Bubble Inc.

Experiment 1, Experimentation & Evaluation 2024

Abstract

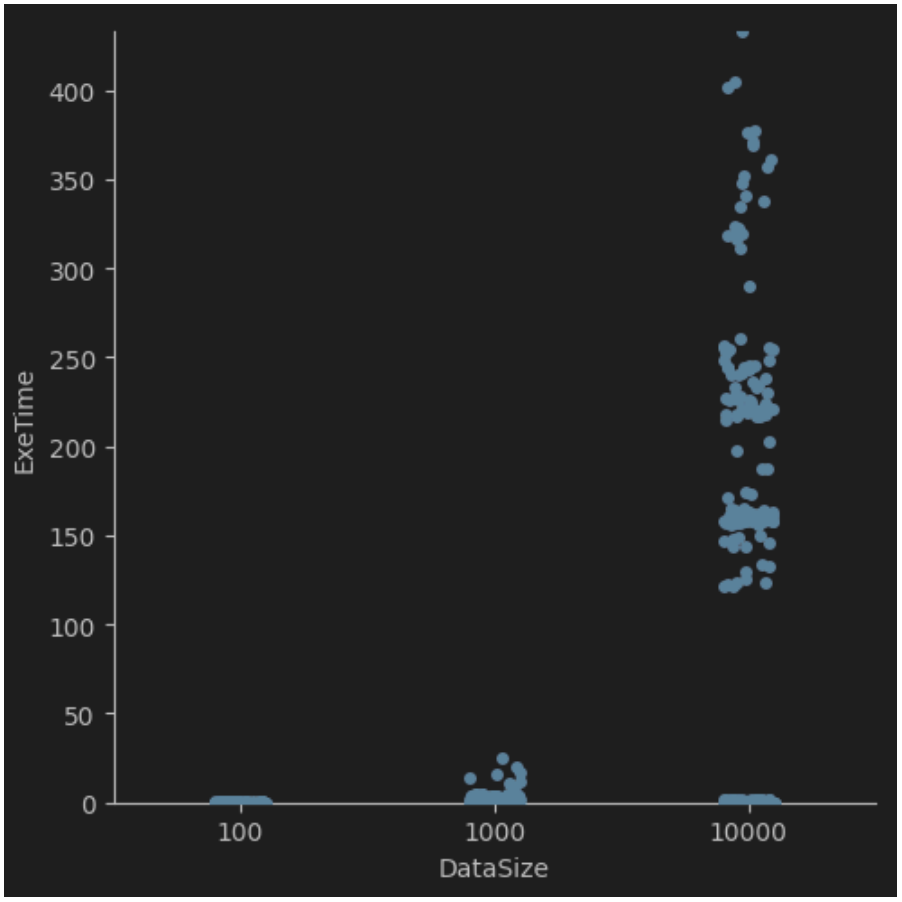
This paper aims at evaluating the performance of the given algorithms BubbleSortUntilNoChange, BubbleSortWhileNeeded, QuickSortGPT, SelectionSortGPT, in order to asses which one is most suitable to include in the final version of their library. The main indicator to assess if one's performance is better than the other is execution time, which will be calculated on a huge amount of instances.

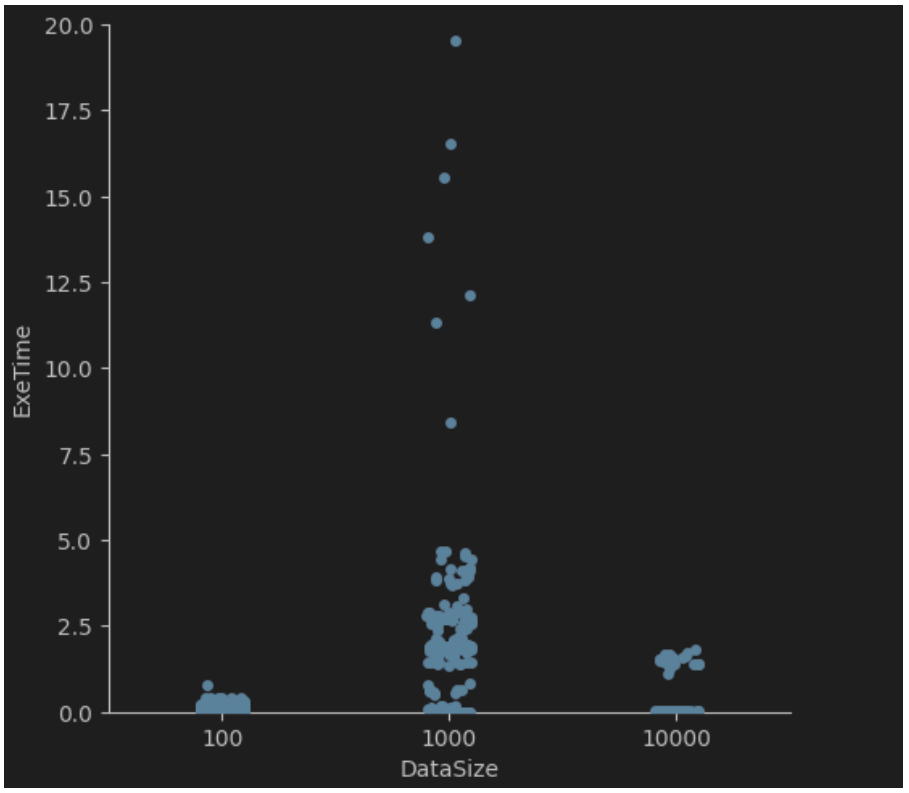
1. Introduction

The proposed experiment will systematically evaluate the execution time of four distinct sorting algorithms: BubbleSortUntilNoChange, BubbleSortWhileNeeded, QuickSortGPT, and SelectionSortGPT. These are sorting algorithms. Sponsored by the company bubble inc, we investigate which algorithm which will be the best performing. The datasets will vary in size (100, 1000, and 10,000 elements) and type (random, sorted, reverse sorted, partially sorted, and containing duplicate values). For each algorithm, execution times will be measured, and the findings will contribute to a deeper understanding of algorithmic performance under varying conditions.

Hypotheses:

1. QuickSortGPT will outperform the other algorithms in execution time for large, randomly ordered datasets.
2. BubbleSortWhileNeeded will show better performance than BubbleSortUntilNoChange across all data types.
3. Also as we can see from the graph below, for small datasets the difference isn't so huge; however, the more the data sizes grow the more you can notice a difference between what we suppose are different algorithms. The following graph plots how our algorithms perform with our data and as we can see in the 10 '000 data set there is a big space between two sets of blue points which we can suppose means that some algorithms are way better than others.





Y-axis is execution time in ms.
X-axis is the size of Data Sizes.

2. Method

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

2.1 Variables

Independent variable	**Levels**				
Array Sizes	100.00		1’000.00		10’000.00
Order of Arrays	Random	Sorted	Reverse Ordered	Partially Sorted	Duplicates Values

Algorithms: The algorithms used are the ones provided by Bubble Inc. and not changed.

Array Sizes: We have three types of data sizes. Let me explain why. First, we don't need sizes which are smaller than 100 because they are very irrelevant, and would not really be important for the scope of this experiment. Also it is very unlikely that sorting algorithms you want to know which algo works best in size are so small because it will likely save you a not valuable amount of time. On the other hand, sizes bigger than 10’000 are not tested on, because from the testing it is already possible to start see the differences of algorithms on a test case of data size of 10’000 so on bigger data size it will just be more marked however we can already draw the conclusion on which algo is gonna perform better on big data sizes by looking at the 10’000 case. So our sizes are 100, 1 '000, 10' 000, which are the most useful sizes in our opinion.

Order of Arrays: We test our instances also on different types of ordered arrays to see if there is any difference, if there is, in doing so, we can suggest to you the best algorithms for the best situation at many different times.

Dependent variable	**Measurement Scale**
Execution Time	The time is in ms

Execution Time: Basically the purpose of our experiment. Execution time is the thing Bubble Inc. asked us to measure so it's the dependent variable.

Control variable	**Fixed Value**
Programming Language	Java
Arrays Data Type	Integers
IDE	Visual Studio Code
Warm-up run time	No Warmup

Machine	MacBook Pro

Programming Language: The programming language used to compute the times is java as requested.
Arrays Data Type: Only int type array are used for this experiment
IDE: The only IDE used for this Experiment was Visual Studio Code.
Warm-up run time: We have decided to not do any warm-up as to us it is very uncommon that people using algorithms do warm up cycles before using it so since we want to give the best recommendation to Bubble Inc. we will test it with 0 warm-up rounds.
Machine: We only use 1 machine because we suppose Bubble Inc. mainly uses 1 type of machine as well to run their algorithms and is the one we are using.

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

Explanation

(1)

Our investigation follows the scientific approach, which emphasises observation, experimentation, and hypothesis validation. We started by defining hypotheses about the performance of different sorting algorithms, namely BubbleSortUntilNoChange, BubbleSortWhileNeeded, QuickSortGPT, and SelectionSortGPT, under various data sizes. The experiment involved manipulating variables, such as the type of algorithm, data type, dataset size, while also measuring the execution time as the independent variable.

This approach focuses on collecting and analysing data, enabling us to understand how factors like data size and arrangement affect sorting performance in real-world applications.

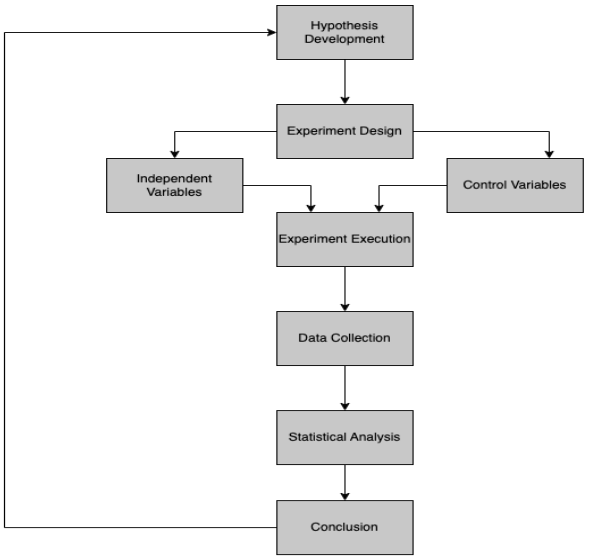
Our study also involves analysis across various data types, namely Random, Sorted, ReverseSorted, PartiallySorted, and DuplicateValues, which provide insights on how different data conditions impact algorithm performance.

For scientific integrity purposes, our experiment is designed in a reproductive way. All the details we provide make sure that other interested researchers can replicate and adapt our study results.

Last but not least, the gathered data is analysed by us to provide a clear overview of trends, while also highlighting how different algorithms behave in various conditions.

The following figure illustrates the sequence of our experimental design.

The diagram starts with "Hypothesis Formation" at the top, leading down to "Experiment Design." This then splits into "Independent Variables" (like algorithm type, data type, array size, and operating system) and "Control Variables" (such as Programming language and machine type). These branches come together at "Execution of Experiment," followed by "Data Collection" and "Statistical Analysis." The flowchart ends with "Conclusion and Interpretation," which loops back to "Hypothesis Formation" to show the repeatable cycle of the scientific method.



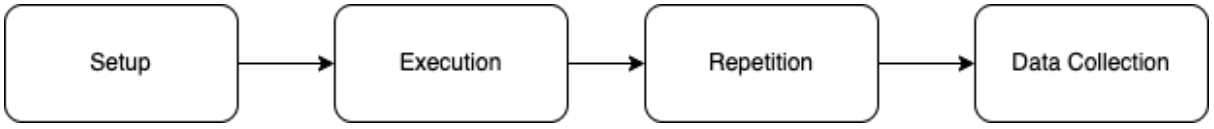
2.3 Apparatus and Materials

CISC Environment	
Computer	Macbook Pro Intel Core i7 processor and 16GB RAM.
Software	Java JDK 17 for Java programs, and VS code for development, testing, and plotting.
Timing Mechanism	System.nanoTime()

Time Mechanism : We use System.nanoTime() but then transform the time in milliseconds(ms)

Also, it is important to mention some libraries that are also used in our project, libraries such as pandas, seaborn and numpy. In order to get the graphs we got, it is essential to install those and then include them in your python or pynotebook file.

2.4 Procedure



1. **Setup:** Each sorting algorithm is implemented in Java, following the Sorter interface from the assignment.
2. **Execution:** Arrays of different sizes and types (random, sorted, reverse sorted, etc.) are generated and sorted by each algorithm. The sorting time is measured using System.nanoTime() and then converted into milliseconds(ms).

3. **Repetition:** Each sorting test is repeated 10 times to account for differences in execution time.
4. **Data Collection and Visualization**: Execution times are recorded in results.csv. The main.ipynb reads the results.csv and plots graphs, comparing performance by data size, type, and algorithm.

1. Environment Setup: The first step is to open SortExperiment.java in either VS Code or IntelliJ, we used VS Code. It will create a separate folder data_csv which will store the csv file. Also ensure that all the sorting algorithms are in fact in the same directory.

2. Before Execution: In SortExperiment.java, the script creates arrays of 3 data sizes which are 100, 1000 and 10000 along with data types, namely Random, Sorted, Reverse Sorted and Partially Sorted as well as Duplicate Values. If changes need to be done, the relevant portions of this code named SortExperiment.java can be changed with respect to the size of the arrays or the data types such as values from the SIZES array and dataTypes array.

3. Execution: Now compile the SortExperiment.java file which runs every sorting algorithm for every size of the array and the specified data types. Each run is timed up with the use of System.nanoTime() and the execution times are stored in a CSV file called results.csv in milliseconds. There are 10 runs done for each type of experiment in order to consider variation in time execution.

4. Repetition: The SortExperiment.java runs every sorting algorithm for 10 times and for 60 combinations comprising 3 data sizes, 5 data types, and 4 algorithms which equals 600 executions of the algorithms altogether.

5. Data Visualization: Upon completion, go to the main.ipynb notebook and construct the charts comparing the performance of algorithms for each algorithm used. The jupyter notebook file uses pandas, matplotlib and seaborn libraries to group and create statistical graphs, thus letting us analyse the execution times. These graphs show algorithm performance based on data size, data type, and execution time, allowing easy comparison across algorithms and configurations.

3. Results

BubbleSortUntilNoChange:

- Scalability: This algorithm scaled quite well as the dataset size increased, with a gradual rise in median execution time compared to others.
- Consistent Performance: It showed a relatively small interquartile range (IQR), suggesting it performed consistently across different data types and sorting orders.
- Efficiency on Small Datasets: BubbleSortUntilNoChange handled smaller datasets efficiently, displaying relatively low median execution times across all data types.

BubbleSortWhileNeeded :

- Outstanding Efficiency: This algo consistently recorded the low median times, being the fastest across 1' 000, and 10' 000 data sizes.
- Scalability: It scaled remarkably well with larger datasets, managing to keep its performance strong and its execution times low.
- Stable Results: The algorithm maintained an average IQR, meaning its performance was somehow relatively changeable depending on the case and predictable no matter the dataset size.

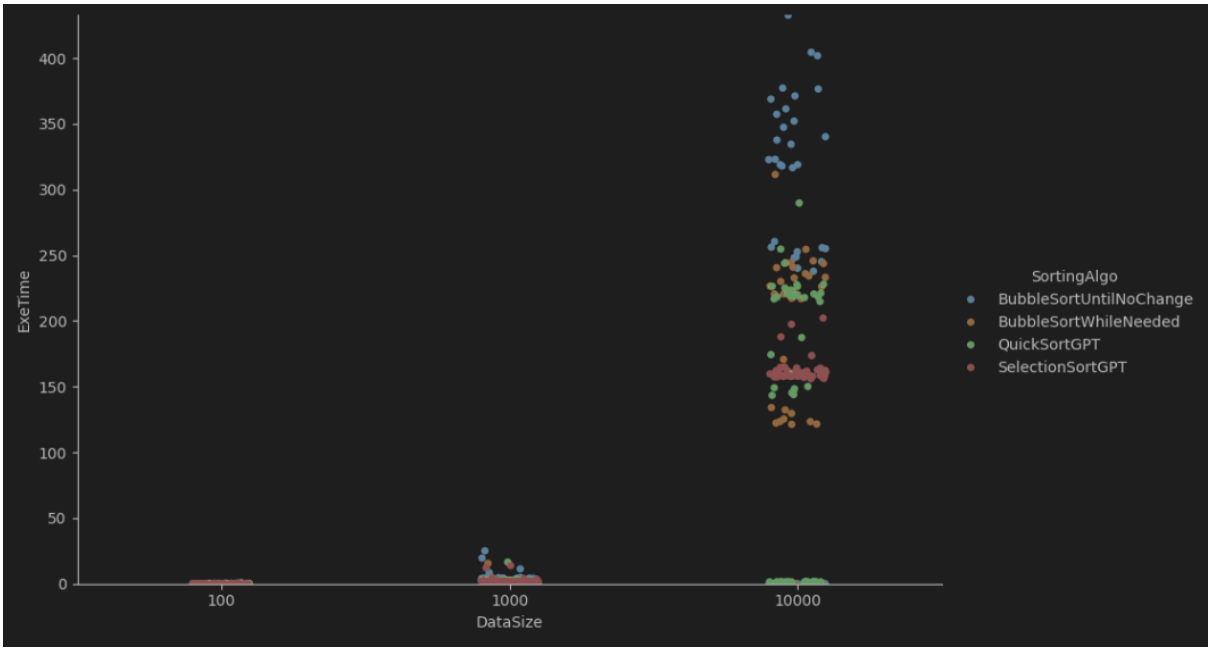
QuickSortGPT:

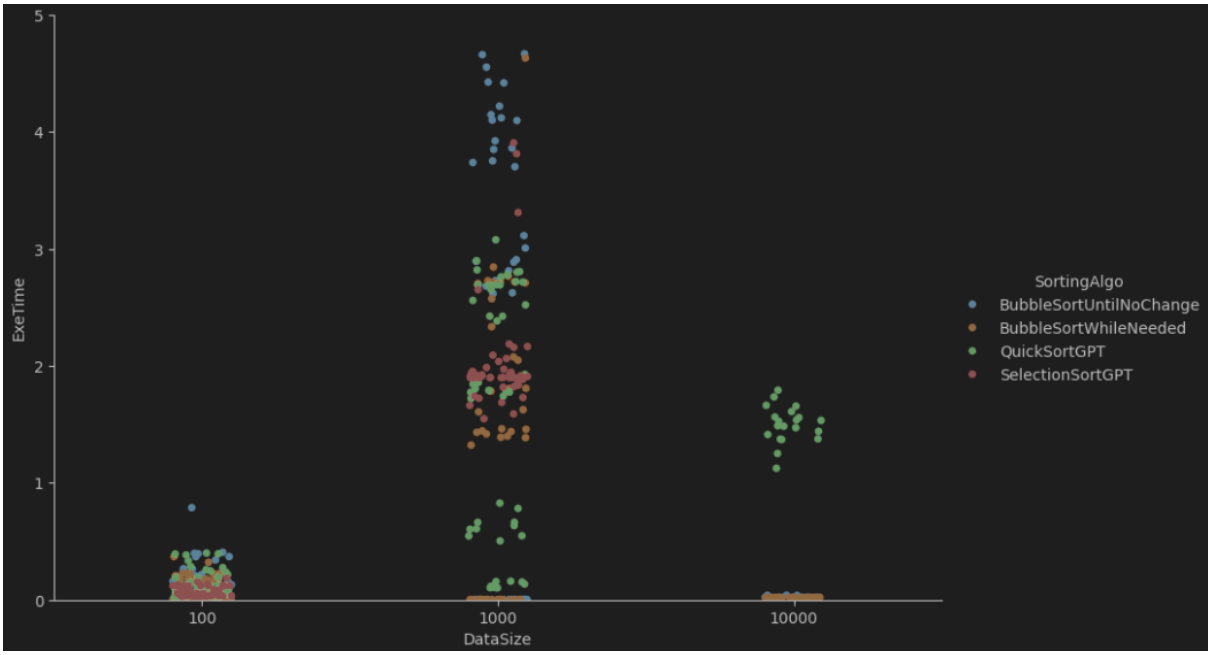
- Moderate Execution Time: QuickSortGPT showed average execution times, which noticeably increased as dataset sizes grew, especially with more complex datasets.
- Variable Range: Its IQR widened with larger datasets and specific data types (e.g., descending order), indicating some variability in execution time.
- Larger Datasets: While the execution time rose with dataset size, QuickSortGPT still scaled better than the BubbleSort algorithms overall.

SelectionSortGPT:

- Mid-Range Performance: This algorithm's execution times were moderate, generally falling between the BubbleSorts variants and QuickSortGPT.
- Consistent Results: SelectionSortGPT had a stable IQR, indicating reliable performance across runs, though the upper execution time limits increased with larger datasets.
- Impact of Data Type: Execution time varied based on the initial ordering of the dataset, with the highest times observed in descending and random orders.

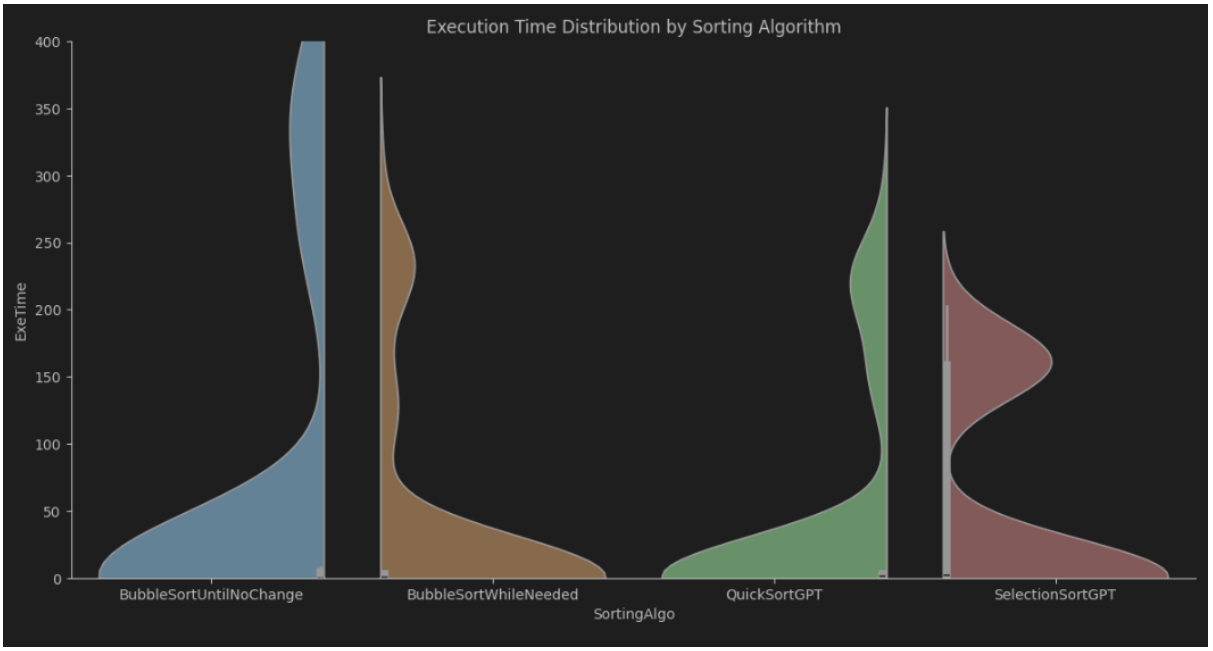
3.1 Visual Overview





The above graph is the same as the one in the hypothesis but zoomed more; outliers which are above 5 of Exetime(ms) are left out of the picture because they are not important.

As we can see from the graph, we were pretty right during our hypothesis in saying that the difference between the points are mainly because different algos are used instead of because different orders are given. However that must influence too as we see agglomerations of points still swing up and down.

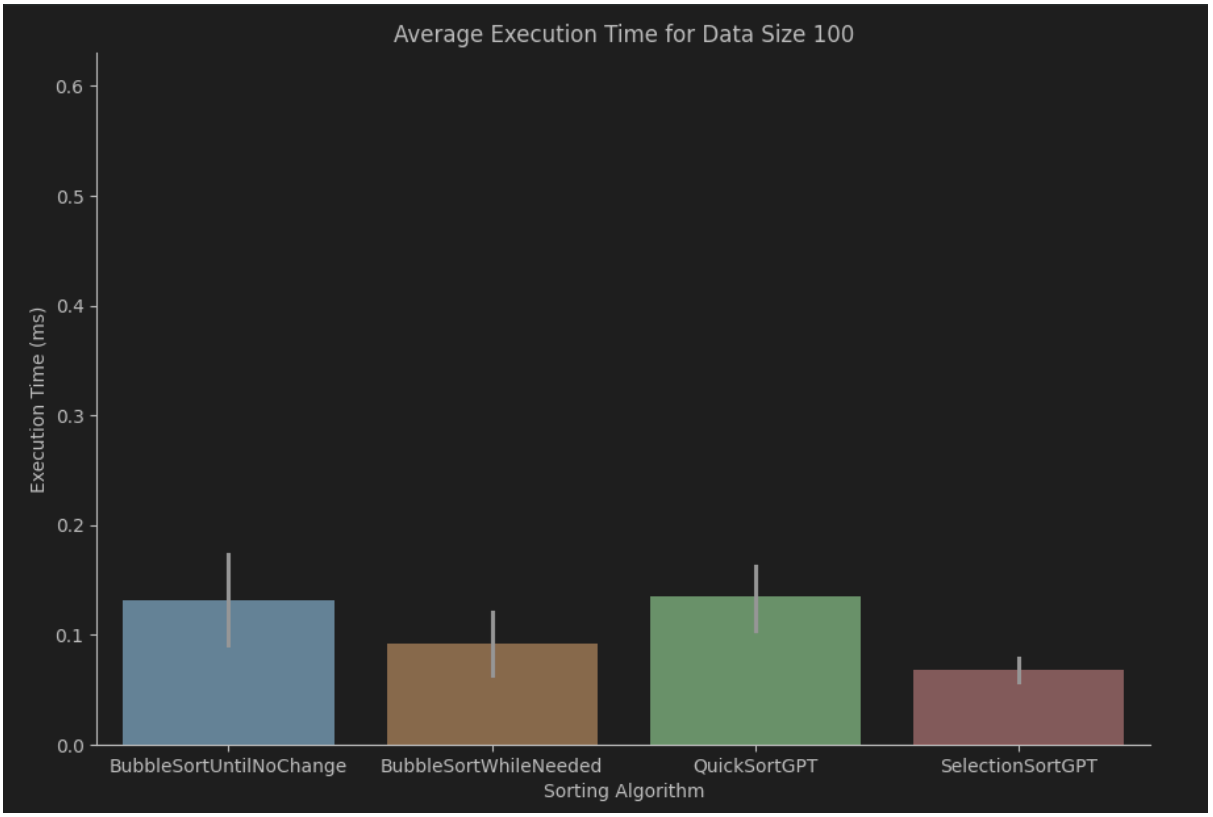


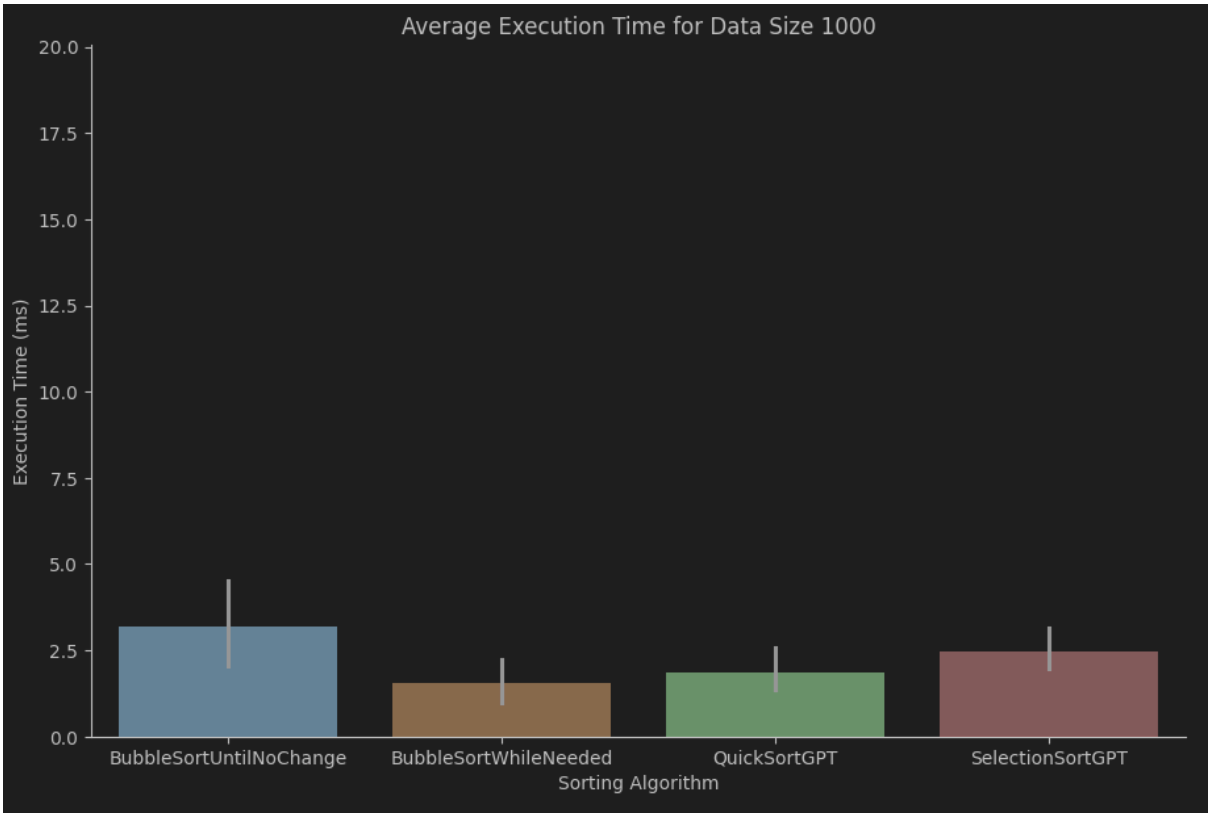
This violin plot gives us a detailed look at how different our sorting algorithms perform by showing how much these execution times can vary. It’s incredibly useful because it lets us see not just which algorithms are faster on average, but also how consistent they are.

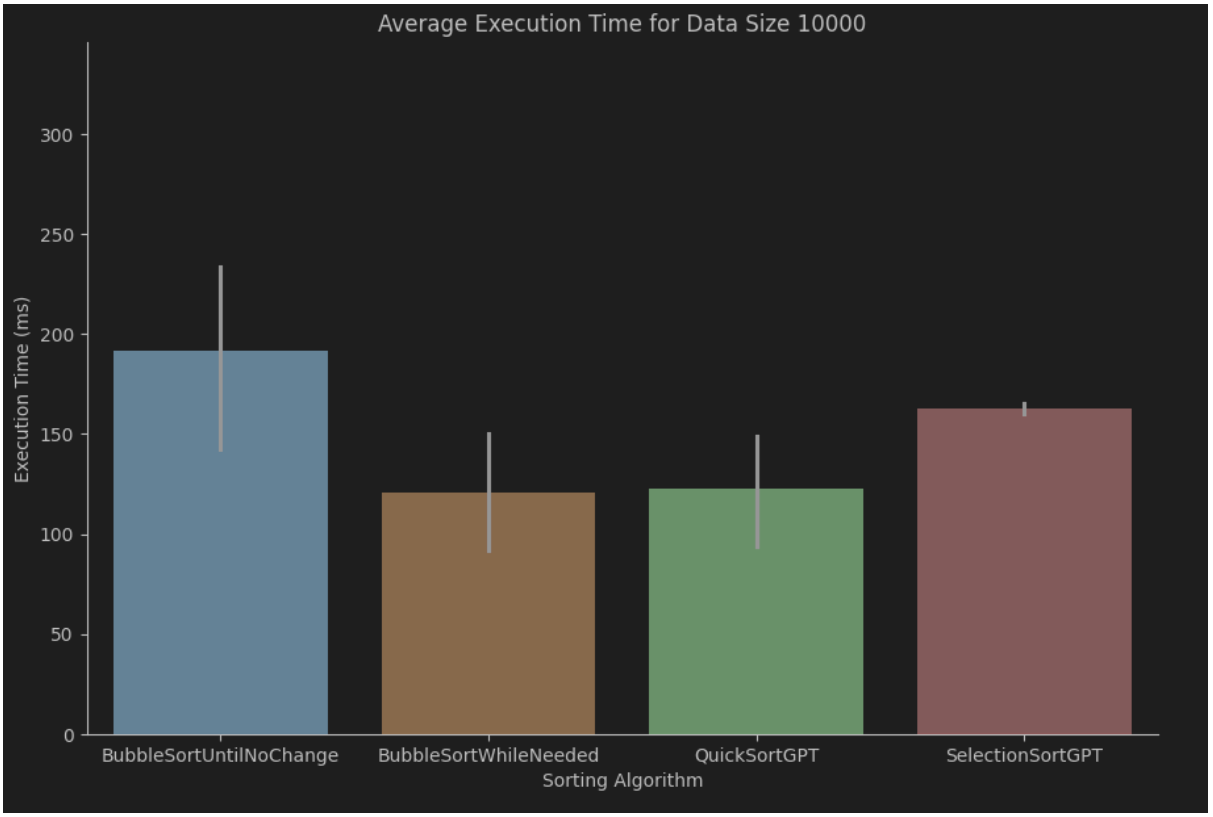
For example, the widespread use of times for BubbleSortUntilNoChange and BubbleSortWhileNeeded suggests that these algorithms may be more sensitive to specific types of data, resulting in fluctuating performance.

Also, in this graph all datasets are combined together meaning the wideness close to 0 execution time is due to 100 and some of 1 '000 data sizes and the higher ones are of course outliers and the 10’ 000. This graph is useful to see how they generally perform and it's important to notice how wide is the graph the higher the execution time because that means a lot of points are present in that range.

Having this information is important for anyone choosing the best sorting algorithm, particularly in applications where consistent performance matters just as much as speed. While the BubbleSort variants show a tendency to spike in execution time under certain conditions, QuickSortGPT seems to offer more stable performance. This insight can help developers pick an algorithm that best fits their specific needs, balancing both efficiency and reliability.


Average Execution Time for Data Size 100
Execution Time (ms)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
BubbleSortUntilNoChange
BubbleSortWhileNeeded
QuickSortGPT
SelectionSortGPT
Sorting Algorithm


Average Execution Time for Data Size 1000
Execution Time (ms)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
BubbleSortUntilNoChange
BubbleSortWhileNeeded
QuickSortGPT
SelectionSortGPT
Sorting Algorithm



These bar plots give us a clear look at the average execution time of different sorting algorithms across three dataset sizes: 100, 1’000, and 10’000 items. They help reveal how each algorithm’s performance changes as the dataset size increases, showing efficiency or better how much the algo is efficient compared to the others.

For the smallest dataset size of 100 items, all algorithms perform relatively quickly, with execution times well below 1 millisecond. At this scale, BubbleSortWhileNeeded and SelectionSortGPT stand out as the fastest, while BubbleSortUntilNoChange and QuickSortGPT are slightly slower but still in a similar range.

When we move up to the dataset size of 1’000 items, the differences between the algorithms start to become more noticeable. BubbleSortWhileNeeded maintains efficient performance with a lower average execution time, and QuickSortGPT also continues to perform well. However, BubbleSortUntilNoChange shows a slight increase in execution time. SelectionSortGPT remains relatively stable but doesn’t perform as quickly as BubbleSortWhileNeeded and QuickSortGPT at this mid-sized dataset.

For the largest dataset size of 10,000 items, the differences in performance become more visible. BubbleSortUntilNoChange sees an increase in execution time, reaching around 200 milliseconds, which indicates scalability issues. BubbleSortWhileNeeded performs significantly better than BubbleSortUntilNoChange, but it still doesn’t match the performance of QuickSortGPT and SelectionSortGPT. In this large dataset scenario, QuickSortGPT and SelectionSortGPT achieve the lowest execution times, with QuickSortGPT leading slightly.

3.2 Descriptive Statistics

<table>
<tr><td>Algorithms</td><td>Execution time in ms.</td></tr>
<tr><td>BubbleSortUntilNoChange</td><td>
<table>
<tr><td></td><td>Min</td><td>1st Q</td><td>Median</td><td>3rd Q</td><td>Max</td></tr>
<tr><td>100</td><td>0.001260</td><td>0.002080</td><td>0.077363</td><td>0.196786</td><td>0.788071</td></tr>
<tr><td>1’000</td><td>0.003056</td><td>0.003424</td><td>2.850073</td><td>4.099855</td><td>25.07573</td></tr>
<tr><td>10’000</td><td>0.022007</td><td>0.022887</td><td>250.7776</td><td>336.9177</td><td>432.7150</td></tr>
</table>
</td></tr>
<tr><td>BubbleSortWhileNeeded</td><td>
<table>
<tr><td></td><td>Min</td><td>1st Q</td><td>Median</td><td>3rd Q</td><td>Max</td></tr>
<tr><td>100</td><td>0.001088</td><td>0.002906</td><td>0.042574</td><td>0.179014</td><td>0.369680</td></tr>
<tr><td>1’000</td><td>0.002405</td><td>0.002905</td><td>1.425447</td><td>2.270302</td><td>15.53412</td></tr>
<tr><td>10’000</td><td>0.020931</td><td>0.021307</td><td>124.4408</td><td>229.2133</td><td>311.4547</td></tr>
</table>
</td></tr>
<tr><td>QuickSortGPT</td><td>
<table>
<tr><td></td><td>Min</td><td>1st Q</td><td>Median</td><td>3rd Q</td><td>Max</td></tr>
<tr><td>100</td><td>0.012514</td><td>0.051254</td><td>0.100030</td><td>0.191701</td><td>0.400076</td></tr>
<tr><td>1’000</td><td>0.102566</td><td>0.562022</td><td>1.788097</td><td>2.690953</td><td>16.53178</td></tr>
<tr><td>10’000</td><td>1.125301</td><td>1.542993</td><td>148.7726</td><td>219.9297</td><td>289.7843</td></tr>
</table>
</td></tr>
<tr><td>SelectionSortGPT</td><td>
<table>
<tr><td></td><td>Min</td><td>1st Q</td><td>Median</td><td>3rd Q</td><td>Max</td></tr>
<tr><td>100</td><td>0.031183</td><td>0.034040</td><td>0.052609</td><td>0.111306</td><td>0.185034</td></tr>
<tr><td>1’000</td><td>1.548210</td><td>1.894836</td><td>1.908195</td><td>1.982463</td><td>13.82694</td></tr>
<tr><td>10’000</td><td>156.0657</td><td>158.1741</td><td>160.0343</td><td>162.6052</td><td>202.2458</td></tr>
</table>
</td></tr>
</table>

This table provides five-numbers (min, 1st quartile, median, 3rd quartile, max) for each sorting algorithm across different dataset sizes (100, 1000, and 10,000 items), giving us a straightforward look at the range and times (in milliseconds) for each algorithm’s performance.

- **Minimum (Min)**: This is the fastest execution time recorded for each algorithm and dataset size. It represents the best-case scenario, where the algorithm completed its task in the shortest possible time. For instance, BubbleSortUntilNoChange has a minimum time of 0.001260 ms on the 100-item dataset.
- **First Quartile (1st Q)**: This figure marks the point where 25% of the recorded times fall below it. It indicates the lower end of typical performance, showing where faster times tend to be. For example, with BubbleSortWhileNeeded on the 100-item dataset, 25% of its runs finished in under 0.002906 ms.
- **Median**: The median is the midpoint, with half of the recorded times below it and half above it. This represents a "typical" execution time for the algorithm under normal conditions. For example, QuickSortGPT has a median time of 148.7726 ms on the 10,000-item dataset.
- **Third Quartile (3rd Q)**: This value shows the point below which 75% of the execution times fall, representing the upper end of typical performance. It indicates where usually slower times begin to appear. For example, SelectionSortGPT has a third quartile of 162.6052 ms for the 10,000-item dataset.
- **Maximum (Max)**: This is the longest recorded execution time, showing the worst-case scenario for each algorithm and dataset size. It represents the worst case scenario in our dataset and it could possibly be an outlier but still it is important to look at. For example, BubbleSortUntilNoChange has a maximum of 432.7150 ms on the 10,000-item dataset.

Altogether, these five numbers give us a clear overview of each algorithm's performance range, from the fastest to the slowest, and where most of the times are at. This summary does not just show us average performance; it highlights the variability and occasional outliers in each algorithm's execution times across different dataset sizes.

4. Discussion

4.1 Compare Hypothesis to Results

1. QuickSortGPT

   Hypothesis: We expected QuickSortGPT would be the fastest on large, random datasets.

   Results: This was mostly true. QuickSortGPT handled larger datasets better than the BubbleSort algorithms, with faster times as data size grew. It scaled more effectively than the BubbleSort methods, which slowed down with bigger data. However, QuickSortGPT's speed wasn't as steady on structured data, especially in descending order, where its times varied more. Still, it remained one of the top choices for large data, as expected.

2. BubbleSortWhileNeeded

   Hypothesis: BubbleSortWhileNeeded is expected to do better than BubbleSortUntilNoChange on all data types.

   Results: This was supported. BubbleSortWhileNeeded had lower median times than BubbleSortUntilNoChange and was the fastest for the 1,000 and 10,000 data sizes. It scaled well with larger datasets and had steady results, although its IQR did vary with data type. By contrast, BubbleSortUntilNoChange showed slower increases in time as data grew. Overall, these results back up the idea that BubbleSortWhileNeeded is better for various data types.

3. Data Size Impact

   Hypothesis: We expected small performance differences on smaller datasets, with bigger gaps for larger ones.

   Results: This turned out to be correct. Smaller datasets showed only slight differences among given algorithms, but as we reached the 10,000 mark, it is clear from the tables above that the performance gap between algorithms became much more noticeable, which was not the case with the 100 data size set.

Discrepancies:

QuickSortGPT's times became less consistent with larger data, especially in descending order, where it slowed down noticeably. In contrast, SelectionSortGPT showed steadier performance across different data types, although its upper times also increased with bigger datasets.

4.2 Limitations and Threats to Validity

While this study offers insights into sorting algorithm performance, several factors might affect the wider relevance of the results:

1. **Hardware and Software Constraints**

   Description: Testing was done on only one system, specifically macOS.

   Impact: Results might differ on other platforms, like Windows or Linux, where system resources and processes could affect performance.

   Solution: Future studies should consider testing on multiple operating systems to check if results hold across different setups.

2. **Programming Language and Environment**

   Description: The sorting algorithms were tested solely in Java within VS Code.

   Impact: This setup might influence results, as other languages or development environments could handle algorithms differently.

   Solution: Expanding to other languages and IDEs would help verify whether performance trends are consistent across various programming contexts.

3. **Data Variety and Scale**

   Description: The study included specific data types and sizes, like random and descending orders, but didn't test all possible scenarios.

   Impact: The findings might not capture how these algorithms perform on very large datasets or other data structures.

   Solution: Adding a broader range of data types and sizes would give a broader picture of each algorithm's strengths and limitations.

Threats to Validity

These limitations suggest that the results might not generalise to all setups or real-world cases. Future studies could address these points to make the findings more broadly applicable.

4.3 Conclusions

The key conclusions drawn from this study are:

Top for Large Datasets: QuickSortGPT and SelectionSortGPT worked best on large datasets. QuickSortGPT was fast with random data but slowed with descending. Both did much better than BubbleSorts on bigger data, making them strong choices for large-scale sorting.

Reliable for Medium Datasets: BubbleSortWhileNeeded did better than BubbleSortUntilNoChange across all sizes and held up well with mid-sized data. Its median value was lower than those of QuickSortGPT and SelectionSortGPT on the largest datasets, but it was steady and reliable.

Best for Small Datasets: BubbleSortUntilNoChange had trouble scaling with big data but worked fine with smaller sets. It's best used when you're working with limited data sizes.

Choosing the Right Algorithm: This study shows each algorithm has strengths depending on data size. QuickSortGPT and SelectionSortGPT work best for large data, BubbleSortWhileNeeded for medium, and BubbleSortUntilNoChange for small.

In conclusion: We suggest to implement in your library BubbleSortWhileNeeded because given all this information in our opinion is the algorithm which perform best for you situation, one key factor in deciding this was that this algo has the lowest median for the 1'000 and 10'000 arrays datasets, making it the most stable algorithm out of 4 tested ones.

Summary: This study breaks down how each algorithm handles various amounts of data. With this information, developers and researchers can choose the best option to get fast and reliable results for what they are working on.

Appendix

A. Materials

Following are present all the materials needed to replicate the experiment, the consent form by Bubble Inc. and the informations provided by them:

```java
public final class BubbleSortWhileNeeded<T extends Comparable<T>> implements Sorter<T> {

    public void sort(final T[] items) {
        int n = items.length;

        do {
            int maxIndex = 0;
            for (int i = 1; i < n; i++) {
                if (items[i - 1].compareTo(items[i]) > 0) {
                    final T item = items[i - 1];
                    items[i - 1] = items[i];
                    items[i] = item;
                    maxIndex = i;
                }
            }
            n = maxIndex;
        } while (n > 0);

    }

}

```

```java
interface Sorter<T extends Comparable<T>> {

    void sort(T[] items);


}
```

```java
public final class SelectionSortGPT<T extends Comparable<T>> implements Sorter<T> {

    public void sort(final T[] items) {
        if (items == null || items.length == 0) {
            return;
        }

        int n = items.length;
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (items[j].compareTo(items[minIndex]) < 0) {
                    minIndex = j;
                }
            }

            if (minIndex != i) {
                swap(items, i, minIndex);
            }
        }
    }

    private static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

}
```

```java
public final class BubbleSortUntilNoChange<T extends Comparable<T>> implements Sorter<T> {

    public void sort(final T[] items) {
        boolean changed;
        do {
            changed = false;
            for (int i = 0; i < items.length - 1; i++) {
                if (items[i].compareTo(items[i + 1]) > 0) {
                    final T item = items[i];
                    items[i] = items[i + 1];
                    items[i + 1] = item;
                    changed = true;
                }
            }
        } while (changed);
    }

}
```

```java
public final class QuickSortGPT<T extends Comparable<T>> implements Sorter<T> {

    public void sort(final T[] items) {
        if (items == null || items.length == 0) {
            return;
        }
        quickSort(items, low:0, items.length - 1);
    }

    private static <T extends Comparable<T>> void quickSort(T[] array, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(array, low, high);
            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }

    private static <T extends Comparable<T>> int partition(T[] array, int low, int high) {
        T pivot = array[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (array[j].compareTo(pivot) <= 0) {
                i++;
                swap(array, i, j);
            }
        }

        swap(array, i + 1, high);
        return i + 1;
    }

    private static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

}
```

Consent Form for Use of Data and Sharing of Information

Bubble Inc. Consent Form

We, the undersigned, representing Bubble Inc., hereby grant Federico Bonezzi and Polad Bakhishzade,
university students, and any other researchers or participants wishing to replicate this study, the consent to utilize data generated for Bubble Inc. for research purposes. We understand that this data
may be shared with others within the academic and research community for the purpose of analysis,

replication, and verification of results.

The consent extends to the collection, processing, analysis, and distribution of data and findings derived from this study. By signing below, we acknowledge our understanding and agreement to the

use and sharing of our data in accordance with these terms.

Furthermore, Bubble Inc. understands that any personal or sensitive data will be anonymized and handled
in compliance with applicable data protection laws.

Signature: Bubble Inc. Authorized Signature

Date: 2024-10-31

B. Reproduction Package (or: Raw Data)

All the data is present in the reproduction package submitted together with this report.