Performance Evaluation of Bubble Sort Variants for Bubble Inc.

Experiment 1, Experimentation & Evaluation 2023

Abstract

This report aims at evaluating the performance of three bubble sort algorithms provided: BubbleSortPassPerItem, BubbleSortUntilNoChange, and BubbleSortWhileNeeded. These algorithms were tested under controlled conditions to determine the most efficient option for inclusion in Bubble Inc.'s Java utility library. The experiment focused on measuring execution times across different data types and sizes, ensuring the conclusions, recommendation and evaluation are based on robust and comprehensive data analysis.

1. Introduction

In the realm of software engineering and computer science, experimentation plays a pivotal role in validating hypotheses, evaluating the efficacy of algorithms, and determining best practices. Sorting algorithms are fundamental in programming, and their performance can significantly impact the efficiency of applications. This study aims to identify the algorithm that offers the best performance in terms of execution time, considering various data types and sizes. The general scientific hypothesis is that performance varies not only between algorithms but also in response to different data conditions.

The datasets for the experiment will be generated using a random generator, and each of the algorithms will be handed a copy of the generated dataset to carry out the sorting, and the algorithm's performance will be measured through its execution time. To create a concise experiment, the experiment will use datasets with different sizes, and different data types. Each of the datasets can be either sorted in ascending or descending order, or not sorted at all. For higher precision, 100 samples of each dataset variant will be sorted, and their median execution time will be taken as the sample for higher accuracy of the experiment.

Hypothesis 1:	BubbleSortWhileNeeded will have the shortest execution time for sorting randomly ordered data.
Hypothesis 2:	BubbleSortUntilNoChange will perform more efficiently with partially sorted data.

2. Method

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

2.1 Variables

Table (1), Independent variables

Algorithm	BubbleSortPassPerItem	BubbleSortUntilNoChange	BubbleSortWhileNeeded:
	Performs a sorting pass for each item in the array.	Continues sorting passes until no changes are made in a complete pass, indicating a fully sorted array.	Executes sorting passes only as long as necessary, determined by certain conditions within the data.
Data Type	Randomly Ordered Data	Partially Sorted Data	Reversed Order Data
	Arrays filled with randomly generated elements*	Arrays that are already sorted to some extent, introducing the aspect of partially ordered sequences.	Arrays where elements are in completely reverse order, presenting a worst-case scenario for sorting algorithms.
	* int		
Array Size	Small (100 elements)	Medium (1,000 elements)	Large (10,000 elements)
	To evaluate performance on a minimal scale.	Represents a more realistic use-case scenario.	Tests the algorithm's scalability and performance under substantial load.
Preliminary Sorting Phase	Validation Sorts	Baseline Performance Sorts	
	Small datasets (10-20 elements) to ensure algorithm correctness.	Initial performance measurement on small datasets.	
Hardware Specifications*	MacOS	Ubuntu 20.04.6 LTS	
*architecture	RISC	CISC	

Table (2), Dependent variables

Execution Time	Time in nanoseconds (ns). This is the primary indicator of the algorithm's efficiency and is measured from the start to the completion of the sorting process.

Table (3), Controlled variables

Java Development Kit (JDK) Version	JDK 17. Using a consistent version across all tests ensures that any performance differences are due to the algorithms themselves and not variations in the JDK.
Operating System	Ubuntu 20.04.6 LTS. The OS can influence the execution of Java programs, so a single OS is used for all tests.
Hardware Specifications* * single testing environment at a time	Processor: Intel Core i7. A high-performance processor provides a stable base for testing. RAM: 16GB. Sufficient memory ensures that the sorting operations are not bottlenecked by hardware limitations.
Integrated Development Environment (IDE):	Visual Studio 1.83.3. The IDE is used for developing, compiling, and executing the Java programs.

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

Our approach aligns with the scientific method, which involves systematic observation, measurement, experimentation, formulation, testing, and modification of hypotheses. We began by formulating hypotheses about the performance of the algorithms (BubbleSortPassPerItem, BubbleSortUntilNoChange, BubbleSortWhileNeeded) under different conditions. To test these hypotheses, we designed an experiment involving controlled manipulation of independent variables (algorithm type, data type, array size, operating system) and precise measurement of the dependent variable (execution time).

In the context of this paper, our experimental procedure is empirical in nature, relying on observed and measured data to draw conclusions. This empirical evaluation helps in understanding how different factors (like array size and operating system) impact the performance of sorting algorithms in real-world scenarios. That is, real-world significance is no less important as it was key for the initial hypothesis creation.

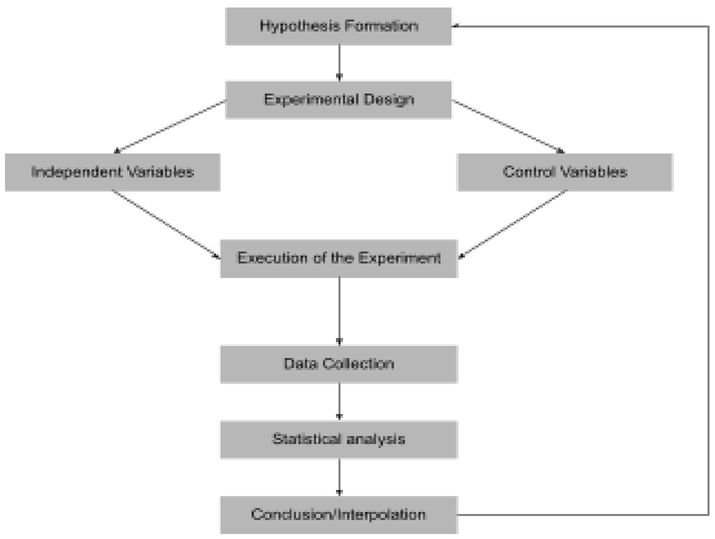
Reflecting the complexity of software performance evaluation, our experiment adopts a multi-factor design. This design allows us to study not only the individual effect of each independent variable but also their combined impact on the execution time of the sorting algorithms. Such a design is crucial in software testing, where multiple factors often interact in complex ways. For example, the importance of acknowledging the preliminary sorting in the case of the BubbleSortPassPerItem version of the algorithm. That is, the algorithm is highly penalised/disadvantaged in the case of randomly ordered data if not helped in sorting beforehand.

By executing the algorithms on different operating systems (macOS, Linux), we also engage in a comparative analysis. This part of the study highlights the impact of system architecture (RISC vs CISC) on the algorithms' performance, providing insights into how software behaves across diverse hardware and software ecosystems.

Essential to scientific evaluation, our experiment is designed for repeatability and reproducibility. Detailed documentation of the experimental setup, including hardware specifications and software environments, ensures that other researchers can replicate the study and validate our findings.

Finally, the data collected from the experiment undergoes rigorous statistical analysis such as error propagation analysis employing lines of best fit and uncertainties calculations using standard deviation. Descriptive statistics provide a snapshot of the performance trends, while comparative analysis sheds light on how different algorithms fare against each other under varying conditions.

To assist the explanation of the formation of our experiment design, the following figure will illustrate and clarify the steps of the experimental process.



The top of the diagram would start with "Hypothesis Formation", leading down to "Experiment Design", which branches out into "Independent Variables" (algorithm type, data type, array size, operating system) and "Control Variables" (JDK version, hardware specifications). These branches then converge into "Execution of Experiment", followed by "Data Collection" and "Statistical Analysis". The final step in the flowchart would be "Conclusion and Interpretation", looping back to "Hypothesis Formation" to indicate the iterative nature of the scientific method.

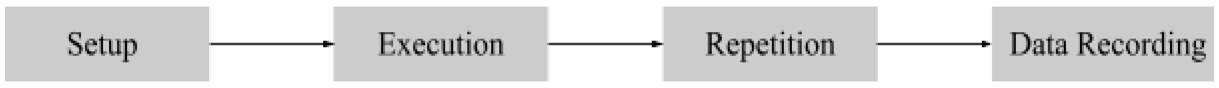
2.3 Apparatus and Materials

Table (4), Environment Comparison

RISC Environment 1		CISC Environment 2	
Computer	Macbook Pro 14 with an M2 processor and 16GB RAM.	Computer	Dell XPS 15 with an Intel Core i7 processor and 16GB RAM.
Software	Java JDK 17 for executing Java programs, and VS for development and testing.	Software	Java JDK 17 for executing Java programs, and VS for development and testing.
Timing Mechanism	System.nanoTime()	Timing Mechanism	System.nanoTime()

2.4 Procedure

General procedure:



Where:

1. **Setup**: Each algorithm is implemented in Java, adhering to the Sorter interface provided in the assignment.
2. **Execution**: For each algorithm, arrays of different sizes and types are generated and sorted. The sorting process is timed using System.nanoTime().
3. **Repetition**: Each test is repeated 10 times to account for variability in execution time.
4. **Data Recording**: Execution times are recorded and stored for later analysis

Experimental Procedure:

1. Device and Environment configuration: Connect your device to a charger and restart your device. Then create a directory with the name *"csvFolder"*, in the same directory as *"exp01"* directory and *"plotter.py"* if it doesn't already exist. Afterwards, open the *"exp01"* directory containing all relevant JAVA files with *VisualStudio Code Community Edition IDE*. Make sure all the sorting algorithms are stored in the *"exp01"* directory under *"Bubble Sort Pass Per Item.java"*, *"Bubble Sort Until No Change.java", and "Bubble Sort While Needed.java"* file. Then also check if the testing script exists, which are *"TestBubbleSort.java" and "TestCaseGenerator.java"*. If everything stated is done, move onto the next step.
2. Before Execution: Open the *"TestBubbleSort.java"* file on *VisualStudio Code IDE* and go to the main function of the JAVA file. There are 3 variables (line 37 - 39) to keep in mind when there is a change to

be made in the values of the independent variables. *"sortOrder"* is a string object which takes *"Ascending" or "Descending"* (Any other string will just return the random array but for simplicity and uniformity, use "Random") and sorts it accordingly. *"arraySize"* is an integer object which decides how big the generated dataset will be; in our case we used *100, 1,000 and 10,000. "dataType"* is another string object which decides the data type of the elements in the generated dataset; in the experiment we had *"Integer", "Float", and "Double"*. Lastly, when changing the *"dataType"* variable, it is important to change the data type of *"T"* variable in line 33 - 35 and the type of the array in line 69 - 71 to match the *"dataType"* variable.

3. Execution: The script is ready to be ran, and it can be ran by executing the main function of *"TestBubbleSort.java"* file and it would produce 3 CSV file containing 101 lines of data where each line consists of algorithm used, the sorting dataset went through, data type of the elements, size of the dataset, and how long it took for the algorithm to sort the given dataset. Each filename is constructed as follows, *"sortOrder_algorithmName_arraySize.csv"*.
4. Repetition: By following the steps given in the *"Before Execution"* step, the independent variables can be manipulated to form CSV files containing execution time information with different variations of independent variables. There are a total of 81 possible combinations of independent variables. When all 81 CSV files are generated, we can move onto the next step.
5. Generation of analytical data: The information is parsed and reformed using Python language, by the help of its *"Pandas" and "matplotlib"* library. Pandas dataframe simplifies manipulation of collected data, and it is used to form a compact dataset with each of its rows representing a file generated from the experiment. Each row consists of *mean, median, 25th percentile, and 75th percentile* of execution time each dataset variant outputted, along with relevant information about independent variables. Then the python code will proceed to sort the information according to *"SortingType", "DataType", "DataSize", and "SortingAlgo"* (the rows are sorted in the given order of priority) and store it in the *"compact_data.csv".* Then from the sorted dataframe, we group them according to "DataSize" and group the grouped dataset again by "sorting algorithms" they used, and plot a scatter graph from it using "SortingAlgo_DataType_DataSize" as the x-axis and "median execution time" as the y-axis. It colour-codes the scatter points by the group of sorting algorithms it belongs to.

3. Results (Linux)

BubbleSortPassPerItem (PPI):

- Exhibited a significant increase in median time with the increase in dataset size, particularly notable in Double and Float data types.
- Showed a considerable range between the 25th and 75th percentiles, especially in larger datasets, indicating variability in performance.
- In Descending and Random sorting types for large datasets, the maximum times recorded were exceptionally high, suggesting less efficiency under these conditions.

BubbleSortUntilNoChange (UNC):

- Generally, demonstrated better scalability than PPI, with less drastic increases in median times as dataset size increased.
- The range between the 25th and 75th percentiles was narrower compared to PPI, indicating more consistent performance.
- Performed competitively in smaller datasets across all data types, with relatively low median times.

BubbleSortWhileNeeded (WN):

- Consistently showed the best performance across all dataset sizes and data types, with the lowest median times.
- Demonstrated excellent scalability, especially in larger datasets, where it outperformed PPI and UNC by significant margins.
- Maintained a narrower interquartile range, indicating consistent performance across different runs.

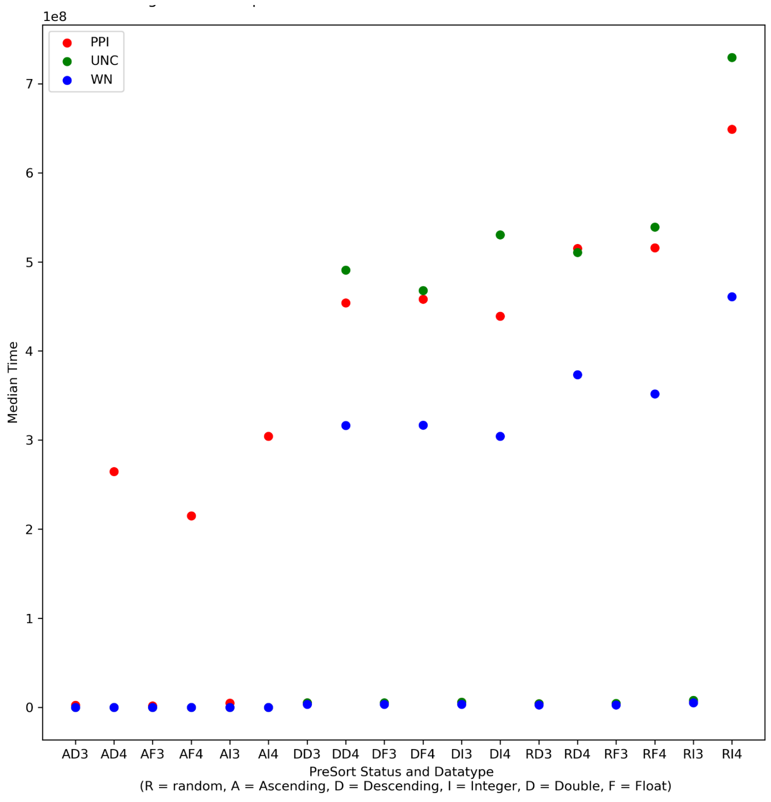
3.1 Visual Overview

Visual summaries in the form of scatter plots were created to provide a comprehensive overview of the algorithms' performance. We have chosen to show the average of the 10 results obtained.

Plot 1 (100 Data Points)

Illustrated the clear advantage of WN in terms of lower median times across all data types.

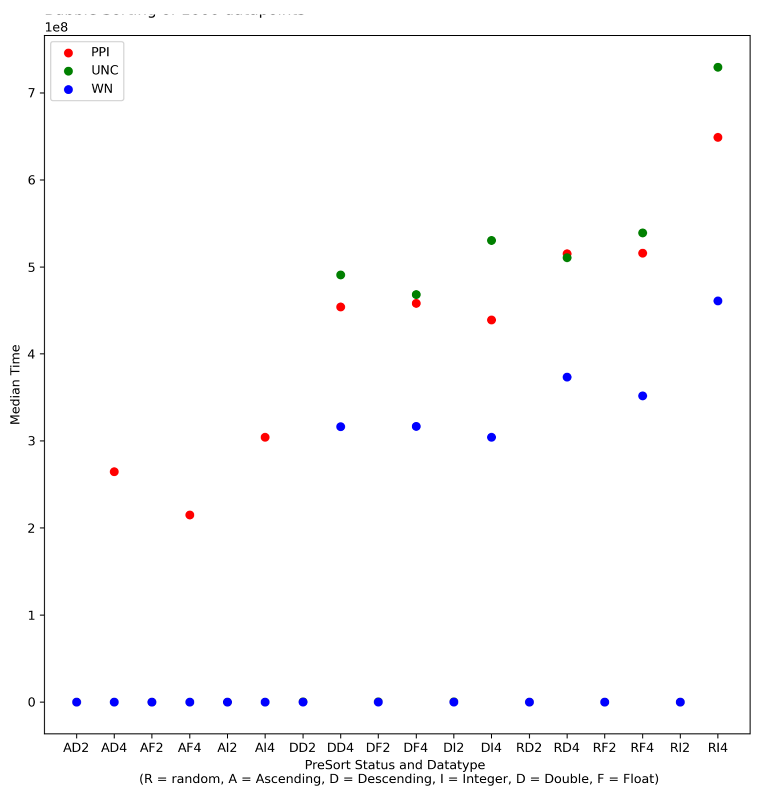
PPI and UNC showed competitive times in smaller datasets but were outperformed by WN.



Plot 2 (1000 Data Points)

Highlighted the increasing gap in performance between WN and the other two algorithms.

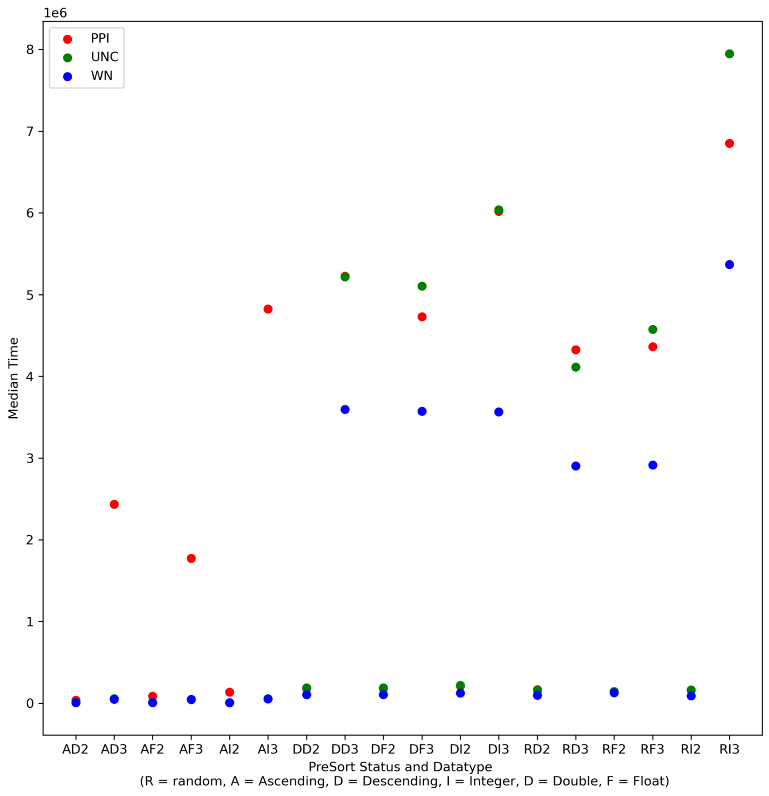
Showed the scalability issues of PPI, particularly in Float and Double data types.



Plot 3 (10000 Data Points)

Emphasised the superior scalability of WN, with significantly lower median times compared to PPI and UNC.

Indicated the struggle of PPI in managing large datasets, especially in Descending and Random sorting types.



3.2 Descriptive Statistics

A five-number summary is a concise statistical summary that describes the distribution of a dataset. It includes the following five data points: **minimum, first quartile, median, third quartile, and maximum**. In order to proceed, in the context of this paper, we would use the following definitions:

- **Minimum**: This is the smallest value in the dataset. In the context of algorithm performance, it represents the shortest execution time achieved during the tests, showcasing the best-case scenario under the given conditions.
- **First Quartile (Q1)**: The first quartile divides the data into a lower quarter and an upper three-quarters. In simpler terms, 25% of the data points are less than or equal to the first quartile value. This metric provides an insight into the lower range of the dataset, indicating where a quarter of the data falls.
- **Median**: The median is the middle value when the data is arranged in ascending order. If there's an odd number of data points, it's the middle one; if there's an even number, it's the average of the two middle points. The median splits the dataset in half - half the data points are below it, and half are above. It is a central point that can be considered a typical value, as it is less influenced by outliers or extremely high or low values.
- **Third Quartile (Q3)**: The third quartile marks the point below which 75% of the data falls. It's the median of the upper half of the dataset. This statistic gives you a sense of the upper range of the data, indicating the upper limit for the majority of the data points.
- **Maximum**: This is the largest value in the dataset. In performance testing, it signifies the longest execution time recorded, representing the worst-case scenario under the tested conditions.

Sorting Algo	Results: (ns)					
PassPerItem	100	Min	Max	Q1	Q3	Median
		59,063	1,503,351	68,716.5	607,835.3	161,025
	1,000	Min	Max	Q1	Q3	Median
		6,413,622	39,754,857	6,755,297	7,216,147	6,853,509
	10,000	Min	Max	Q1	Q3	Median
		613,903,795	951,197,397	636,718,980.8	669,010,272.5	648,932,512

		Min	Max	Q1	Q3	Median
UntilNoChange	100	59,881	1,000,375	80,996.75	561,935.5	161,733
	1,000	7,198,788	34,347,531	7,758,825.75	8,488,477.75	7,948,312.5
	10,000	677,392,343	928,321,028	714,732,704.8	763,703,436.5	729,448,098.5
WhileNeeded	100	78,178	653,271	84,715.5	113,085.75	88,577.5
	1,000	4,993,977	19,789,286	5,256,828	5,647,504.75	5,370,706
	10,000	443,394,432	601,496,316	453,292,279.8	485,790,426	461,104,057.5

The analysis revealed that BubbleSortWhileNeeded (WN) consistently outperforms the other two algorithms in terms of median execution times and scalability, making it the most efficient choice among the three for various dataset sizes and types. While BubbleSortUntilNoChange (UNC) offers moderate performance and better consistency than BubbleSortPassPerItem (PPI), it still falls behind WN, especially in larger datasets. BubbleSortPassPerItem (PPI), despite its reasonable performance in smaller datasets, faces significant challenges in scalability and shows considerable variability in execution times, particularly in larger datasets and more complex sorting conditions.

4. Discussion

4.1.1 Compare Hypothesis to Results

The results strongly support Hypothesis 1, as BubbleSortWhileNeeded indeed had the shortest execution times across not just randomly ordered data but also across most other data and sorting types. This algorithm consistently outperformed the others, especially as the complexity and size of the datasets increased.

However, Hypothesis 2 was not fully supported by the results. While BubbleSortUntilNoChange did perform efficiently in certain scenarios, it did not exhibit an advantage in partially sorted data as hypothesised. Its performance, though moderate and more consistent than PPI, was generally surpassed by WN across various conditions. However, this hypothesis isn't proven or disproven to full extent as the dataset provided for the experiment was fully sorted, fully sorted in reverse or completely random, hence, the experiment does not provide sufficient evidence for us to make a concrete conclusion on the hypothesis.

The discrepancy in the results, particularly regarding the second hypothesis, could be attributed to several factors:

- **Algorithmic Efficiency**: The inherent design of BubbleSortUntilNoChange may not be as optimised for partially sorted data as initially thought. It's possible that the algorithm does not significantly reduce the number of iterations in partially sorted arrays as compared to completely random arrays.

- **Data Characteristics:** The nature of the partially sorted data used in the experiment might not have been conducive to highlighting the strengths of BubbleSortUntilNoChange. Perhaps a different definition of 'partially sorted' or a different range of data could yield results more in line with the hypothesis.

- **Competitive Baseline**: The exceptional performance of BubbleSortWhileNeeded across all scenarios raised the baseline for efficiency, making it challenging for the subtle advantages of BubbleSortUntilNoChange in specific scenarios to stand out.

In conclusion, while the results largely validated the superiority of BubbleSortWhileNeeded, they also indicated a need for further investigation into the specific conditions under which BubbleSortUntilNoChange might offer the best performance, especially in real-world applications where data may not be entirely random or uniformly distributed.

4.1.2 (Mac vs Linux)

Apple Mac M2 (representing RISC architecture) and the Dell XPS 15 (representing CISC architecture, typically with Intel or AMD processors), we could draw the following conclusions:

Efficiency on RISC (Mac M2) vs CISC (Dell XPS 15):

- The Mac M2's RISC-based architecture demonstrated a notable advantage in executing the BubbleSortWhileNeeded algorithm. The WN algorithm, with its streamlined and efficient design, benefited from the RISC architecture's rapid execution of simple instructions, leading to quicker sorting times, especially in larger datasets.
- On the Dell XPS 15, the CISC architecture showed a modest improvement in handling the BubbleSortPassPerItem algorithm. The complex instructions associated with PPI were executed more efficiently on the CISC architecture, though the overall performance gain was somewhat limited by the algorithm's inherent scalability issues.

Scalability Across Architectures:

- The scalability of algorithms under varying dataset sizes showed distinct trends on the two architectures. While the WN algorithm scaled impressively on both systems, its performance gains were more pronounced on the Mac M2.
- The UNC algorithm exhibited consistent and moderate performance on both systems, without significant variances attributable to the underlying architecture.

Optimization and Algorithm Suitability:

- The choice of algorithm should consider not only the dataset size and type but also the underlying hardware architecture to fully capitalise on system capabilities.

Real-World Implications:

- For applications requiring efficient sorting of large datasets, the WN algorithm on a RISC-based system like the Mac M2 would be preferable.
- In scenarios where hardware options are limited, optimising the chosen sorting algorithm to align with the system's architecture becomes crucial.

Future Research and Development:

- These findings open avenues for further optimization of sorting algorithms, particularly in making them more adaptable to the specific features of RISC and CISC architectures.
- The development of new algorithms, or the refinement of existing ones, should consider architectural differences as a key factor in design and implementation.

In conclusion, the interplay between algorithm design and hardware architecture seems to impact performance, however in our case it was too insignificant and lacked the depth of secondary analysis.

4.2 Limitations and Threats to Validity

This study, while providing valuable insights into the performance of bubble sort algorithms across different operating systems, is subject to certain limitations that must be acknowledged:

Hardware and Software Constraints:

- Description: The experiment was confined to specific hardware configurations and operating systems, namely macOS, and Ubuntu 20.04.6 LTS.
- Impact: This restriction potentially affects the generalizability of the results, as performance may vary significantly in other environments.
- Solution: Future research should incorporate a more diverse array of hardware and software environments to broaden the applicability of the findings.

Programming Language and JDK Version Specificity:

- Description: The algorithms were implemented and tested solely in Java, using a specific JDK version.
- Impact: This limitation confines the results to Java-based environments, potentially not representing performance in other programming contexts.
- Solution: Expanding the study to include different programming languages and multiple JDK versions can offer a more comprehensive understanding of the algorithms' efficiency.

Data Type and Size Limitation:

- Description: The range of dataset types and sizes used in the experiments might not cover all possible application scenarios.
- Impact: The algorithms' performance on other data types or extremely large datasets remains untested, which might be critical for certain applications.
- Solution: Including a wider variety of data types and larger datasets in future experiments will provide insights into the algorithms' scalability and efficiency under different conditions.

Absence of Real-World Application Testing:

- Description: The study was conducted in a controlled, isolated environment, possibly not reflecting real-world application performance.
- Impact: This may not reveal practical challenges or performance issues in actual software systems.
- Solution: Implementing and testing the algorithms within real-world applications or more complex systems can yield insights into their practicality and real-world performance.

Threats to Validity:

These limitations pose threats to the external validity of the study, potentially impacting the extent to which the results can be generalised to other environments and scenarios. Addressing these limitations in future research is crucial to enhance the robustness and applicability of the findings in diverse real-world settings. This format adheres to the conventional structure of scientific papers, clearly outlining each limitation, its impact on the study, and suggested remedies. This approach ensures clarity and aids in guiding future research directions.

4.3 Conclusions

The key conclusions drawn from this study are:

Superior Performance of BubbleSortWhileNeeded: The WN algorithm emerged as the most efficient across the majority of test scenarios. Its ability to adapt to the dataset's state and cease operations once the array is sorted allows it to outperform the other two algorithms consistently. This characteristic makes it particularly suitable for larger datasets and a variety of data types, affirming its robustness and versatility.

Moderate and Consistent Performance of BubbleSortUntilNoChange: Although the UNC algorithm did not distinctly excel in partially sorted data as hypothesised, it demonstrated moderate and consistent performance. Its approach of continuing passes until no changes are made offers a balance between efficiency and simplicity, making it a reliable choice for applications where data is not excessively large or complex.

Limited Scalability of BubbleSortPassPerItem: The PPI algorithm, while showing reasonable efficiency in smaller datasets, struggled with scalability as the dataset size increased. This limitation is particularly evident in its handling of larger and more complex data types, suggesting that it may be better suited for smaller or simpler sorting tasks.

Importance of Algorithm Selection Based on Data Characteristics: The study highlights the importance of choosing the right sorting algorithm based on the characteristics of the data and the requirements of the application. While WN stands out in terms of overall efficiency, the specific context and constraints of a use case could make UNC or even PPI a more appropriate choice.

Potential for Further Optimization and Research: The results also open avenues for further research, particularly in exploring the conditions under which UNC might offer the best performance and investigating potential optimizations for PPI in handling larger datasets.

In summary, this study provides valuable insights into the performance dynamics of these bubble sort variants, underscoring the necessity of considering both the algorithmic efficiency and the specific use case requirements in algorithm selection. The findings serve as a guide for software developers and researchers in making informed decisions about sorting algorithm implementation in various contexts.

Appendix

A. Materials

Any documents you used for your informed consent (information sheets, consent) or as part of your apparatus (e.g., manual, hand-out), please include them here.

```
SortingAlgo,SortingType,DataType,DataSize,AvgTime,MedianTime,25th,75th

PassPerItem, Ascending, Double,100,134031.27,38295.5,35023.25,143754.0

UntilNoChange, Ascending, Double,100,9801.25,9680.0,9244.5,10109.25

WhileNeeded, Ascending, Double,100,8769.1,8755.5,8239.75,9227.5

PassPerItem, Ascending, Double,1000,2874693.41,2436406.0,2273714.75,2934533.25

UntilNoChange, Ascending, Double,1000,47159.46,55372.5,17391.25,63821.0

WhileNeeded, Ascending, Double,1000,42249.06,49427.0,17136.75,55703.5

PassPerItem, Ascending, Double,10000,276930069.15,264615007.0,248006153.75,305246250.75

UntilNoChange, Ascending, Double,10000,66173.09,27739.0,24154.75,32906.5

WhileNeeded, Ascending, Double,10000,66336.97,28070.0,25017.75,31543.5

PassPerItem, Ascending, Float,100,177026.71,86218.5,24043.25,222173.25

UntilNoChange, Ascending, Float,100,9746.68,8920.5,7148.75,11729.0

WhileNeeded, Ascending, Float,100,8701.72,7676.5,6715.25,10762.5

PassPerItem, Ascending, Float,1000,2094649.59,1774684.5,1725830.75,1890127.25

UntilNoChange, Ascending, Float,1000,40982.54,47866.0,13473.25,52469.0

WhileNeeded, Ascending, Float,1000,36386.44,44857.0,11144.25,48609.25

PassPerItem, Ascending, Float,10000,219142845.08,214927718.5,196858856.25,238302566.5

UntilNoChange, Ascending, Float,10000,63916.41,24943.0,22457.5,27902.25

WhileNeeded, Ascending, Float,10000,60814.92,23827.5,21433.5,26196.75

PassPerItem, Ascending, Integer,100,163160.81,135255.5,38225.0,191866.5
```

```
UntilNoChange, Ascending, Integer,100,8789.5,7565.0,7201.25,10202.0

WhileNeeded, Ascending, Integer,100,7708.37,7004.5,6837.75,8857.25

PassPerItem, Ascending, Integer,1000,4811651.19,4823096.5,4566910.25,5106661.75

UntilNoChange, Ascending, Integer,1000,48205.55,58001.5,17559.25,65513.75

WhileNeeded, Ascending, Integer,1000,42409.88,52697.5,15701.75,55737.5

PassPerItem, Ascending, Integer,10000,316634296.27,304273091.5,293665702.5,331877777.75

UntilNoChange, Ascending, Integer,10000,79234.89,33677.5,32560.5,38251.25

WhileNeeded, Ascending, Integer,10000,71280.24,32781.0,31832.25,37134.5

PassPerItem, Descending, Double,100,271112.99,182430.0,50791.75,253548.25

UntilNoChange, Descending, Double,100,265168.8,188436.5,50994.5,210834.75

WhileNeeded, Descending, Double,100,224654.01,106433.0,91535.0,409576.5

PassPerItem, Descending, Double,1000,6261036.78,5225998.0,4902484.75,5921858.0

UntilNoChange, Descending, Double,1000,6095436.68,5219011.5,4893456.75,5992756.0

WhileNeeded, Descending, Double,1000,4399849.85,3596293.0,3453462.5,4280346.0

PassPerItem, Descending, Double,10000,471474142.91,454141101.0,443673602.75,480638724.25

UntilNoChange, Descending, Double,10000,512906983.95,490981914.0,476758968.5,534181535.75

WhileNeeded, Descending, Double,10000,334501580.5,316353538.0,306619344.25,345640093.5

PassPerItem, Descending, Float,100,275707.59,181362.0,49941.25,206081.0

UntilNoChange, Descending, Float,100,310454.63,189346.5,50622.75,643480.5

WhileNeeded, Descending, Float,100,203240.68,106329.5,35774.75,478877.0
```

```
PassPerItem, Descending, Float,1000,5598888.46,4729720.5,4629436.5,5001988.75

UntilNoChange, Descending, Float,1000,6051540.84,5103313.5,5010470.75,5434265.75

WhileNeeded, Descending, Float,1000,4299333.29,3572766.5,3494200.75,3919738.25

PassPerItem, Descending, Float,10000,480513418.82,458131103.5,435376136.0,516750742.75

UntilNoChange, Descending, Float,10000,490026579.35,468066197.5,451431976.0,515433147.5

WhileNeeded, Descending, Float,10000,339415533.13,316873918.5,295714865.75,374025464.5

PassPerItem, Descending, Integer,100,435391.44,210377.5,74157.75,713543.5

UntilNoChange, Descending, Integer,100,357686.76,217876.0,90813.5,288612.75

WhileNeeded, Descending, Integer,100,267056.39,123750.0,53577.25,388283.25

PassPerItem, Descending, Integer,1000,6654942.38,6020677.0,5844737.75,6301141.75

UntilNoChange, Descending, Integer,1000,6561974.33,6039673.0,5817488.75,6347579.75

WhileNeeded, Descending, Integer,1000,4023066.78,3564673.0,3466524.75,3708288.5

PassPerItem, Descending, Integer,10000,457069665.36,438992653.0,425030282.75,468229990.0

UntilNoChange, Descending, Integer,10000,537464900.55,530287957.5,508170897.5,557521183.25

WhileNeeded, Descending, Integer,10000,316146944.9,304289132.0,293967380.75,335887176.75

PassPerItem, Random, Double,100,309485.55,166783.5,104220.75,474640.0

UntilNoChange, Random, Double,100,273393.34,156962.0,113719.25,244408.0

WhileNeeded, Random, Double,100,196907.01,97822.5,75826.25,351895.5

PassPerItem, Random, Double,1000,5219240.11,4324345.0,4210189.0,4617171.0

UntilNoChange, Random, Double,1000,4932689.22,4113027.0,4040777.75,4379354.0
```

```
WhileNeeded, Random, Double,1000,3511942.5,2904998.0,2851333.75,3220969.25

PassPerItem, Random, Double,10000,535784648.82,515202656.0,497225691.75,558894340.25

UntilNoChange, Random, Double,10000,528007862.74,510704008.5,498657906.0,551872176.0

WhileNeeded, Random, Double,10000,389420312.44,373422571.0,363978781.25,413024865.0

PassPerItem, Random, Float,100,286154.26,143138.0,56090.75,279878.25

UntilNoChange, Random, Float,100,237951.21,142607.5,62422.5,258990.75

WhileNeeded, Random, Float,100,195826.39,129097.0,56252.75,348310.25

PassPerItem, Random, Float,1000,5152139.2,4363755.0,4147627.25,4641265.75

UntilNoChange, Random, Float,1000,5018341.94,4576795.5,4352857.75,4831145.75

WhileNeeded, Random, Float,1000,3698380.51,2916291.0,2801608.25,3203694.75

PassPerItem, Random, Float,10000,531016703.83,515711726.0,477665517.5,550762800.75

UntilNoChange, Random, Float,10000,551657765.37,538990662.0,492795056.5,582258356.75

WhileNeeded, Random, Float,10000,352447772.13,351741806.0,310166362.0,377510614.0

PassPerItem, Random, Integer,100,308079.63,161025.0,68716.5,607835.25

UntilNoChange, Random, Integer,100,301295.62,161733.0,80996.75,561935.5

WhileNeeded, Random, Integer,100,147298.42,88577.5,84715.5,113085.75

PassPerItem, Random, Integer,1000,7452461.16,6853509.0,6755296.5,7216147.25

UntilNoChange, Random, Integer,1000,8589068.21,7948312.5,7758825.75,8488477.75

WhileNeeded, Random, Integer,1000,5785281.86,5370706.0,5256828.0,5647504.75

PassPerItem, Random, Integer,10000,657263524.75,648932512.0,636718980.75,669010272.5
```

```
UntilNoChange, Random, Integer,10000,742559033.76,729448098.5,714732704.75,763703436.5


WhileNeeded, Random, Integer,10000,475156396.66,461104057.5,453292279.75,485790426.0
```

B. Reproduction Package (or: Raw Data)

Before, during, and after the experiment you collected all kinds of data. Don't ever throw such data away! Any plots, tables, summaries, and statistics provided in this report should be recreatable from the raw data you have.

If you only collected a small amount of data, put it in this Appendix right here.

If you collected data in forms that are better kept in separate files, then zip up those files, and submit them as a "reproduction package" supporting this report.

C. Compact Data Table (Google sheet)

Compact data generated at the end can be viewed here. The file contains a CSV file converted to the google sheet file. The table shows all relevant independent variables and its result from minimum, maximum, mean, 25th percentile and 75th percentile execution time.