What is the best performing sorting algorithm?

Experimental analysis commissioned by Bubble Inc

Ghilardini Matteo, Toscano Sasha

November 11, 2024

Abstract

In this document, we analyze and compare the performance of four sorting algorithms implemented in Java in order to determine which one represents the optimal choice in terms of efficiency and speed for the implementation of a utility function within a Bubble Inc. library.

We measured and analyzed the sorting speeds of BubbleSortUntilNoChange, BubbleSortWhileNeeded, QuickSortGPT, and SelectionSortGPT in various scenarios by modifying the size, data type, and initial order of the dataset.

To ensure a good level of reliability in our analysis, we repeated the sorting process multiple times for each combination of independent variables.

The performance, i.e., the sorting time of each algorithm, was measured in nanoseconds using the Java function System.nanoTime().

The collected data show that the QuickSortGPT algorithm offers the best performance on randomly filled datasets, which is the most common scenario in which sorting will be required. This is due to its Average Time Complexity of $O(n \log n)$.

We also noticed that SelectionSortGPT and both implementations of BubbleSort have scenarios where their performance is better than that of QuickSortGPT, but these are very remote scenarios that are extremely unlikely to occur in the real world.

1 Introduction

In this experiment, we aim to answer an important question: *Which sorting algorithm implementation in Java is most efficient?* Efficient sorting is critical in computer science because many applications rely on sorting as a fundamental operation, whether in data processing, search optimization, or user interfaces. Selecting the most effective sorting algorithm can significantly impact the performance of these applications, especially when handling large datasets.

In this study, we will analyze several Java sorting algorithm implementations with distinct characteristics:

- BubbleSortUntilNoChange
- BubbleSortWhileNeeded
- QuickSortGPT
- SelectionSortGPT

To determine the best sorting performance, we will compare the algorithms by measuring the time required to sort arrays of varying sizes and data types. To enhance the reliability of our findings, we will average sorting times across multiple repetitions for each scenario.

By examining the efficiency of each algorithm in this way, we hope to provide insights into which implementation is most suitable for different use cases.

Hypothesis

The QuickSort algorithm is the best one in terms of performance since its complexity is $O(n * logn)$; compared to BubbleSort $O(n^2)$ and Selection Sort $O(n^2)$.

We expect this behavior for every data type, array size and starting array order.

2 Method

2.1 Variables

Independent variable	Levels	Description
Different algorithms	4	4 different algorithms
Data types	3	Integer (4 bytes), Char (2 bytes), Boolean (1 bit)
Size of the array	2	Integer number, we selected 1,000 and 10,000
Starting array order	3	Sorted, Shuffled, or Reverse-sorted

Table 1: Independent variables

Dependent variable	Measurement scale	Measurement method
Time to complete the sorting	nanoseconds	`System.nanoTime()`

Table 2: Dependent variables

Control variable	Fixed value
Number of active applications	None outside of IDE and WSH Terminal
Computer HW, OS, JDK	Intel i7-9700k, 32GB RAM, Windows 10, OpenJDK 23.01

Table 3: Control variables

2.2 Design

2.2.1 Study Type

Observational Study	Quasi-Experiment	Experiment
☐	☐	☒

Table 4: Study type selection

We are conducting an **experiment** because we are actively *manipulating the independent variables* (i.e., we observe how performance changes with different sorting algorithms, data types, array sizes, and initial orders). This manipulation allows us to examine cause-and-effect relationships between these variables and the time taken to sort.

Additionally, we are using **randomization** in our experimental design. The data used to populate the arrays for each trial are generated randomly, ensuring that each algorithm operates on arrays that are equivalent in content and structure for a given data type and size. This helps balance out any potential biases or differences, as each algorithm sorts the same sets of data under comparable conditions.

We also have **control conditions**, where we can compare each algorithm's performance as a benchmark or "status quo" against others within the same experimental conditions. This allows us to determine if certain algorithms outperform others or if the effects vary based on the data characteristics.

2.2.2 Number of Factors

Single-Factor Design	Multi-Factor Design	Other
☐	☒	☐

Table 5: Number of factors selection

Our experiment uses a **Multi-Factor Design** since we are manipulating multiple *independent variables* to observe their effect on the dependent variable, the sorting time.
This design allows us to observe the effect of all independent variables on the dependent variable.
By analyzing all combinations, we are performing a *Full factorial design* resulting in:

$$\#\text{algorithms} \times \#\text{dataTypes} \times \#\text{sizes} \times \#\text{startingOrders} = 4 \times 3 \times 2 \times 3 = 72 \text{ different results}$$

To visualize this concept, we can imagine it as a flowchart diagram, where each dependent variable is a state with n possible outcomes from it. As follows (to avoid drawing all 72 possible paths, we have simplified the drawing by following the path of *QuickSortGPT, 1'000, Random, ...*):

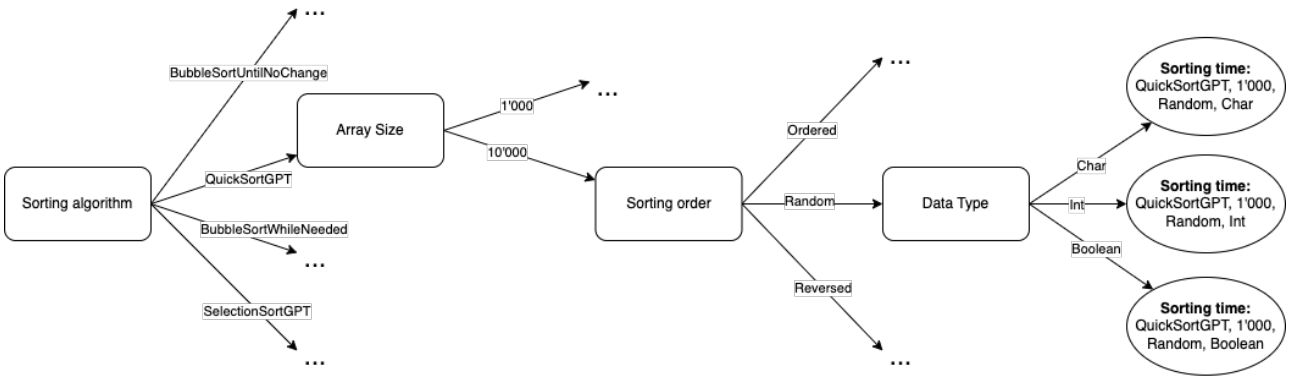


Figure 1: Multi-Factor Design

2.3 Apparatus and Materials

We will run all experiments on a desktop computer with the following hardware specifications:

- **CPU:** Intel Core i7-9700k
- **RAM:** 32GB DDR4

The computer runs on Windows 10 as the operating system, but all experiments and the JDK are executed through the WSL with Ubuntu 23.04.

We are using OpenJDK 23.01 as the Java version and as IDE we used Microsoft VSCode. For all experiments performed, we save all metrics in a CSV file, and to plot the data, we use Microsoft Excel 365.

To measure the metrics of the experiments, we use the Java function `System.nanoTime()` and save the value before and after sorting. The elapsed time is then computed as `startTime - endTime`.

2.4 Procedure

The procedure for running the experiment begins by restarting the PC (so as to make sure there are no background applications running). When the machine is restarted, we start only the IDE (VSCode) and the WSL terminal. At this point we execute our code.

We have implemented a function `runTests(repetitions, arraySize)`, and we call it from the `Main` method 3 times:

- `runTests(10, 1000)` is a warm-up run, so we only need a few iterations on an array of discrete size
- `runTests(100, 1000)`
- `runTests(100, 10000)`

We opted to choose 10 repetitions of an array of size 1'000 as a valid warm up as during our testings we saw that most of the data stabilized after roughly 2-5 iterations. So we accounted for the worst-case scenario and added some extra margin to prevent possible outliers. We also saved that data in different files *(results/warmup*)* to be able to compare it and make sure that there was still a significant difference in the results. And as we can see from comparing the results from table 2b and the results of the warm up averages (which can be seen in figure 22) there is about a 30% difference in all averages.

The function `runTests(...)` sets up the environment for the testing; it creates all the arrays of different types (int, char, boolean) and different orders (random, sorted, reverse-sorted) with the corresponding functions, and sets the path of all the CSV files where we will save the results. After that, we call the function `performSortingTests(...)` that is responsible to measure and save the results of all sorting algorithms.

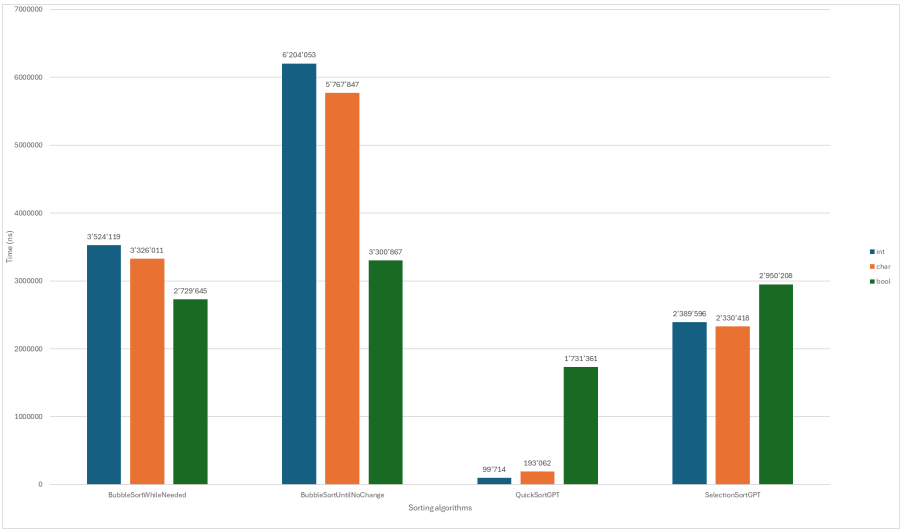
The function `performSortingTests(...)` runs all sorting algorithms for the given combination of independent variables, and stores the resulting data and the average sorting time in a specific CSV file.

3 Results

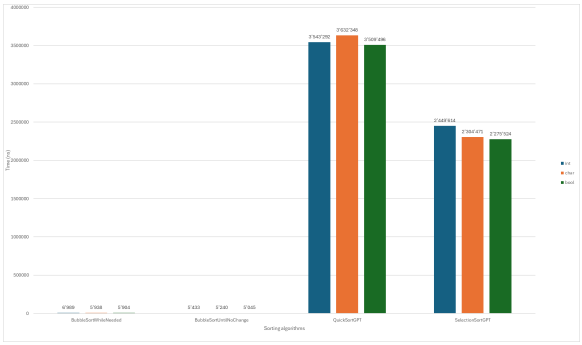
3.1 Visual Overview

In the visual overview we have chosen to report graphs representing the average sorting times of the various algorithms. We have chosen to keep the graph for the randomly filled arrays larger, and therefore more visible, since it represents the "average case" for which our library will be used.

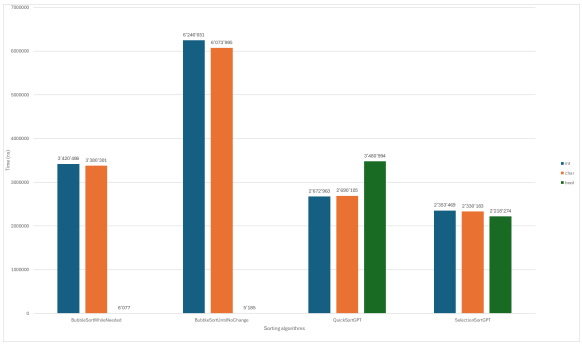
As an appendix to the paper, however, graphs representing the sorting times of each combination of the independent variables for each of the 100 repetitions performed are also shown in chapter 5. In these graphs it can be easily seen that the performance of each sorting algorithm tends to be smooth with only a few outliers.



(a) Random filled array

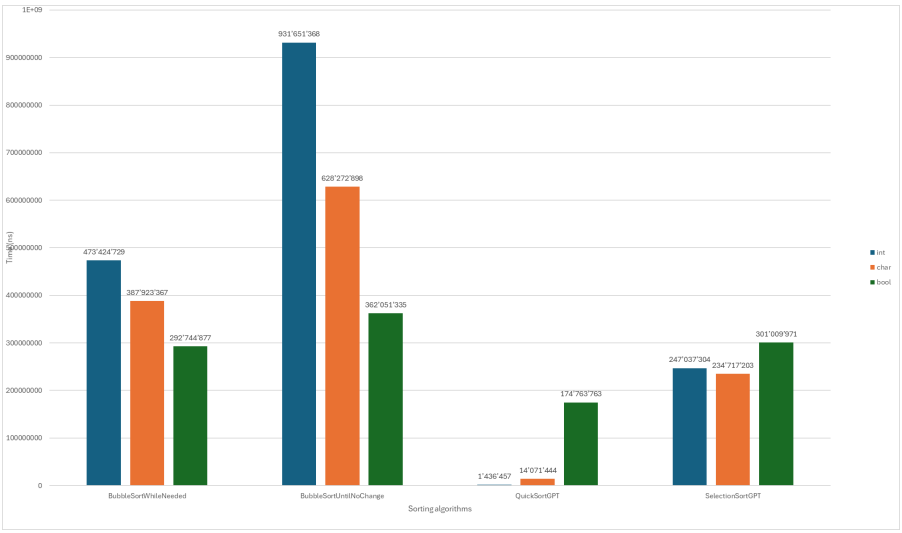


(b) Sorted array

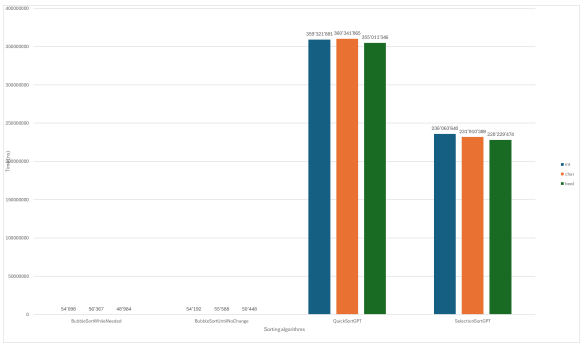


(c) Reverse sorted

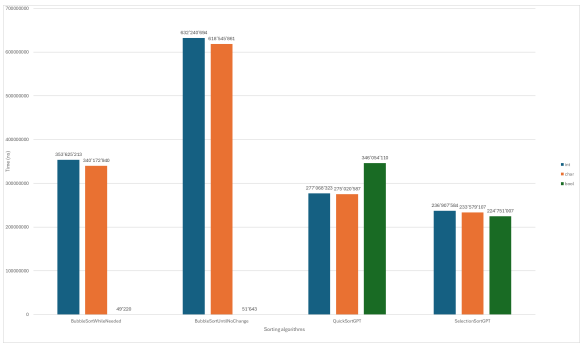
Figure 2: Average sorting time - 1000



(a) Random filled array



(b) Sorted array



(c) Reverse sorted

Figure 3: Average sorting time - 10'000

3.2 Descriptive Statistics

In the tables below, we have reported the main statistics from our analysis. The data is formatted so that the best (the smallest) results are highlighted in green, the worst (the largest) in red, and the other data are formatted using a color gradient within the range.

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	3'374'700	5'913'800	93'800	2'274'100
First Quartile	3'415'900	6'017'750	96'200	2'302'075
Median	3'446'250	6'084'250	97'550	2'332'300
Third Quartile	3'497'225	6'255'450	98'925	2'392'950
Maximum	7'183'400	9'874'000	173'000	4'390'900
Mean	3'524'119	6'204'053	99'714	2'389'596
Standard Deviation	390'501	439'298	10'025	228'574

Figure 4: Statistics for int[1000] random filled

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	5'100	5'000	3'200'700	2'235'300
First Quartile	5'500	5'100	3'423'425	2'264'175
Median	5'700	5'200	3'490'650	2'292'750
Third Quartile	6'000	5'300	3'572'600	2'347'600
Maximum	64'400	19'300	4'801'600	13'310'800
Mean	6'989	5'433	3'543'292	2'449'614
Standard Deviation	6'791	1'659	238'810	1'107'629

Figure 5: Statistics for int[1000] sorted

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	3'124'500	5'818'400	2'484'400	2'253'100
First Quartile	3'296'375	6'074'225	2'572'825	2'279'475
Median	3'429'550	6'170'800	2'664'600	2'306'900
Third Quartile	3'500'450	6'283'925	2'717'150	2'376'550
Maximum	4'030'900	8'227'600	3'184'200	3'462'300
Mean	3'420'486	6'246'651	2'672'963	2'353'469
Standard Deviation	176'370	367'856	134'855	154'508

Figure 6: Statistics for int[1000] reverse

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	456'400'206	902'975'593	1'307'600	236'549'396
First Quartile	465'684'873	916'135'572	1'381'100	242'104'154
Median	468'468'954	925'407'299	1'396'850	245'107'149
Third Quartile	476'198'023	940'537'274	1'441'350	249'225'933
Maximum	533'931'604	996'822'190	2'025'100	279'570'901
Mean	473'424'730	931'651'369	1'436'457	247'037'304
Standard Deviation	13'575'116	21'628'438	118'773	7'458'218

Figure 7: Statistics for int[10000] random filled

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	47'000	50'200	350'005'100	229'824'600
First Quartile	48'100	50'300	354'067'300	232'692'346
Median	48'700	50'400	356'320'801	234'663'150
Third Quartile	51'400	50'725	361'037'075	237'252'449
Maximum	148'400	218'000	428'085'900	256'548'800
Mean	54'098	54'192	359'321'882	236'063'641
Standard Deviation	14'931	17'852	11'391'738	5'157'531

Figure 8: Statistics for int[10000] sorted

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	324'670'599	604'465'499	260'517'800	231'840'001
First Quartile	348'446'000	623'432'077	273'491'398	233'934'325
Median	350'801'700	628'083'941	275'689'001	235'382'097
Third Quartile	355'031'928	636'445'025	277'942'925	237'794'005
Maximum	425'545'601	780'993'301	317'155'499	259'537'999
Mean	353'625'213	632'240'695	277'068'323	236'907'584
Standard Deviation	11'630'692	19'751'399	7'686'852	4'919'102

Figure 9: Statistics for int[10000] reverse

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	3'173'200	5'522'300	182'700	2'238'400
First Quartile	3'216'875	5'589'850	184'300	2'262'050
Median	3'241'750	5'659'850	185'150	2'283'250
Third Quartile	3'364'950	5'874'350	191'400	2'334'775
Maximum	3'996'500	7'075'600	360'600	2'993'400
Mean	3'326'011	5'767'847	193'062	2'330'418
Standard Deviation	194'762	267'427	24'166	127'414

Figure 10: Statistics for char[1000] random filled

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	5'300	4'900	3'143'500	2'176'500
First Quartile	5'600	5'000	3'430'950	2'201'675
Median	5'900	5'000	3'466'300	2'232'850
Third Quartile	6'100	5'200	3'621'425	2'311'050
Maximum	7'800	20'900	12'371'100	3'192'100
Mean	5'938	5'240	3'632'348	2'304'471
Standard Deviation	412	1'587	917'065	188'040

Figure 11: Statistics for char[1000] sorted

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	3'026'000	5'668'600	2'472'100	2'252'300
First Quartile	3'243'975	5'907'275	2'581'450	2'264'300
Median	3'369'550	6'042'550	2'660'800	2'285'350
Third Quartile	3'449'875	6'127'150	2'716'450	2'349'400
Maximum	4'446'900	7'159'800	5'337'500	3'551'700
Mean	3'380'301	6'073'995	2'690'105	2'330'183
Standard Deviation	239'388	254'798	294'176	145'974

Figure 12: Statistics for char[1000] reverse

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	379'069'901	615'740'904	13'569'800	229'152'500
First Quartile	382'927'926	620'495'501	13'743'200	231'181'928
Median	384'955'200	623'746'750	13'841'900	232'753'051
Third Quartile	388'717'148	633'483'123	14'255'625	235'296'174
Maximum	462'397'296	662'901'594	16'466'900	267'623'403
Mean	387'923'368	628'272'899	14'071'444	234'717'203
Standard Deviation	10'188'733	11'683'480	494'569	6'317'093

Figure 13: Statistics for char[10000] random filled

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	48'400	48'700	323'990'101	224'363'800
First Quartile	49'275	49'000	355'137'450	228'423'250
Median	50'000	49'101	359'004'702	230'531'351
Third Quartile	56'850	49'800	363'536'175	233'424'025
Maximum	219'700	317'900	434'187'200	266'099'801
Mean	56'367	55'588	360'341'866	231'910'390
Standard Deviation	19'590	30'568	12'265'542	6'738'428

Figure 14: Statistics for char[10000] sorted

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	312'389'200	590'013'899	256'095'700	229'109'400
First Quartile	336'745'153	610'711'406	271'975'373	230'259'900
Median	338'152'251	613'073'049	273'138'102	231'345'401
Third Quartile	341'232'570	619'630'801	275'453'525	234'005'825
Maximum	389'551'398	766'051'997	339'136'699	268'770'300
Mean	340'172'940	618'545'862	275'020'587	233'579'107
Standard Deviation	8'526'306	20'522'701	8'895'466	6'377'468

Figure 15: Statistics for char[10000] reverse

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	2'610'300	3'213'500	1'690'900	2'845'600
First Quartile	2'634'775	3'238'075	1'704'450	2'866'975
Median	2'661'700	3'262'000	1'717'350	2'889'800
Third Quartile	2'745'925	3'330'450	1'733'625	2'955'325
Maximum	4'401'700	3'721'500	2'072'200	3'965'000
Mean	2'729'645	3'300'867	1'731'361	2'950'208
Standard Deviation	220'281	99'204	53'153	165'158

Figure 16: Statistics for bool[1000] random filled

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	5'100	4'900	3'370'800	2'162'700
First Quartile	5'400	4'975	3'406'725	2'184'475
Median	5'600	5'000	3'433'900	2'204'350
Third Quartile	5'800	5'100	3'550'875	2'259'350
Maximum	31'100	5'800	4'131'300	3'552'300
Mean	5'904	5'045	3'509'496	2'275'524
Standard Deviation	2'563	137	159'800	202'116

Figure 17: Statistics for bool[1000] sorted

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	5’300	4’800	3’360’600	2’163’100
First Quartile	5’400	4’975	3’391’150	2’178’075
Median	5’600	5’000	3’421’200	2’192’250
Third Quartile	5’800	5’100	3’481’150	2’223’300
Maximum	29’700	15’400	4’963’100	2’557’900
Mean	6’077	5’185	3’480’994	2’218’274
Standard Deviation	2’909	1’152	192’928	70’408

Figure 18: Statistics for bool[1000] reverse

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	286’545’401	354’424’309	170’943’098	294’936’500
First Quartile	288’952’600	358’075’724	172’831’201	297’155’609
Median	291’277’102	360’008’999	173’552’850	298’901’549
Third Quartile	294’343’949	363’153’200	175’302’776	302’837’055
Maximum	321’725’404	407’870’307	189’177’400	327’962’503
Mean	292’744’878	362’051’335	174’763’763	301’009’971
Standard Deviation	5’751’398	7’252’438	3’302’351	6’138’947

Figure 19: Statistics for bool[10000] random filled

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	47’600	48’600	346’004’989	221’815’600
First Quartile	47’800	48’700	350’191’376	224’643’399
Median	47’900	48’800	352’474’002	226’296’901
Third Quartile	48’200	49’000	357’105’551	229’247’501
Maximum	78’100	110’500	394’554’301	268’939’301
Mean	48’984	50’448	355’011’347	228’229’475
Standard Deviation	4’391	7’227	8’276’809	6’478’175

Figure 20: Statistics for bool[10000] sorted

	BubbleSortWhileNeeded	BubbleSortUntilNoChange	QuickSortGPT	SelectionSortGPT
Minimum	47’600	48’600	341’278’999	220’486’799
First Quartile	47’700	48’700	342’597’625	221’642’850
Median	47’800	48’800	343’736’447	223’276’349
Third Quartile	48’100	48’900	346’826’648	225’416’624
Maximum	89’300	238’600	374’620’700	252’783’701
Mean	49’220	51’643	346’054’110	224’751’007
Standard Deviation	5’780	19’355	6’431’539	5’217’278

Figure 21: Statistics for bool[10000] reverse

4 Discussion

4.1 Compare Hypothesis to Results

Analyzing the results obtained, we find that the performance of the graphs representing the sorting time of the various algorithms is very similar between the tests done on the 1,000-elements and 10,000-elements arrays. So all algorithms suffer from the same "scalability" based on the size of the data to be sorted. This allows us to make a more general analysis since in this respect the algorithms are nearly equivalent.

Our initial hypothesis was formulated considering the average use case of the sorting algorithms, so with a shuffled array.

That said, we had not considered edge cases where one might find oneself having to order arrays that are already sorted, or reverse-sorted.

So analyzing these edge cases as well, we can see that rightly so when dealing with already sorted arrays, the BubbleSortWhileNeeded and BubbleSortUntilNoChange algorithms perform excellently when compared to QuickSortGPT or SelectionSortGPT. But this was to be expected given the very nature of the algorithms, which, precisely, if the array turns out to be already sorted, need only iterate over it without performing any extra operations. However, this case cannot represent a "relevant" example since we imagine there would be no need to sort an already sorted array, and therefore it would not make sense to use the implemented library.

In the case of the revert-sorted array, we lose this "advantage" of the two BubbleSorts over QuickSortGPT and SelectionSortGPT by finding, precisely for the BubbleSorts, equal or worse performance than in the average case (random filled array).
However, we find that in this case, the best performing algorithm would turn out to be SelectionSortGPT. Looking at the values, however, we find that SelectionSortGPT tends to (with the same type of data) have performance that is about 15% better than that of QuickSortGPT; in contrary, with randomly filled arrays, QuickSortGPT has performance that is around 90% better than SelectionSortGPT.

Thus, considering the average use case, that is, the one represented by randomly filled arrays, we find that the best performing algorithm is QuickSortGPT.

This confirms our initial hypothesis that, in average case, the complexity of QuickSort, i.e., $O(n * logn)$, leads it to be the best performing algorithm compared to the proposed competitors.

4.2 Limitations and Threats to Validity

Some of the factors that we can consider as limitations in terms of ensuring absolute validity of our experiment may be the following:

- **Hardware constrains:** We conducted the tests on a single machine; thus, our results are dependent on the hardware's performance, and we lack a comparison with other machines with different hardware configurations. It might have been interesting to perform the same tests on sample machines similar to those on which Bubble Inc. expects its library to be used, in order to ensure that the obtained results are consistent.

- **Background processes:** Even by trying to limit this issue by restarting the machine before each measurement, the number of active background processes and applications can still influence the performance of various algorithms to some extent, potentially leading to different results.

- **Performance measurement:** Using Java's `System.nanoTime()` function to measure sorting time may result in inaccuracies due to the machine's clock and interference from certain background processes. In any case, the potential variations would be minimal so we do not think this would be a real limitation (but it is still a present factor, so we wanted to mention it).

- **Scope of tested cases:** Although we attempted to test many variations of arrays in terms of size and data type, these variations are limited. For a more precise and accurate analysis, we could have tested additional variations in terms of array size and data type. We could also have used different filling patterns for the arrays, mixing portions that are already sorted, reversed, or random.

- **Reliability of measurements:** To ensure a good level of reliability, we performed 100 repetitions of each sorting combination. However, this number of repetitions may still be vulnerable to external biases (such as background processes) and thus might not be sufficiently high. The greater the number of repetitions, the higher the reliability of the average value obtained.
  We decided to keep this value at 100 repetitions because by analyzing the trends in the graphs for the

timing of the various algorithms (as can be seen in chapter *5.3 Additional Figures*), they all appear to be stable and more or less constant from the very first iterations (due to the fact that we performed the warm-up). So 100 seemed to us a good compromise to ensure reliability of the results without repeating measurements that would follow the same trend as those already performed.

4.3 Conclusions

In conclusion, we can confirm that our initial hypothesis, according to which the QuickSortGPT algorithm would perform best on random arrays (thanks to its Average Time Complexity of `O(n*logn)`), has been proven correct.

We also noticed that both variants of BubbleSort perform very well on pre-ordered arrays but poorly on unordered datasets (both random and reversed). However, having pre-ordered datasets is quite an unrealistic scenario, which makes both versions unsuitable for general real-world applications.

Similarly, SelectionSortGPT has a scenario in which it performs the best among all the proposed algorithms, specifically when the dataset is reversed. But again, this is a very unlikely scenario in the real world, making it unsuitable.

Considering the goal of the research, with the aim of implementing this sorting algorithm within a library for Bubble Inc as a "utility tool," and given that we have no control over the datasets it will be used with (thus relying on randomized datasets), we suggest the implementation of the QuickSortGPT algorithm.

5 Appendix

5.1 Materials

The documents we used as reference material for this research are the slides from the "Experimentation and Evaluation" course.

5.2 Reproduction Package

All the resources we used, such as Java files, CSV files from which we generated the charts, Excel files used for chart representation, etc., are all available in the following GitHub repository:
`https://github.com/ghi-la/Experimentation-Evaluation`

5.3 Additional Figures

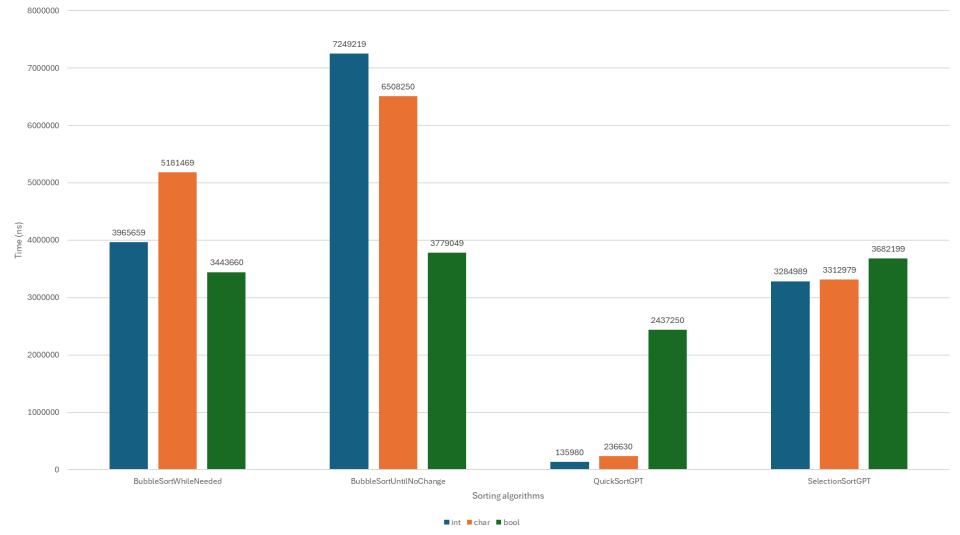


Figure 22: Statistics for the warm up average of a randomly filled array of size 1000

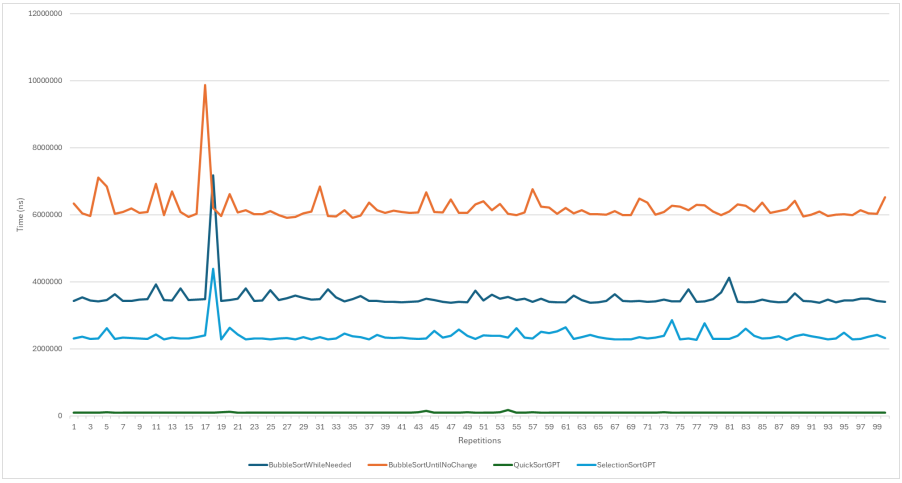


Figure 23: Results of sorting a randomly filled array of size 1000 made from int

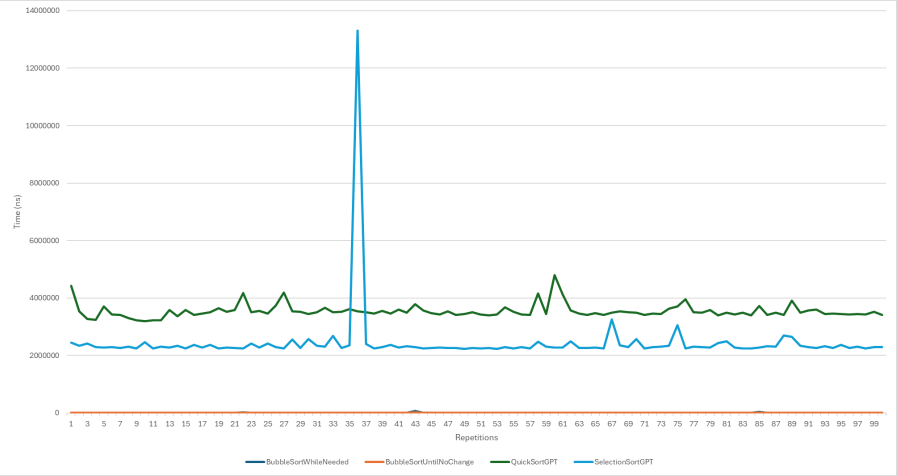


Figure 24: Results of sorting a sorted array of size 1000 made from int

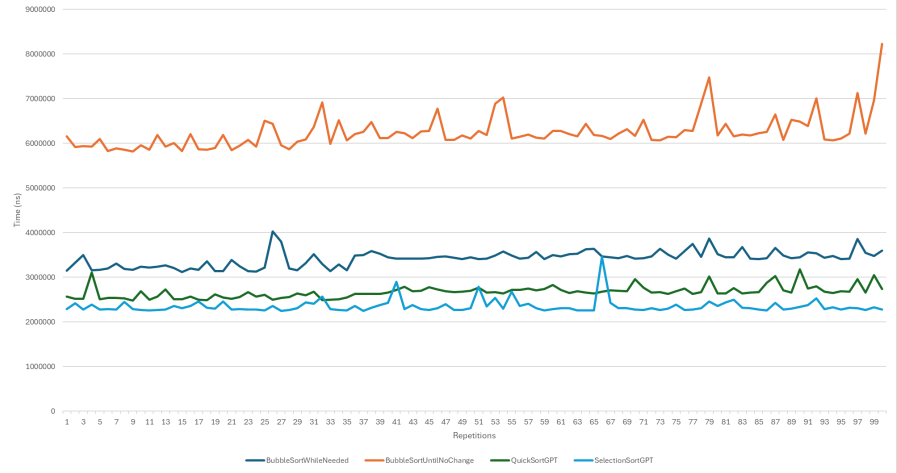


Figure 25: Results of sorting a reverse sorted array of size 1000 made from int

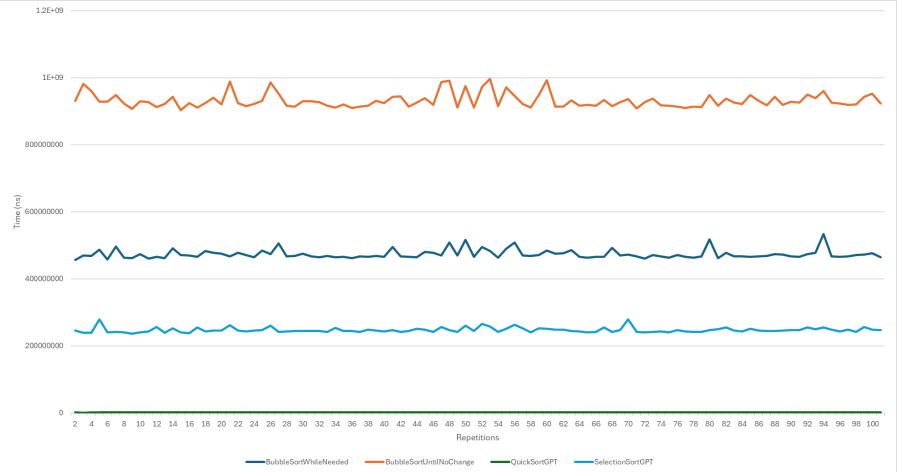


Figure 26: Results of sorting a randomly filled array of size 10000 made from int

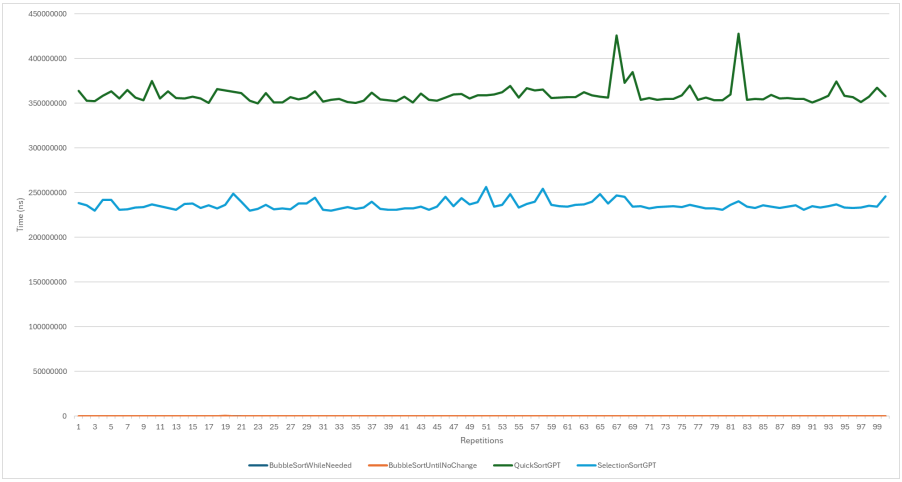


Figure 27: Results of sorting a sorted array of size 10000 made from int

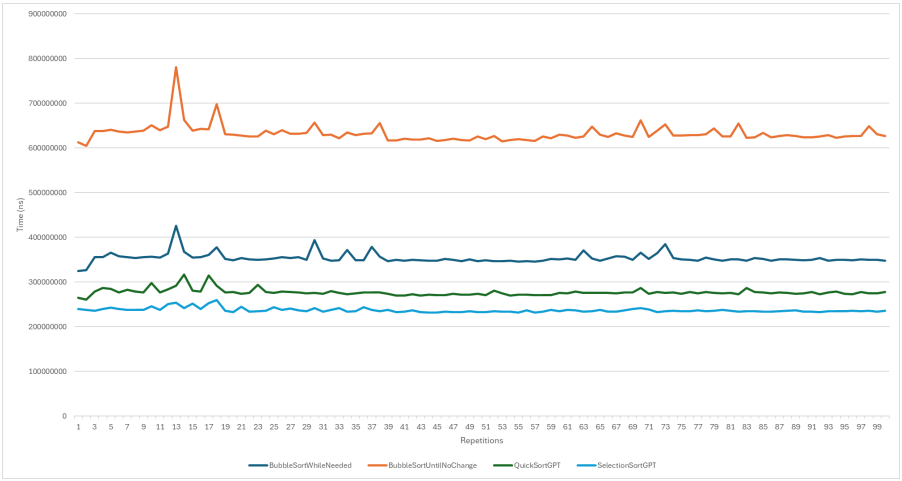


Figure 28: Results of sorting a reverse sorted array of size 10000 made from int

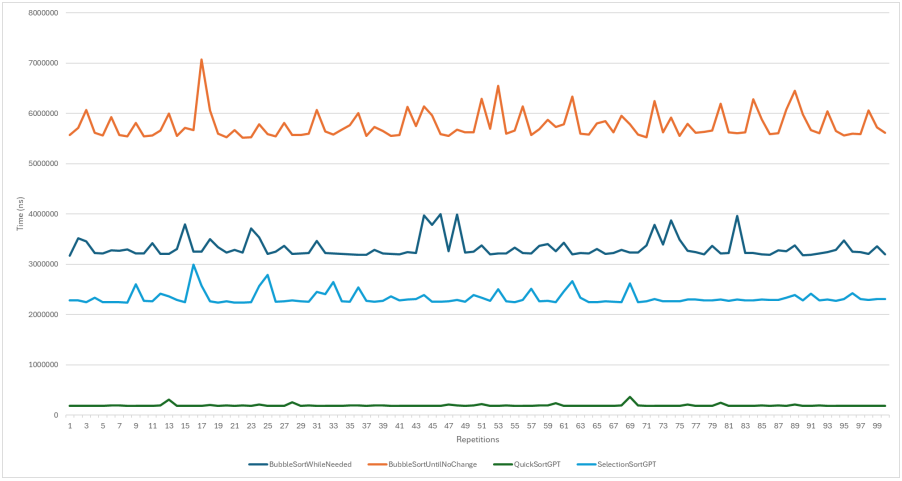


Figure 29: Results of sorting a randomly filled array of size 1000 made from char

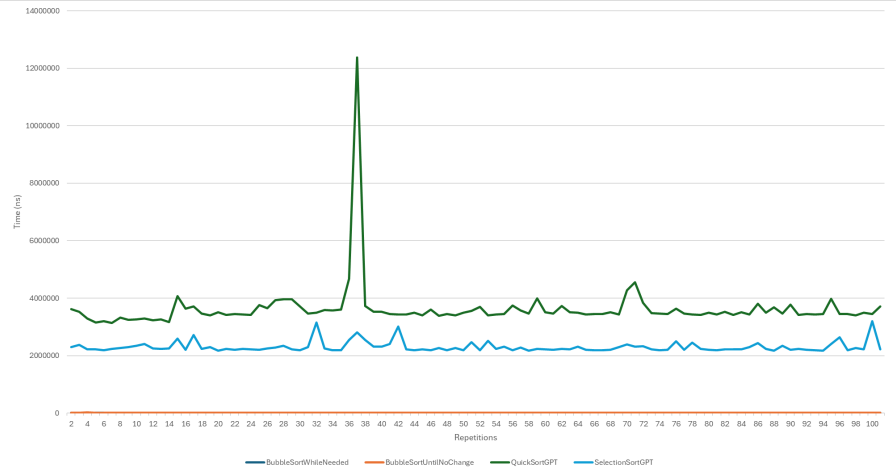


Figure 30: Results of sorting a sorted array of size 1000 made from char

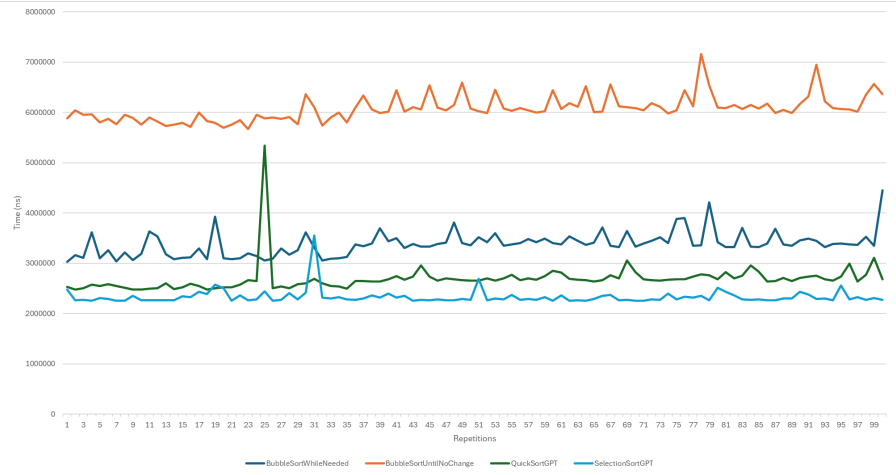


Figure 31: Results of sorting a reverse sorted array of size 1000 made from char

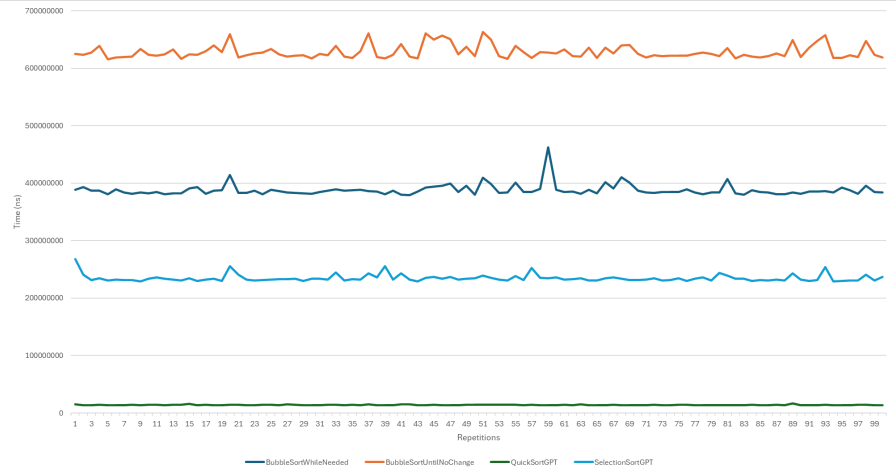


Figure 32: Results of sorting a randomly filled array of size 10000 made from char

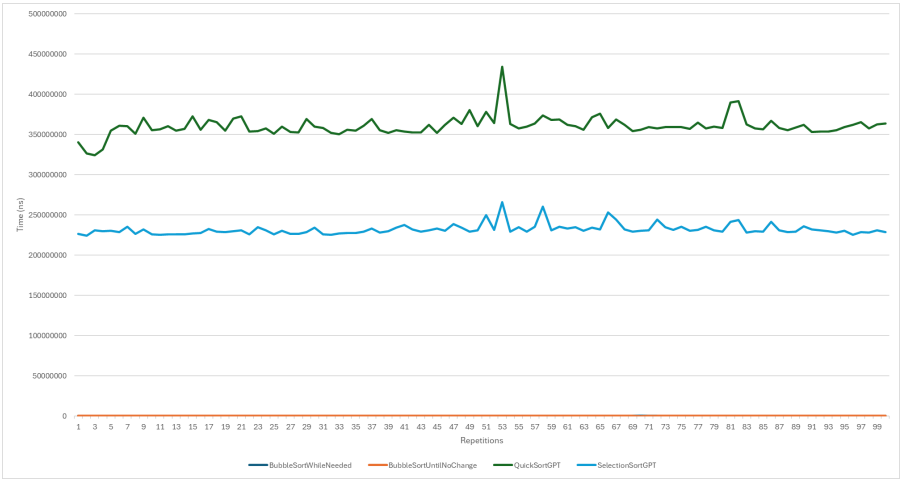


Figure 33: Results of sorting a sorted array of size 10000 made from char

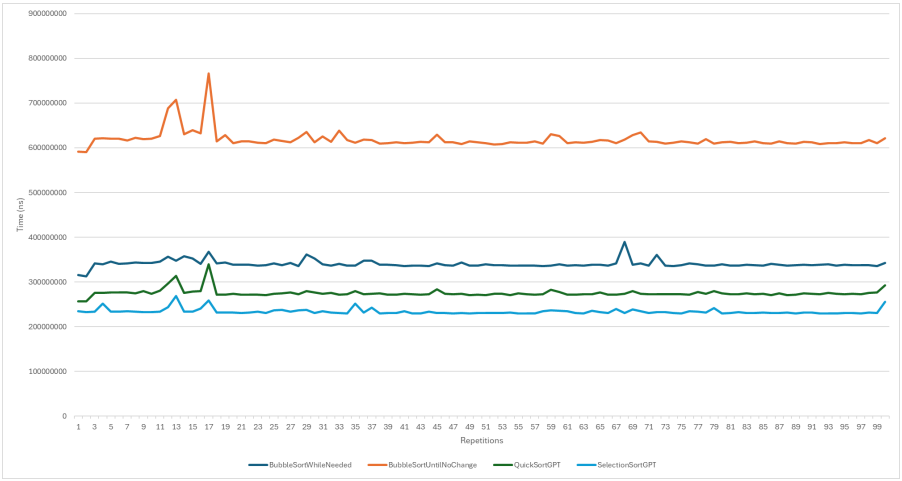


Figure 34: Results of sorting a reverse sorted array of size 10000 made from char

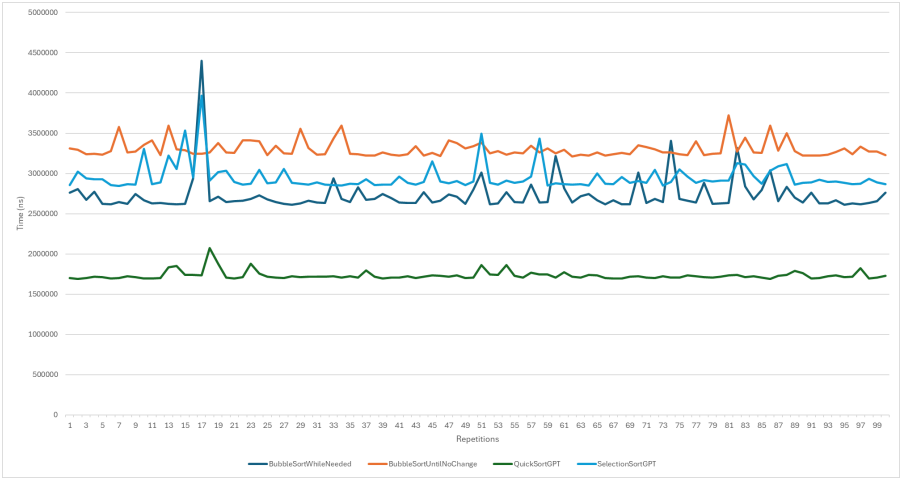


Figure 35: Results of sorting a randomly filled array of size 1000 made from bool

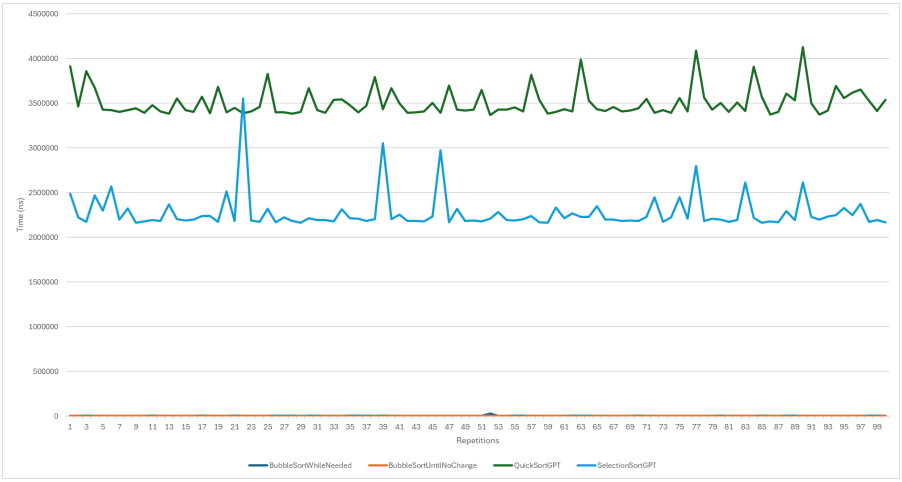


Figure 36: Results of sorting a sorted array of size 1000 made from bool

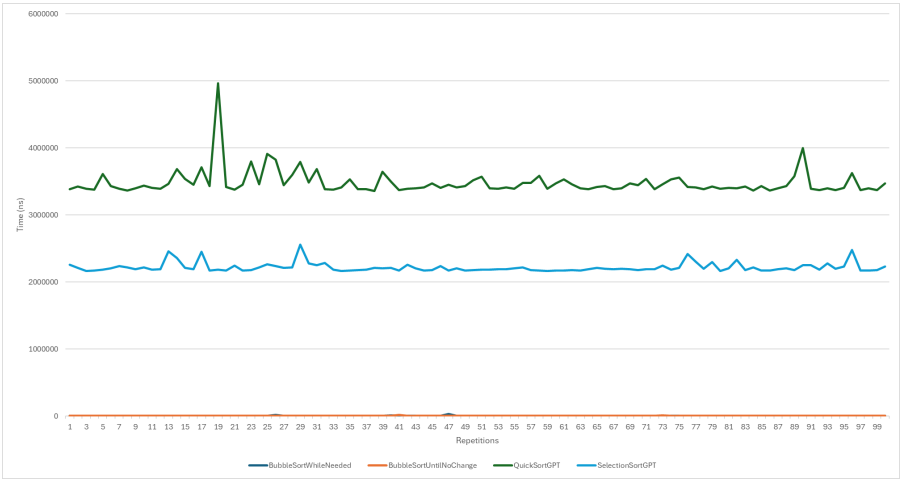


Figure 37: Results of sorting a reverse sorted array of size 1000 made from bool

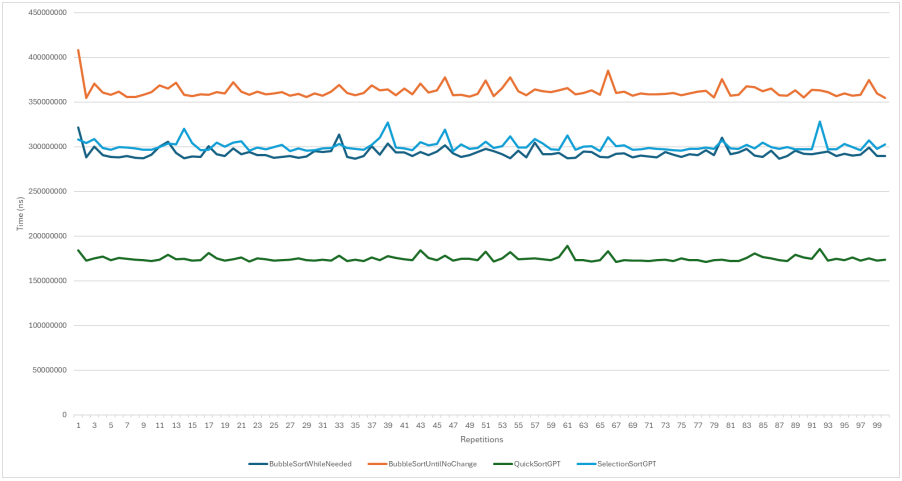


Figure 38: Results of sorting a randomly filled array of size 10000 made from bool

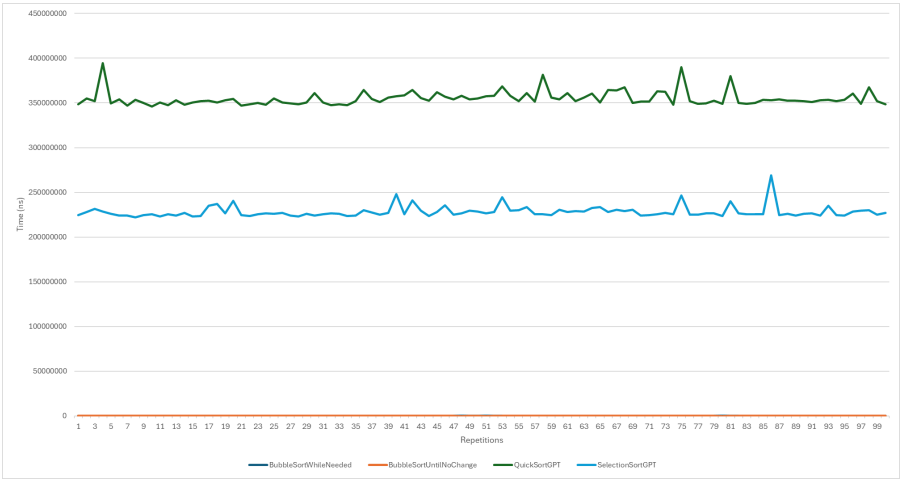


Figure 39: Results of sorting a sorted array of size 10000 made from bool

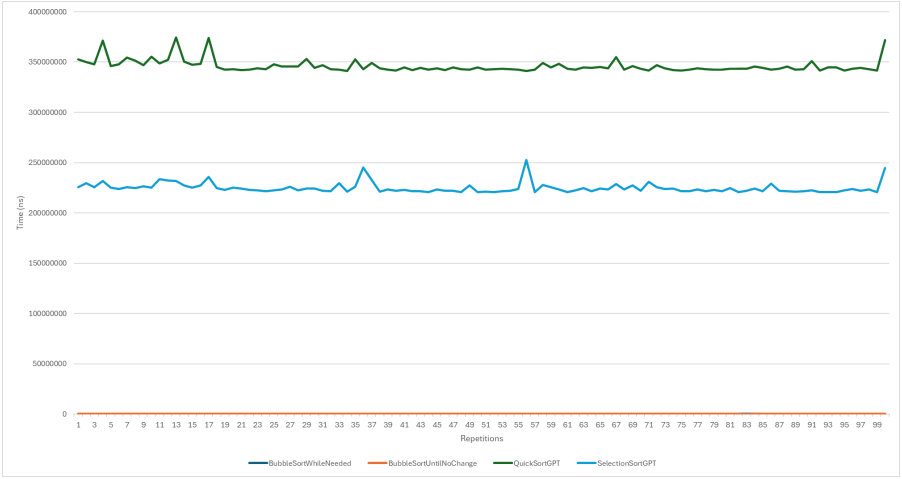


Figure 40: Results of sorting a reverse sorted array of size 10000 made from bool