CS-253 Design and Analysis of Algorithms Practical Assignment: Comparison of Sorting Algorithms

Appaji Nagaraja Dheeraj (241CS110)
Adarsh Malipatil (241CS102)

February 14, 2026

1 Introduction

The objective of this assignment is to conduct a comprehensive study of various sorting algorithms and compare their theoretical time complexities with their actual empirical performance. The algorithms under study include Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort (with multiple pivot strategies), Heap Sort, and Radix Sort.

The primary focus is on performance analysis rather than implementation. By designing careful test cases and measuring execution times across different input sizes and initial orderings (random, sorted, and reverse-sorted), we aim to draw meaningful conclusions about the efficiency and stability of these algorithms in real-world scenarios.

2 Data Generation and Experimental Setup

To ensure the reliability and reproducibility of the results, the following experimental environment and methodology were established:

- **Machine Specifications:** The experiments were conducted on an ASUS Vivobook 16X equipped with an Intel i7 processor and an NVIDIA RTX 4050 Studio GPU, running on Windows.
- **Timing Mechanism:** Execution times were measured using Pythonś `time.perf_counter()` to manage the benchmarking process. The core timing for the C implementations was performed inside the C code using `clock_gettime(CLOCK_MONOTONIC)` to ensure high precision and eliminate Pythonś subprocess overhead.
- **Number of Experiment Repetitions:** Each sorting experiment (a specific algorithm run on a particular input size and type) was repeated **7 times**, as configured in the `benchmark.py` script.
- **Reported Times:** For each experiment, the **average execution time** across the 7 repetitions is reported. Times are presented in **seconds (s)**.

- **Input Selection:** The test data used for benchmarking was pre-generated. The inputs were selected to cover a range of sizes and initial orderings to evaluate best-case, worst-case, and average-case performance:
  - **Input Sizes (N):** The following input sizes were tested: 100, 500, 1000, 5000, 10000, 25000, 50000, 75000, and 100000 elements.
  - **Input Types:** For each input size, three distinct types of data were generated: Random (average-case), Sorted (best-case for some, worst-case for others), and Reverse Sorted (worst-case for many).
- **Consistency of Inputs:** The same set of input data files was used for all sorting algorithms to ensure a fair comparison.

3 Which of the three versions of Quick sort seems to perform the best?

We compared three versions of Quick Sort based on their pivot selection strategies:

1. **Pivot Choice 1 (First Element):** Uses the first element of the array as the pivot.
2. **Pivot Choice 2 (Random Element):** Uses a randomly selected element as the pivot.
3. **Pivot Choice 3 (Median of Three):** Uses the median of the first, middle, and last elements as the pivot.

3.1 Theoretical Analysis

The performance of Quick Sort is highly dependent on the pivot selection. The recurrence relations for the different cases are:

- **Best Case:** $T(n) = 2T(n/2) + O(n) \implies O(n \log n)$
- **Worst Case:** $T(n) = T(n-1) + O(n) \implies O(n^2)$
- **Average Case:** $T(n) = 2T(n/2) + O(n) \implies O(n \log n)$

3.2 Empirical Results

The First Pivot strategy degrades to $O(n^2)$ for sorted and reverse-sorted inputs. Our terminal output shows that for $N = 100,000$ on reverse-sorted data, the First Pivot variant took **9.993 seconds**, while the Median-of-Three and Random Pivot strategies finished in **0.0069 seconds** and **0.0064 seconds** respectively.

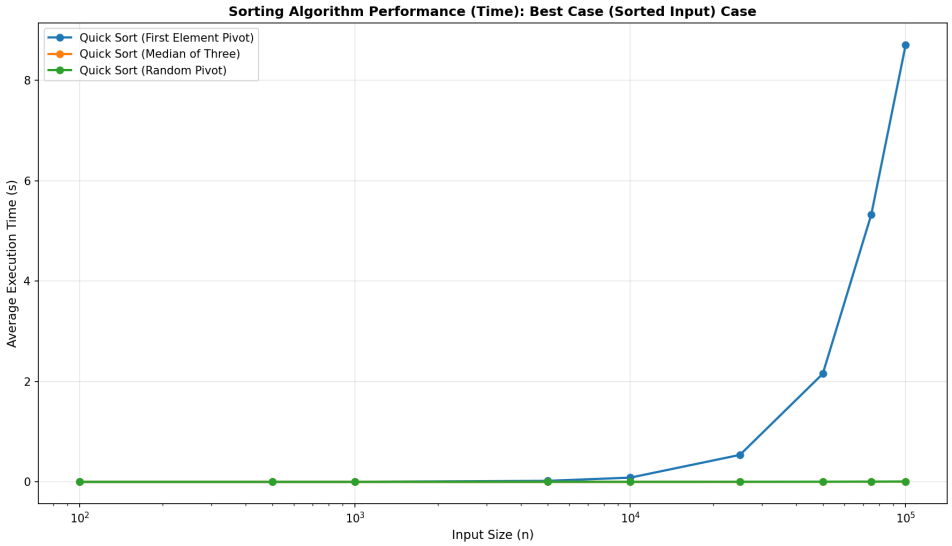

Figure 1: Quick Sort Variants: Best Case (Sorted Input) - Linear Scale

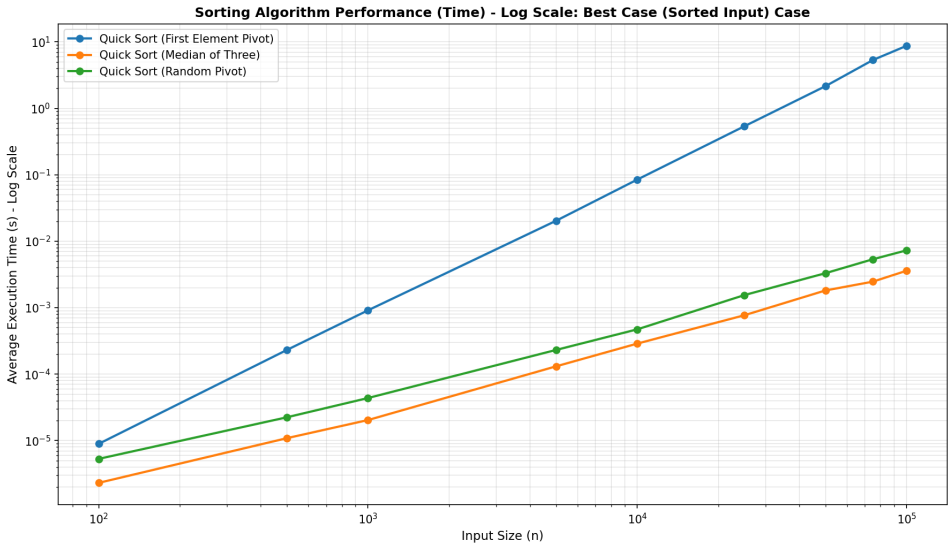

Figure 2: Quick Sort Variants: Best Case (Sorted Input) - Logarithmic Scale

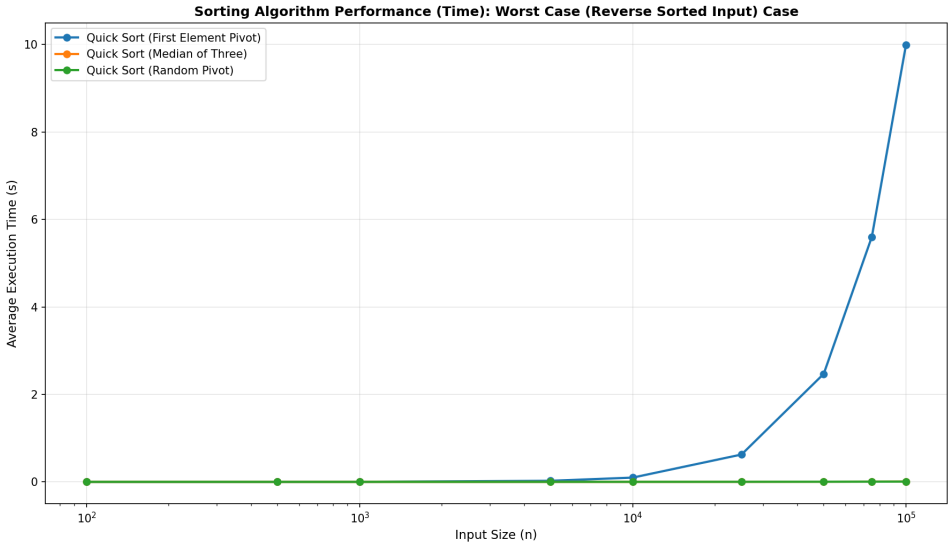

Figure 3: Quick Sort Variants: Worst Case (Reverse Sorted Input) - Linear Scale

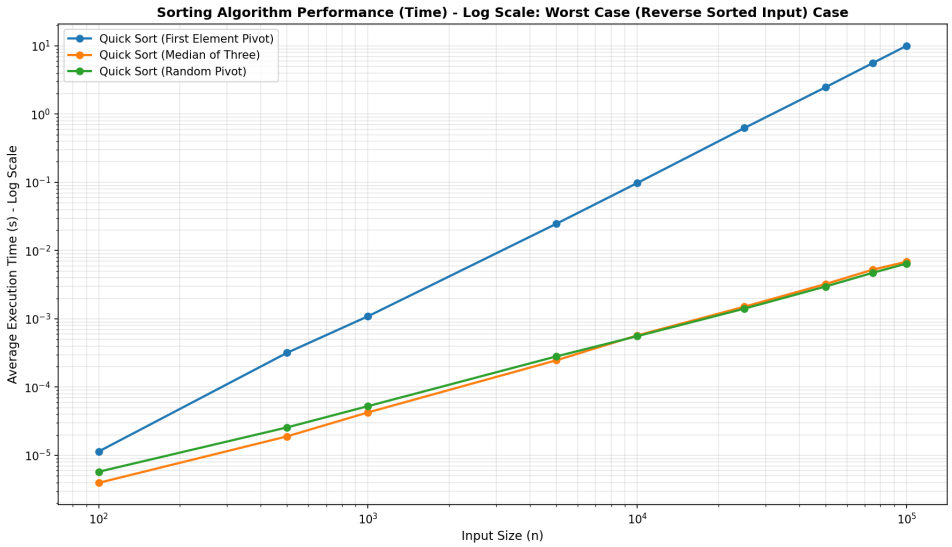

Figure 4: Quick Sort Variants: Worst Case (Reverse Sorted Input) - Logarithmic Scale

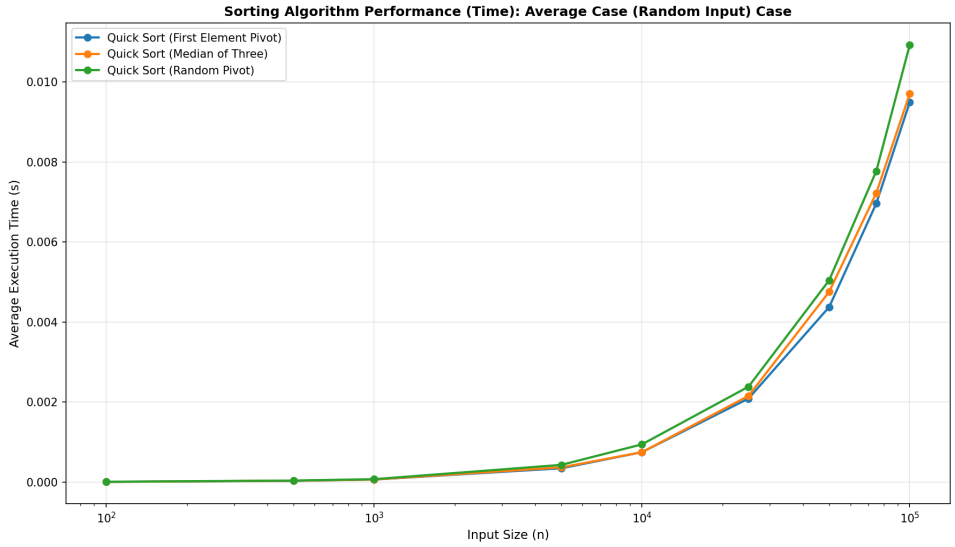

Figure 5: Quick Sort Variants: Average Case (Random Input) - Linear Scale

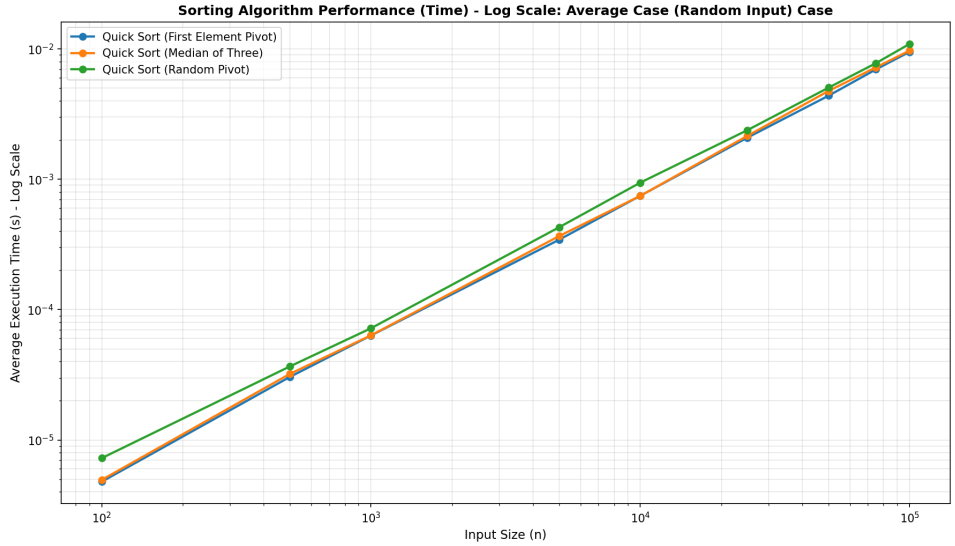

Figure 6: Quick Sort Variants: Average Case (Random Input) - Logarithmic Scale

3.3 Conclusion

The **Median-of-Three Quick Sort performs the best overall** due to its superior pivot selection strategy. In a naive Quick Sort (First Pivot), selecting the first element as the pivot on already sorted or reverse-sorted data results in highly unbalanced partitions (one sub-array of size 0 and another of size $n-1$). This leads to the $O(n^2)$ worst-case complexity and risks a stack overflow due to deep recursion, as observed in our benchmarks where the First Pivot variant took nearly 10 seconds for $N = 100,000$.

The Median-of-Three strategy mitigates this by selecting the median of the first, middle, and last elements. This heuristic significantly increases the probability of picking a pivot that is closer to the actual median of the dataset, ensuring more balanced partitions even on structured data (sorted or nearly sorted). By maintaining a partitioning ratio closer to $1:1$, the recurrence remains $T(n) = 2T(n/2) + O(n)$, keeping the complexity

at $O(n \log n)$. While the Random Pivot strategy also avoids the $O(n^2)$ trap, Median-of-Three is often slightly faster in practice as it avoids the overhead of generating random numbers while providing similar protection against pathological cases.

4 Which of the seven sorts seems to perform the best?

We compared Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort (Median of Three), Heap Sort, and Radix Sort.

4.1 Theoretical Complexities

The following table summarizes the theoretical time and space complexities of the seven algorithms:

Table 1: Theoretical Complexities of Sorting Algorithms

Algorithm	**Best Case**	**Average Case**	**Worst Case**	**Space**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort (Med3)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Radix Sort	$O(d(n+k))$	$O(d(n+k))$	$O(d(n+k))$	$O(n+k)$

4.2 Empirical Analysis

On random data ($N = 100{,}000$), **Radix Sort** was the fastest at **0.0047 seconds**, followed by **Quick Sort (Med3)** at **0.0097 seconds**. For sorted data, **Bubble Sort** and **Insertion Sort** were exceptionally fast (**0.0002 seconds**) due to their $O(n)$ best-case optimization.

In contrast, **Selection Sort** consistently showed poor performance across all input types due to its fixed $O(n^2)$ complexity, making it unsuitable for large datasets. The results also show that $O(n \log n)$ **algorithms** such as Merge Sort and Heap Sort maintained stable and predictable performance regardless of input distribution. Furthermore, the experiments highlight the significant performance gap between quadratic and logarithmic growth rates as input size increases. Both **linear-scale and logarithmic-scale plots** were included to clearly visualize performance trends and growth rates, where the logarithmic graphs particularly emphasize the differences in algorithmic complexity. Overall, the empirical results clearly demonstrate the importance of algorithm efficiency and input characteristics in determining practical runtime performance.

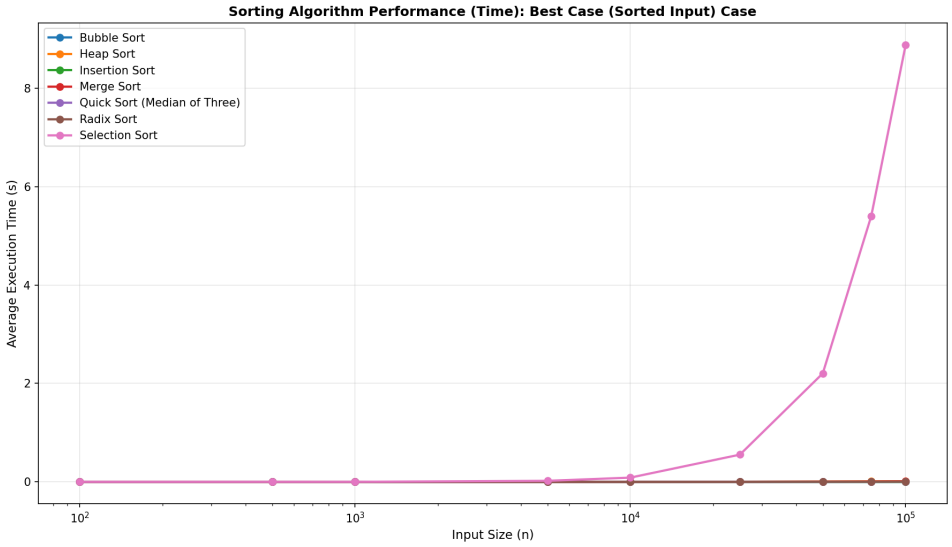

Figure 7: All Sorts: Best Case (Sorted Input) - Linear Scale

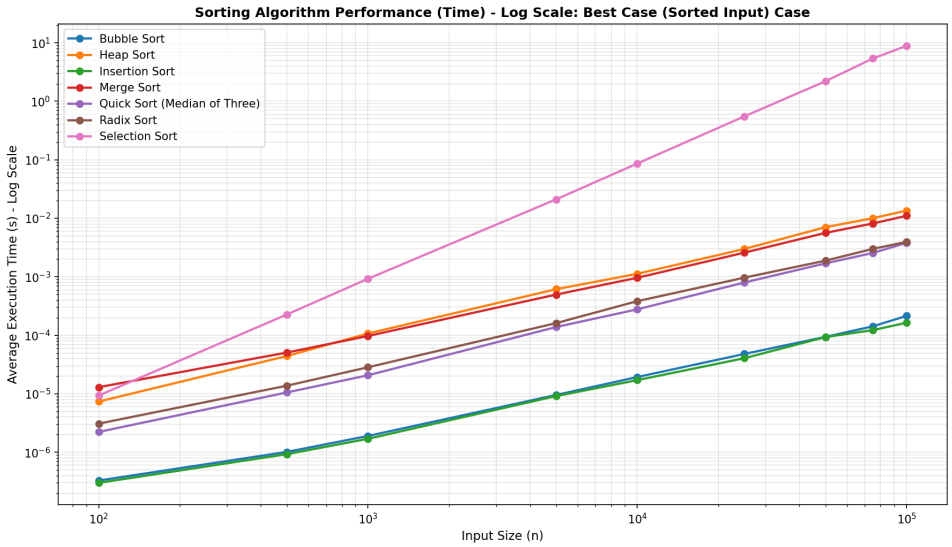

Figure 8: All Sorts: Best Case (Sorted Input) - Logarithmic Scale

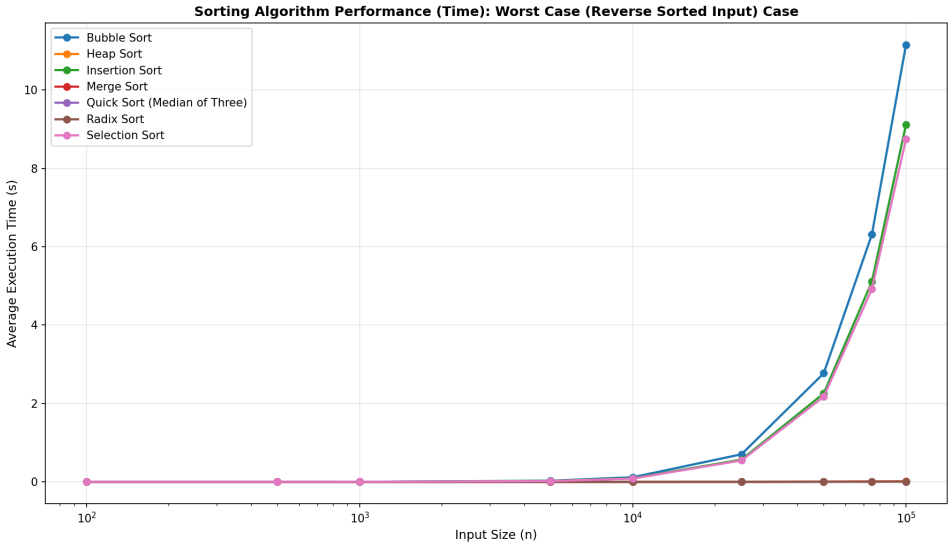

Figure 9: All Sorts: Worst Case (Reverse Sorted Input) - Linear Scale

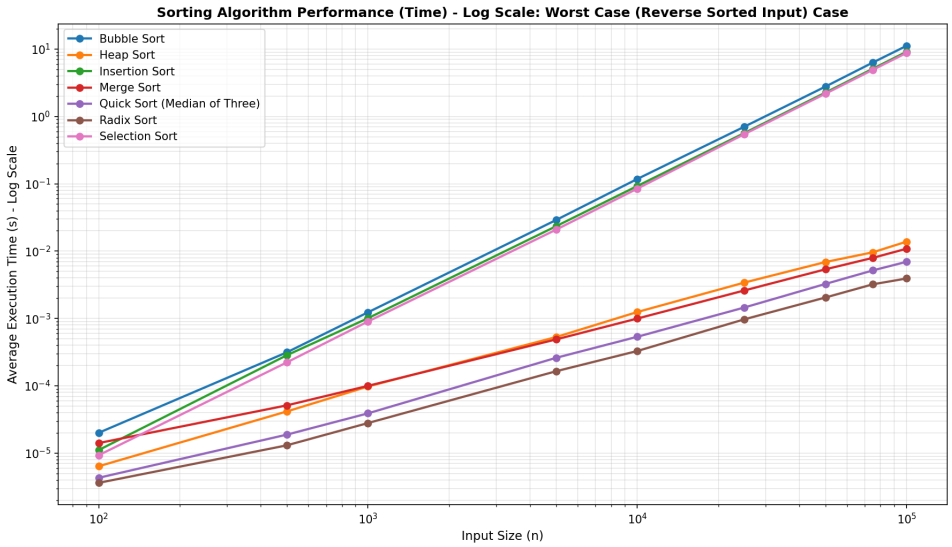

Figure 10: All Sorts: Worst Case (Reverse Sorted Input) - Logarithmic Scale

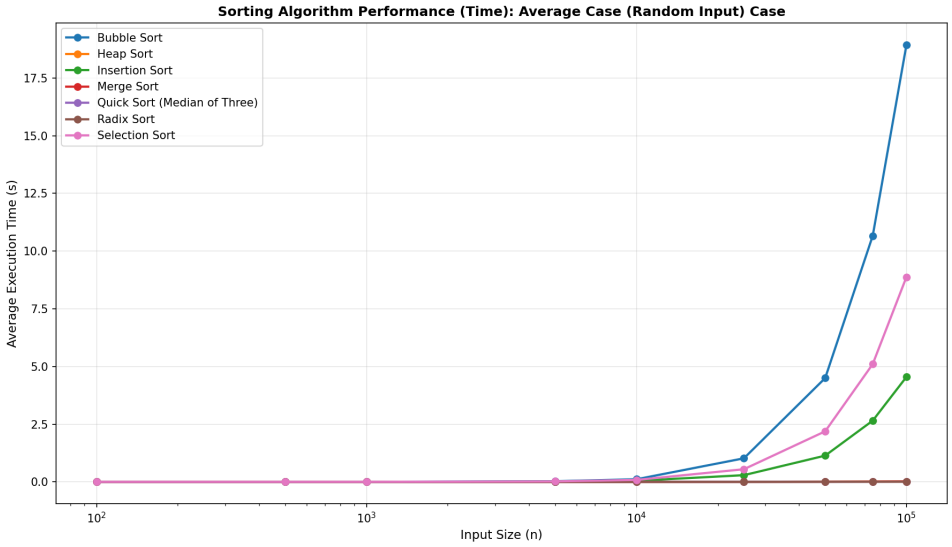

Figure 11: All Sorts: Average Case (Random Input) - Linear Scale

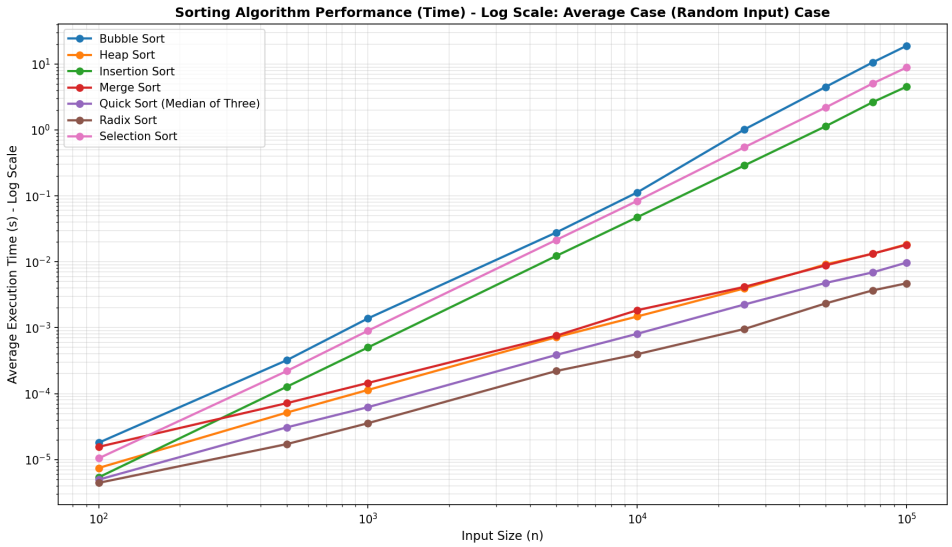

Figure 12: All Sorts: Average Case (Random Input) - Logarithmic Scale

4.3 Conclusion

For large datasets, **Radix Sort** is the overall winner due to its linear-like performance. Among comparison-based sorts, **Quick Sort (Median of Three)** is the most efficient on average, while **Merge Sort** and **Heap Sort** are more reliable in the worst case.

5 Correlation Between Comparisons and Execution Time

The number of comparisons is a key determinant of execution time for comparison-based sorting algorithms. Our analysis confirms a strong positive correlation between these two metrics.

Table 2: Correlation between Comparisons and Time

Algorithm	**Correlation (r)**	**P-value**
Bubble Sort	0.9615	1.53×10^{-15}
Heap Sort	0.9816	1.68×10^{-19}
Insertion Sort	1.0000	2.94×10^{-53}
Merge Sort	0.9976	1.42×10^{-30}
Quick Sort (Med3)	0.8789	1.63×10^{-09}
Selection Sort	0.9996	8.77×10^{-40}

The near-perfect correlation for Insertion Sort ($r = 1.0$) and Selection Sort ($r = 0.9996$) indicates that their execution time is almost entirely driven by the number of comparisons. Radix Sort, being a non-comparison sort, shows a correlation of 0.9933 with its internal operations, but these are not element comparisons in the traditional sense.

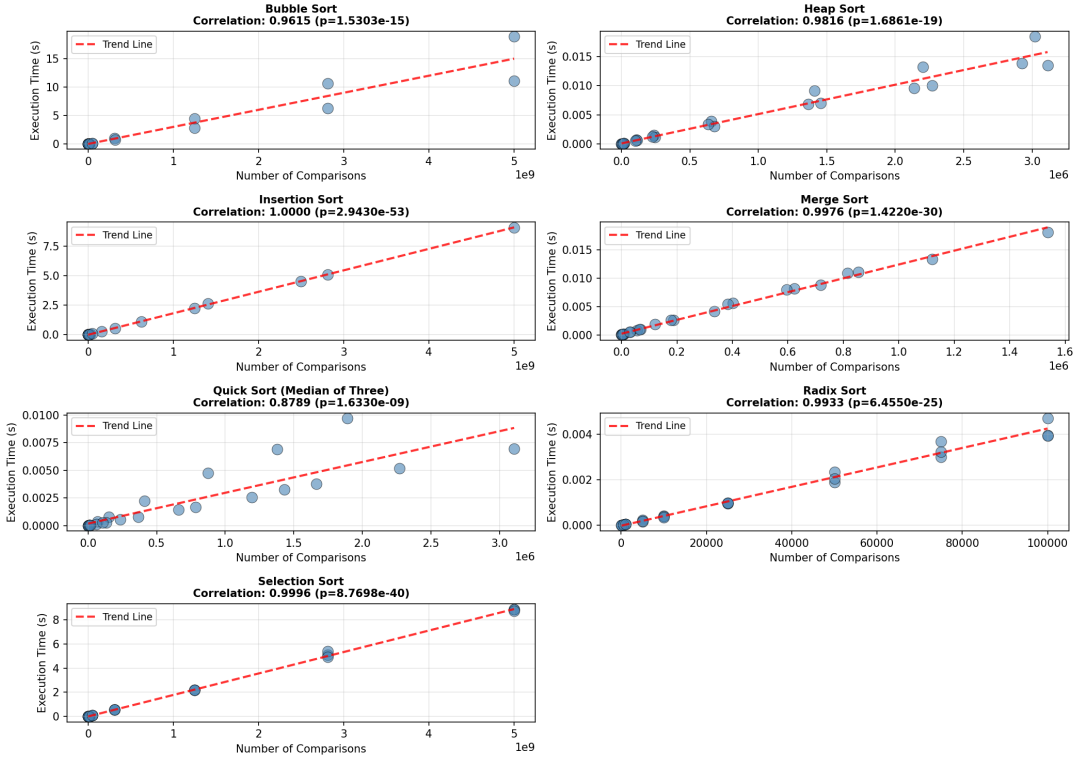

Figure 13: Correlation between Execution Time and Number of Comparisons - Linear Scale

6 Final Conclusion

This study presents an experimental evaluation of various sorting algorithms under different input conditions, including random, sorted (best case), and reverse sorted (worst case) data. The results show strong agreement between **empirical observations and theoretical time complexities**, validating the importance of asymptotic analysis in predicting algorithm performance.

Algorithms with $O(n^2)$ **complexity**, such as Bubble Sort, Selection Sort, and Insertion Sort (average and worst cases), show rapid growth in execution time and comparisons as input size increases. While simple and effective for small datasets, they become **impractical for large inputs due to poor scalability**.

In contrast, $O(n \log n)$ **algorithms** like Merge Sort, Heap Sort, and Quick Sort demonstrate significantly better scalability and performance for large datasets. The results also highlight that **pivot selection in Quick Sort is critical**, as naive strategies perform poorly on sorted inputs while median or random pivot strategies maintain efficient performance.

The study further shows that **input distribution significantly affects performance**. Adaptive algorithms such as Bubble Sort and Insertion Sort achieve linear-time behavior for sorted inputs, whereas non-adaptive algorithms like Selection Sort remain quadratic regardless of input order.

Additionally, **non-comparison-based methods** such as Radix Sort exhibit near-linear performance and outperform comparison-based algorithms under suitable conditions.

Overall, the findings confirm that **algorithm choice should depend on input size, data distribution, and application requirements**, and that asymptotic complexity remains a reliable indicator of performance despite practical factors such as implementation details and system overhead.