhibits worst-case behavior $O(n^2)$ 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 $O(n^2)$ 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 $O(n^2)$ 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 $O(n^2)$ performance on random and reverse-sorted data.
2. **Source Code**:
    - All Java source files, including:
      `SortingExperiment.java`;
      `SortingExperimentSortedInput.java`;
      `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.
3. **GitHub Repository:**
   https://github.com/fatihtkin/Exp-Eval1

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.