Word Count: 1039

CS2001 W10-Complexity-Report

1. Introduction

This report investigates the practical performance of three sorting algorithms: selection sort, merge sort, and quicksort (using the last element as a fixed pivot). Although their theoretical complexities are well understood, this assignment focuses on how they behave *in practice* when sorting arrays of different sizes and structures. I designed a series of experiments in Java to measure their running time across four input types: random, sorted, reverse sorted, and nearly sorted. The results were analysed and compared with theory, and the report concludes with practical recommendations.

2. Design and Implementation of Algorithms and Experiments

2.1 Algorithm Implementations

All algorithms were implemented in straightforward, textbook-style Java without optimisations. Selection sort performs repeated minimum selection, merge sort uses recursive splitting and merging, and quicksort partitions around a fixed last-element pivot. Merge sort uses auxiliary arrays while the other two sort in place.

2.2 Input Types

Each algorithm was tested on four types of input:

1. **Random:** average-case behaviour for all algorithms.
2. **Sorted:** best-case for some, but worst-case for quicksort with a poor pivot strategy.
3. **Reverse Sorted:** another structured ordering that tends to trigger worst-case behaviour.
4. **Nearly Sorted:** a realistic case where only a few elements are out of order.

2.3 Timing Setup

Timing and Repeated Trials
I measured runtime using System.nanoTime(), which provides nanosecond-resolution wall-clock timing. For each combination of algorithm, input type, and input size, I ran the algorithm **five times** on identical copies of the same base array. Rather than averaging in Java, I stored **all five raw timings** in the CSV.

Word Count: 1039

The analysis script (written in Python) then grouped these measurements to compute both the **mean** and **standard deviation**, which were plotted as **error bars**. These error bars exposed significant noise at small inputs, mainly from JVM warm-up or occasional garbage collection pauses.

The array sizes ranged from 1,000 up to **15,000**. I initially attempted larger inputs, but quicksort with a fixed pivot consistently caused a stack overflow around 16,500 elements on sorted inputs due to its worst-case recursion depth. Selection sort also became extremely slow beyond that point, so I capped the largest size at 15,000 for practical and reliability reasons.

2.4 Data Analysis Workflow

I initially used Excel but switched to Python because it offered a repeatable and more reliable analysis pipeline (discussed in reflection).

3. Presentation and Analysis of Results

3.1 Selection Sort (Appendix A)

Selection sort showed the expected **quadratic growth** across all input types. The running time increased steeply as *n* grew, and doubling the input size produced roughly four times the runtime.

Selection sort's error bars were relatively small, particularly for larger inputs. This is expected because the algorithm runs for a long time and its runtime is dominated by deterministic comparisons; short-lived system noise contributes proportionally less to the overall timing.

Because selection sort always compares every element with every other remaining element, its performance was **almost identical** on random, sorted, reverse, and nearly sorted inputs.

This confirmed the theoretical prediction that selection sort is $O(n^2)$ in all cases. It performed reasonably for very small inputs but scaled poorly.

3.2 Merge Sort (Appendix B)

Merge sort produced a smooth and predictable $O(nlog\ n)$ curve. All four input types resulted in very similar times, because merge sort always divides the array and merges in a fixed pattern, regardless of ordering.

In earlier results (especially at n=1000), I observed a noticeable spike for merge sort on random input. With error bars added, it is clear that this point has a large variance and is caused by

measurement noise rather than algorithmic behaviour. The mean is inflated because one or two runs were slowed by JVM warm-up effects.

By *n* = 15,000, merge sort consistently completed in around 1–2 ms, making it one of the fastest and most reliable methods. This matched the theoretical expectation that merge sort is efficient and stable regardless of input order.

3.3 Quicksort (Appendix C)

Quicksort demonstrated the most interesting behaviour. For **random input**, it was extremely fast, often the fastest of all three. Its in-place nature and good cache locality gave it a constant-factor advantage over merge sort.

However, on **sorted and reverse-sorted inputs**, quicksort performed very poorly. At *n* = 15,000, sorted input took around 50–60 ms, and reversed input even longer, matching the theoretical $O(n^2)$ behaviour.

The **nearly sorted** input produced intermediate results: slower than random but far better than the true worst case. Even a small amount of disorder in the array was enough to avoid consistently terrible partitions.

The error bars for sorted and reverse-sorted inputs were small, indicating that the poor performance is highly consistent across runs. This reinforces the fact that these inputs reliably trigger worst-case $O(n^2)$ behaviour due to the fixed last-element pivot.

This strongly illustrated quicksort's sensitivity to pivot choice. With a better pivot strategy (like median-of-three), its performance would be much more stable.

4. Recommendations

For large datasets or unknown input structure, merge sort is the safest and most consistent option. Quicksort can outperform it on random data, but only when using a better pivot strategy; with a fixed pivot, its worst-case behaviour makes it unsuitable for structured inputs.

For very small inputs, all three algorithms perform similarly. Selection sort may be acceptable here due to its simplicity, though merge sort and quicksort still perform well.

Word Count: 1039

5. Reflection

During the project, I learned that designing experiments is as important as implementing algorithms. My first attempt at analysing results in Excel quickly became unmanageable. Small changes in data broke formulas, and recreating graphs manually was inefficient. Switching to Python made the workflow repeatable, reliable, and far easier to maintain.

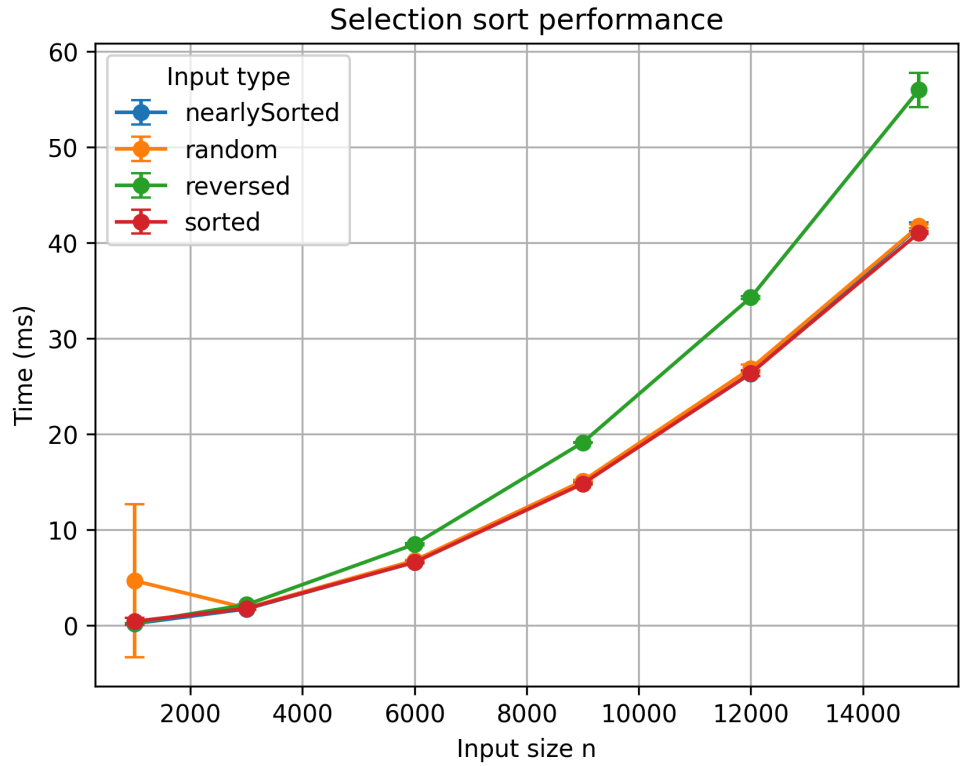
I also saw first-hand that timing experiments are noisy. Using nanoTime() and averaging five runs helped smooth out fluctuations, but occasional anomalies still appeared, reminding me that performance measurement must be handled carefully.

6. Conclusion

The experiments confirmed the textbook complexities of the three algorithms and demonstrated how dramatically input order can affect performance - especially for quicksort. Merge sort proved to be the most reliable choice overall, while quicksort excelled on average-case data but suffered badly on structured inputs. Selection sort behaved exactly as expected: simple, predictable, but not scalable.

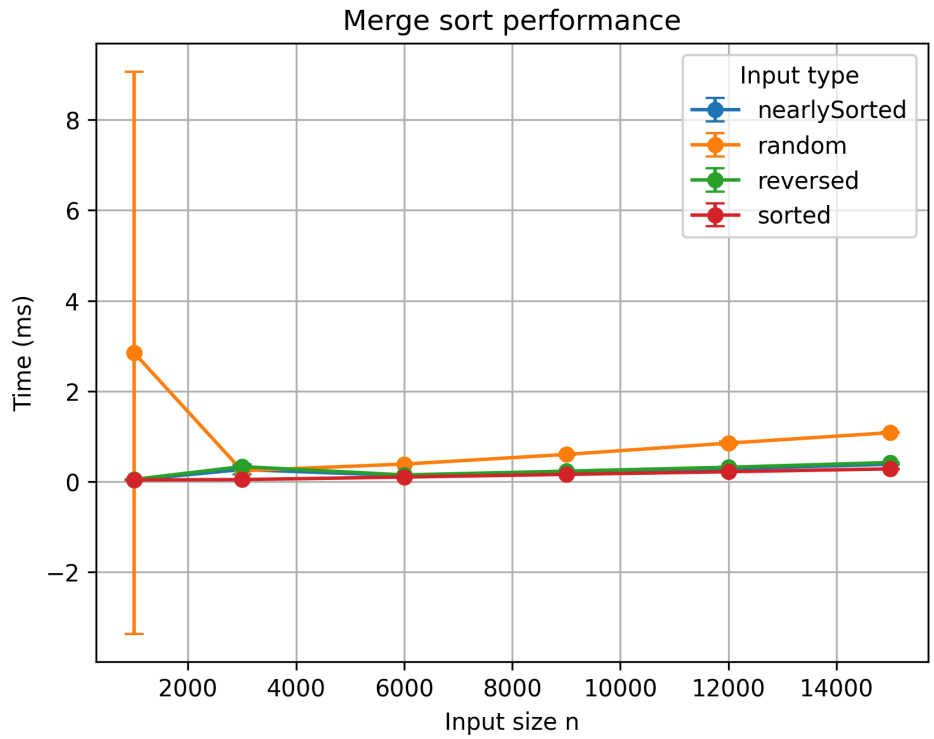
7. Appendix

Appendix A:

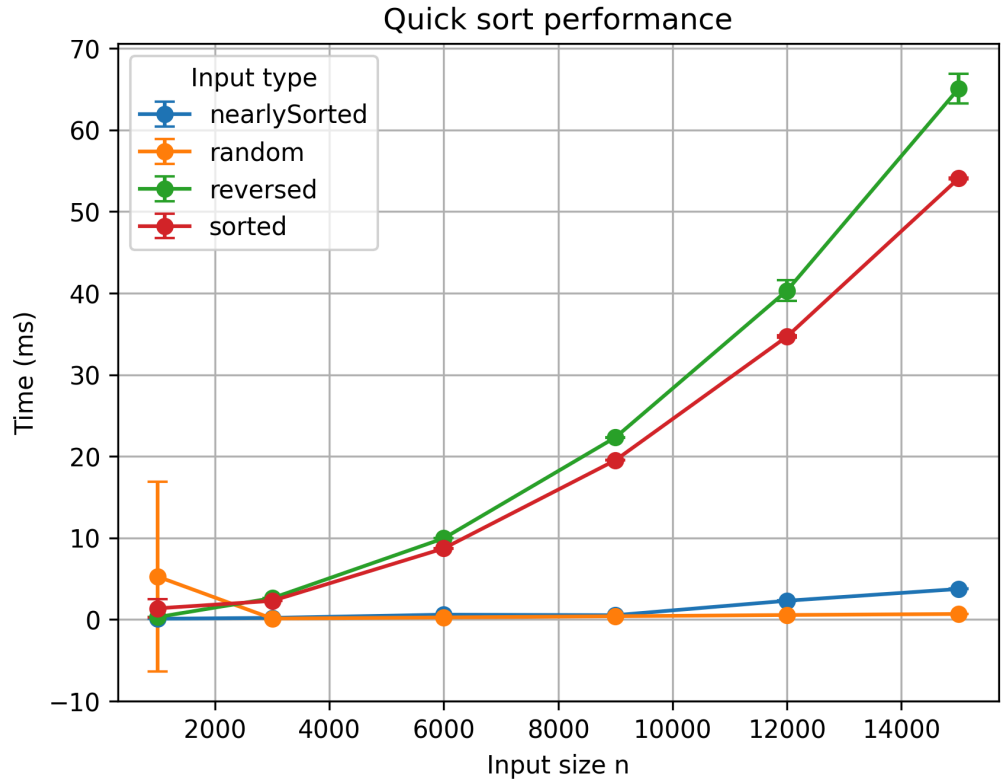


Word Count: 1039

Appendix B:

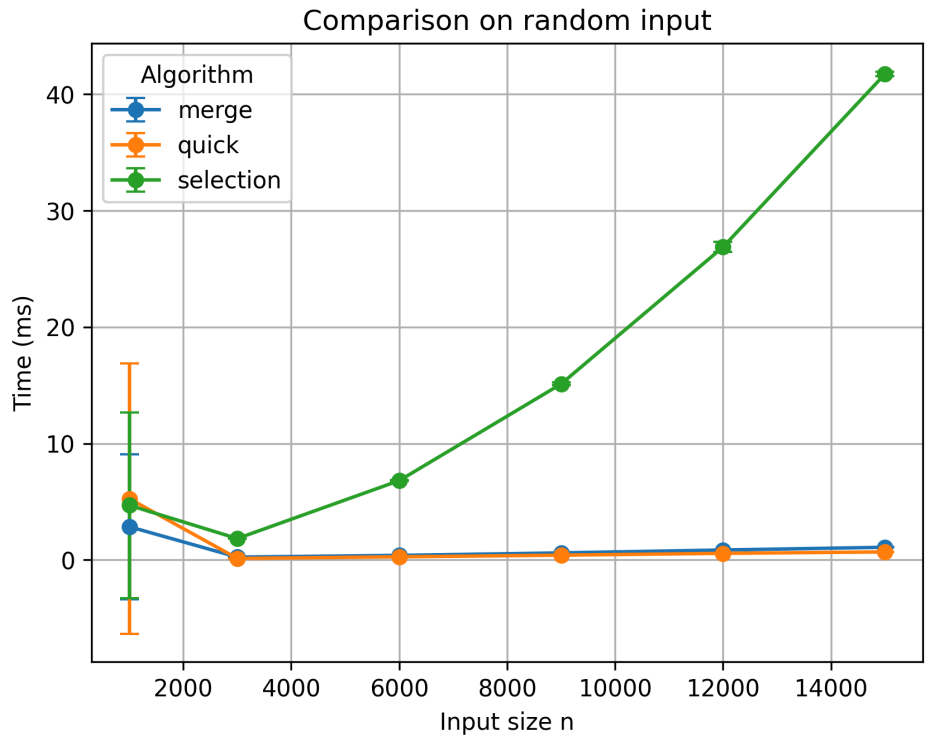


Appendix C:



Word Count: 1039

Appendix D:



Bibliography

[1] Oracle Java Documentation - *System.nanoTime()*.
https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--

[2] GeeksforGeeks - *Quicksort Algorithm*.
https://www.geeksforgeeks.org/quick-sort/

[3] GeeksforGeeks - *Merge Sort Algorithm*.
https://www.geeksforgeeks.org/merge-sort/

[4] GeeksforGeeks - *Selection Sort Algorithm*.
https://www.geeksforgeeks.org/selection-sort/

Word Count: 1039

[5] Baeldung - *Java and CPU Time: How System.nanoTime() Works.*
https://www.baeldung.com/java-system-nanotime

[6] StackOverflow - *Why Does JVM Warm-Up Affect Performance Measurements?*
https://stackoverflow.com/questions/19005712/why-does-jvm-warm-up-affect-benchmarks

[7] StackOverflow - *Why is Quicksort Slow on an Already Sorted Array?*
https://stackoverflow.com/questions/16201656/quicksort-sorted-array-slow

[8] Python pandas Documentation - *GroupBy and Aggregation*.
https://pandas.pydata.org/pandas-docs/stable/user_guide/groupby.html

[9] Matplotlib Documentation - *Errorbar Plots*.
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.errorbar.html

[10] Visualgo - *Interactive Visualisations of Sorting Algorithms*.
https://visualgo.net/en/sorting