

# **Impact of Input Data Arrangement on the Practical Performance of Sorting Algorithms: A Comparative Study**

SHYAM SHIVAJI J<sup>1</sup>, ASWIN A<sup>2</sup>

1. *UG, B. E. Computer Science Engineering, I<sup>st</sup> year, SSN College of Engineering, Kalavakkam, Chennai – 603 110.*
2. *UG, B. Tech. Information Technology, I<sup>st</sup> year, SRM Institute of Science and Technology, Ramapuram, Chennai – 600 089.*

## **Abstract:**

Sorting algorithms are a fundamental component of computer science and are traditionally evaluated using asymptotic time complexity, which primarily considers input size while assuming generic input distributions. However, in practical scenarios, real-world data often exhibits specific structural characteristics such as partial ordering, duplication, or adversarial arrangements that are not adequately captured by theoretical analysis alone. This paper presents a comprehensive experimental study on the impact of input data arrangement on the practical performance of sorting algorithms, ranging from elementary algorithms such as Bubble Sort, Selection Sort, and Insertion Sort to more advanced algorithms including Merge Sort, Quick Sort, and Heap Sort.

All algorithms are implemented in C++ to ensure minimal runtime overhead and accurate performance measurement. Multiple input datasets are systematically generated to represent diverse data arrangements, including random, sorted, reverse-sorted, nearly sorted, duplicate-heavy, and patterned inputs. Performance is evaluated using metrics such as execution time, number of comparisons, and number of element movements.

The results demonstrate that algorithms with identical asymptotic time complexity can exhibit significantly different performance under varying input arrangements. In particular, simple algorithms like Insertion Sort outperform asymptotically faster algorithms on nearly sorted datasets, while divide-and-conquer algorithms show sensitivity to specific input patterns. This study highlights the limitations of relying solely on theoretical time complexity and emphasizes the importance of input data characteristics in algorithm selection. The findings provide practical guidelines for choosing appropriate sorting algorithms based on real-world data properties rather than theoretical bounds alone.

**Keywords:** Sorting Algorithms, Insertion Sort, Bubble Sort, Selection Sort, Merge Sort, Quick Sort, Heap Sort, Time Complexity, Asymptotic Notations, Input Data Characteristics, Execution Speed.

## **1. Introduction:**

Sorting is one of the most fundamental problems in computer science[1], [2] and serves as a building block for numerous applications, including database systems, search engines, data analytics, and operating systems. Over the decades, a wide variety of sorting algorithms have been developed, each with well-established theoretical time and space complexity bounds. Classical algorithm analysis primarily relies on asymptotic notations such as Big-O, Big- $\Omega$ ,

and Big- $\Theta$ [1] to describe performance in terms of input size, providing a valuable abstraction for understanding algorithmic efficiency.

While asymptotic analysis plays a crucial role in algorithm design and comparison, it often assumes idealized or average input conditions and abstracts away constant factors, memory behaviour, and data characteristics. In real-world scenarios, however, input data rarely follows purely random distributions. Practical datasets frequently exhibit properties such as partial ordering, repetition of values, skewed distributions, or structured patterns arising from user behaviour, sensor data, financial records, or incremental updates. These characteristics can significantly influence the actual runtime behaviour of sorting algorithms.

As a result, algorithms with superior theoretical time complexity do not always demonstrate the best performance in practice. For instance, simple algorithms with quadratic worst-case complexity may outperform more advanced divide-and-conquer algorithms when applied to nearly sorted data. Conversely, algorithms that are efficient on random inputs may degrade noticeably when exposed to adversarial or patterned input arrangements. Such discrepancies highlight the gap between theoretical performance guarantees and real-world execution behaviour.

This paper aims to bridge this gap by conducting a systematic experimental evaluation of sorting algorithms under diverse input data arrangements. Rather than focusing solely on input size, this study emphasizes how the structure and ordering of input data affect algorithmic performance. A range of sorting algorithms, from elementary techniques such as Bubble Sort, Selection Sort, and Insertion Sort to more advanced algorithms including Merge Sort, Quick Sort, and Heap Sort, are implemented and analyzed.

All algorithms are implemented in C++ to minimize runtime overhead and allow precise measurement of performance metrics. Multiple datasets are generated to represent realistic and adversarial input conditions, including random, sorted, reverse-sorted, nearly sorted, duplicate-heavy, and patterned datasets. Performance is evaluated using execution time, number of comparisons, and number of data movements, providing a more comprehensive understanding of algorithm behaviour beyond asymptotic bounds.

The primary objective of this study is not to challenge the correctness of theoretical time complexity analysis, but to demonstrate its limitations when used in isolation. By empirically analyzing sorting algorithms across varied input arrangements, this paper seeks to provide practical insights into algorithm selection and highlight the importance of data characteristics in real-world computing environments.

## 2.Related Work:

Sorting algorithms have been extensively studied in computer science literature [1], [2], [5]; with foundational work focusing on their theoretical properties, correctness, and asymptotic time complexity. Classical algorithm textbooks provide detailed analyses of sorting techniques such as Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort, primarily evaluating their performance using worst-case, average-case, and best-case complexity measures. These analyses offer important theoretical guarantees but generally assume abstract machine models and simplified input distributions.

Several studies have explored empirical performance evaluation of algorithms to complement theoretical analysis. Experimental algorithmics emphasizes measuring real execution time, memory usage, and constant factors that are often ignored in asymptotic analysis [4]. Prior

empirical studies have shown that hardware architecture, cache behaviour, and compiler optimizations can significantly influence observed performance [9], even for algorithms with identical theoretical complexity.

Some research has investigated adaptive and hybrid sorting algorithms designed to exploit specific input characteristics. For example, algorithms such as TimSort leverage existing order within data to improve practical performance [8], particularly on partially sorted datasets. These works demonstrate that awareness of input structure can lead to substantial performance improvements over traditional approaches that treat all inputs uniformly.

However, much of the existing literature either focuses on advanced industrial sorting algorithms or evaluates performance primarily with respect to input size. Comparatively fewer studies systematically analyze how different input data arrangements, such as presortedness, duplication, or patterned distributions, affect a broad range of sorting algorithms across multiple complexity classes. In particular, student-level and introductory research often limits evaluation to random datasets, overlooking realistic data patterns commonly encountered in practical applications.

This paper builds upon prior theoretical and empirical research by providing a structured experimental comparison of sorting algorithms under diverse input data arrangements. By examining both elementary and advanced algorithms across multiple dataset characteristics, this study contributes a practical perspective on how data structure influences sorting performance beyond traditional asymptotic analysis.

### **3.Methodology**

This study adopts an experimental approach to evaluate the practical performance of sorting algorithms under varying input data arrangements. The methodology is designed to ensure fairness, reproducibility, and consistency across all experiments by controlling implementation details, dataset generation, and measurement techniques.

#### **3.1 Algorithms Selected**

To analyze sorting behaviour across different complexity classes, both elementary and advanced sorting algorithms are considered. The selected algorithms represent commonly taught and widely used techniques in computer science.

The following algorithms are implemented:

- Elementary Sorting Algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort
- Advanced Sorting Algorithms
  - Merge Sort
  - Quick Sort

- Heap Sort

These algorithms were chosen to compare how input data arrangement affects algorithms with different operational principles, including comparison-based, divide-and-conquer, and heap-based approaches.

All algorithms are implemented manually in C++ without relying on standard library sorting functions to maintain control over internal operations such as comparisons and swaps.

### 3.2 Implementation Details

All sorting algorithms are implemented in C++, chosen for its low-level memory control, predictable execution behaviour, and minimal runtime overhead. This allows accurate measurement of algorithmic performance without interference from language-level abstractions such as automatic garbage collection.

Each algorithm is implemented using a uniform coding style and data structure representation to ensure fair comparison. Care is taken to avoid unnecessary optimizations that could bias results toward specific algorithms. For algorithms involving recursion, such as Merge Sort and Quick Sort, recursion depth is monitored implicitly through input size control.

### 3.3 Input Data Arrangements

To capture the impact of data characteristics on sorting performance, multiple input datasets are systematically generated. Each dataset represents a distinct arrangement commonly observed in practical applications.

The following input data arrangements are used:

1. Random Data: Uniformly distributed random integers.
2. Sorted Data: Data sorted in ascending order.
3. Reverse-Sorted Data: Data sorted in descending order.
4. Nearly Sorted Data: Data where a majority of elements are already in correct order, with a small percentage randomly shuffled.
5. Duplicate-Heavy Data: Datasets containing a limited number of unique values with frequent repetition.
6. All-Equal Data: Datasets where all elements have the same value.
7. Patterned Data: Structured patterns such as alternating high-low values.

These datasets are generated programmatically to eliminate manual bias and ensure repeatability across experiments.

### 3.4 Dataset Size and Scaling

To observe scalability trends, each dataset arrangement is tested across multiple input sizes such as 1000, 5000, 10000. Typical dataset sizes range from small inputs used to highlight algorithmic behaviour to larger inputs used to examine growth trends.

For each input arrangement, datasets are generated in increasing sizes to study how performance scales with both data size and structure. This approach allows separation of size-related effects from arrangement-related effects.

### 3.5 Performance Metrics

Performance evaluation is conducted using multiple metrics to provide a comprehensive understanding of algorithm behaviour:

- Execution Time: Measured using high-resolution timers to capture actual runtime.
- Number of Comparisons: Counts the total comparisons performed during sorting.
- Number of Data Movements: Tracks swaps or element shifts, particularly relevant for insertion-based algorithms.

Using multiple metrics enables analysis beyond raw execution time and helps explain observed performance differences across datasets.

### 3.6 Experimental Environment

All experiments are conducted on the same hardware and software environment to maintain consistency. The system configuration, compiler version, and compilation settings remain fixed throughout the study.

To minimize measurement noise, each experiment is executed multiple times, and average values are recorded. No external background tasks are intentionally introduced during execution to reduce variability.

### 3.7 Experimental Procedure

The experimental process follows these steps:

1. Generate the input dataset for a specific arrangement and size.
2. Apply each sorting algorithm independently to a copy of the dataset.
3. Record execution time, comparisons, and data movements.
4. Repeat the experiment multiple times and compute average values.
5. Compare results across algorithms and input arrangements.

This systematic procedure ensures consistency and fairness across all experiments.

To ensure reproducibility and transparency of results, the complete source code used in this study, including dataset generation modules, sorting algorithm implementations, and performance measurement utilities, is publicly available in a GitHub repository. The repository contains detailed instructions for compilation and execution, enabling independent verification of the experimental results.

Repository link: <https://github.com/shyamdotexe/input-arrangement-sorting-analysis>

### 3.8 Limitations

This study focuses exclusively on comparison-based sorting algorithms implemented in C++. Non-comparison-based algorithms such as Counting Sort and Radix Sort are excluded due to their dependence on input value ranges rather than data arrangement alone. Additionally, hardware-level effects such as cache misses and branch prediction are not explicitly measured, though their impact is reflected indirectly in execution time.

## 4. Results and Analysis

### 4.1 Bubble Sort

Bubble Sort serves as a baseline elementary sorting algorithm due to its well-known quadratic time complexity and deterministic comparison structure. This subsection evaluates its sensitivity to input data arrangement.

#### 4.1.1 Runtime Variation Across Dataset Types

To evaluate the impact of input arrangement at fixed scale, execution time was measured for input size  $n = 10000$  across all dataset types.

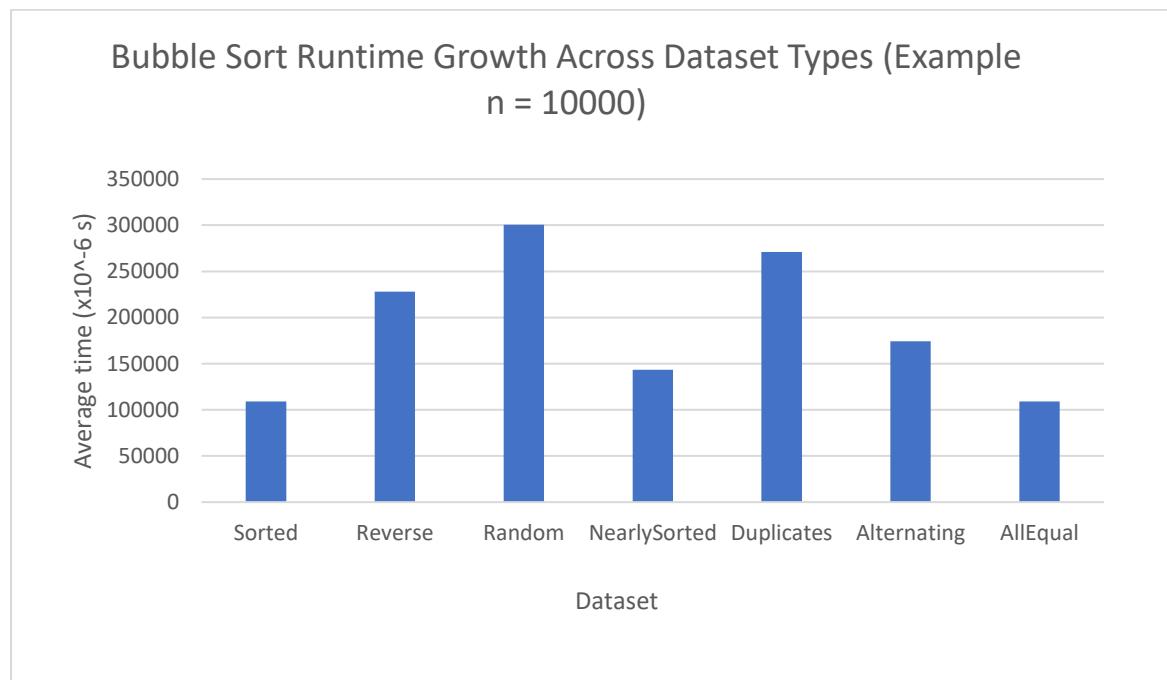


Figure 1: Execution Time of Bubble Sort Across Different Dataset Arrangements ( $n = 10000$ )

The results indicate that execution time varies across dataset arrangements. Reverse-sorted and random datasets exhibit higher runtimes, while sorted and all-equal datasets produce comparatively lower execution times. This variation is primarily attributed to differences in swap frequency rather than comparison count.

Although runtime differences are observable, they remain within the same order of growth, suggesting limited adaptiveness of Bubble Sort to input structure.

#### 4.1.2 Scalability Analysis

To examine growth behaviour, runtime was analyzed across increasing input sizes for all dataset arrangements.

### Bubble Sort Runtime Growth Across Dataset Types

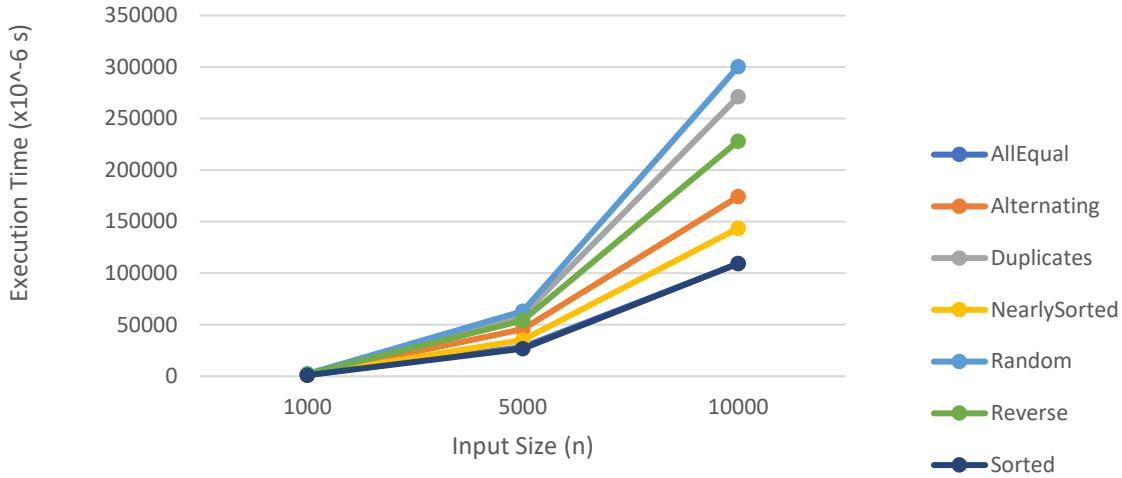


Figure 2: Runtime Growth of Bubble Sort Across Dataset Arrangements

As shown in Figure 2, execution time increases steeply with input size across all dataset types. The growth pattern closely approximates quadratic scaling, as doubling the input size results in approximately fourfold increases in runtime. Notably, the curves for different dataset types exhibit similar slopes, indicating that input arrangement affects constant factors but not asymptotic growth behaviour.

**This confirms that Bubble Sort's time complexity remains  $O(n^2)$  regardless of data ordering.**

#### 4.1.3 Comparison Count Analysis

To validate theoretical expectations, the number of comparisons was analyzed across dataset types and input sizes.

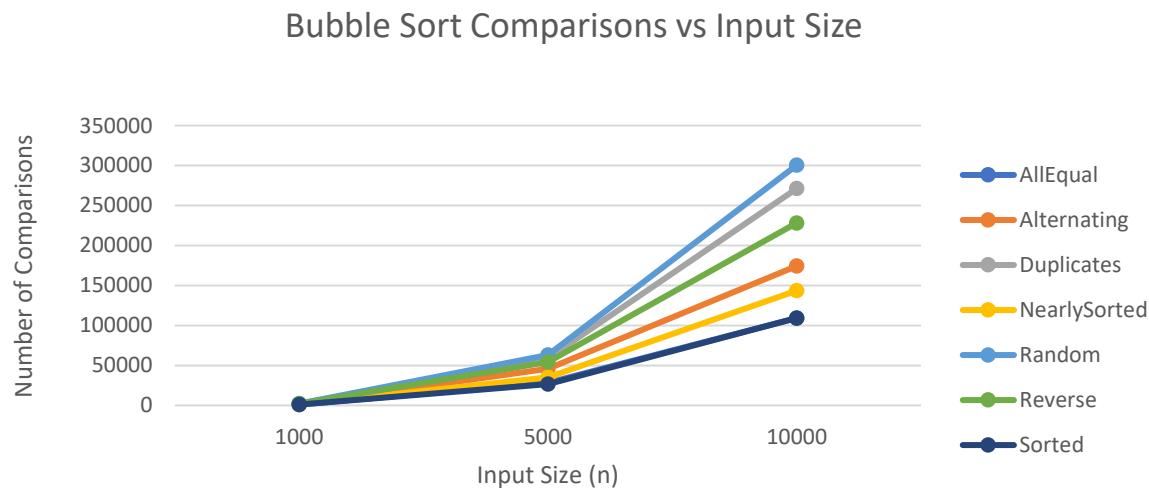


Figure 3: Comparison Count of Bubble Sort as a Function of Input Size

The comparison count remains identical across all dataset arrangements for a given input size. This confirms that Bubble Sort performs a fixed number of comparisons determined solely by input size, independent of data ordering.

The empirical results therefore align precisely with theoretical analysis, which predicts  $n(n-1)/2$  comparisons for Bubble Sort in all cases.

#### 4.1.4 Data Movement Analysis

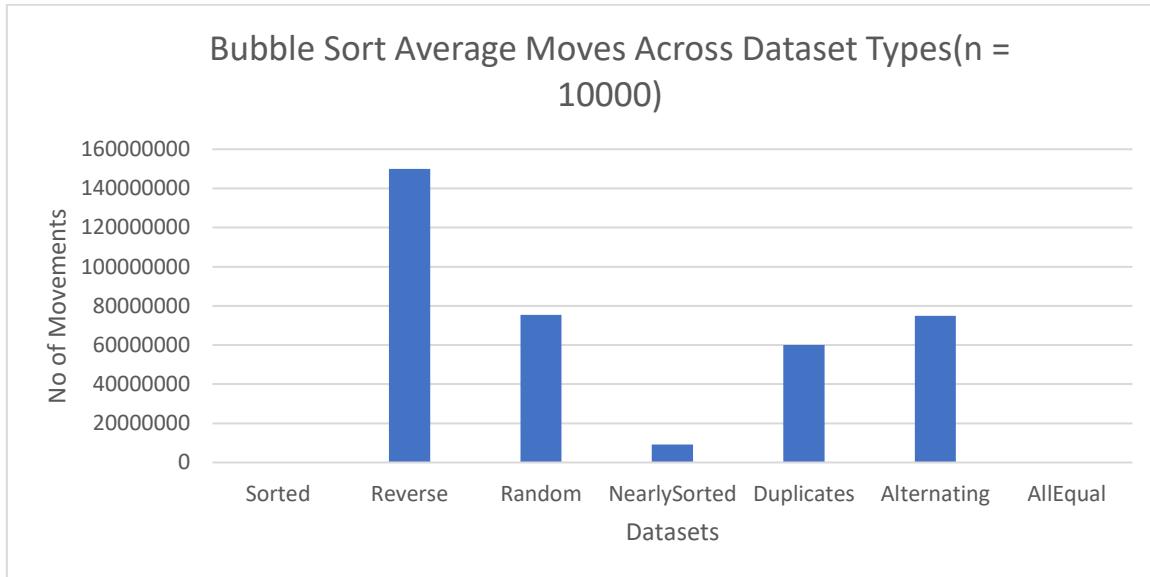


Figure 4: Data Movements of Bubble Sort Across Dataset Types (n = 10000)

The number of data movements in Bubble Sort varied significantly across dataset arrangements.

As illustrated in Figure 4 (n = 10000), sorted and all-equal datasets required zero swaps, whereas reverse-sorted input resulted in the maximum number of element exchanges, approaching 150 million movements. Random, alternating, and duplicate-heavy datasets produced intermediate swap counts, while nearly sorted input required substantially fewer movements. Although the comparison count remains fixed for all datasets, runtime differences are largely explained by variation in swap frequency. This confirms that Bubble Sort is non-adaptive in comparisons but sensitive to inversion density in terms of data movement.

#### 4.1.5 Discussion

From the above observations, Bubble Sort demonstrates:

- Input-insensitive comparison count
- Dataset-dependent swap frequency
- Consistent quadratic scaling across input sizes
- Limited practical adaptiveness

Although the comparison count remains fixed for a given input size, execution time varies significantly across dataset arrangements due to differences in data movement frequency. Reverse-sorted inputs produce maximal swap counts, whereas sorted and all-equal datasets require no element exchanges. Despite these variations in element movement, the overall growth trend remains quadratic. Thus, Bubble Sort exhibits no deviation from its theoretical complexity bounds, with runtime

variation primarily attributable to differences in swap density rather than changes in asymptotic behaviour.

## 4.2 Insertion Sort

Insertion Sort is widely recognized as an adaptive sorting algorithm whose performance strongly depends on the initial ordering of input data. Unlike Bubble and Selection Sort, its best-case complexity is linear, while its average and worst-case complexities are quadratic. This subsection examines how input arrangement influences its practical behaviour.

### 4.2.1 Runtime Variation Across Dataset Types

To evaluate sensitivity at fixed scale, execution time was measured for input size  $n = 10000$  across all dataset arrangements.

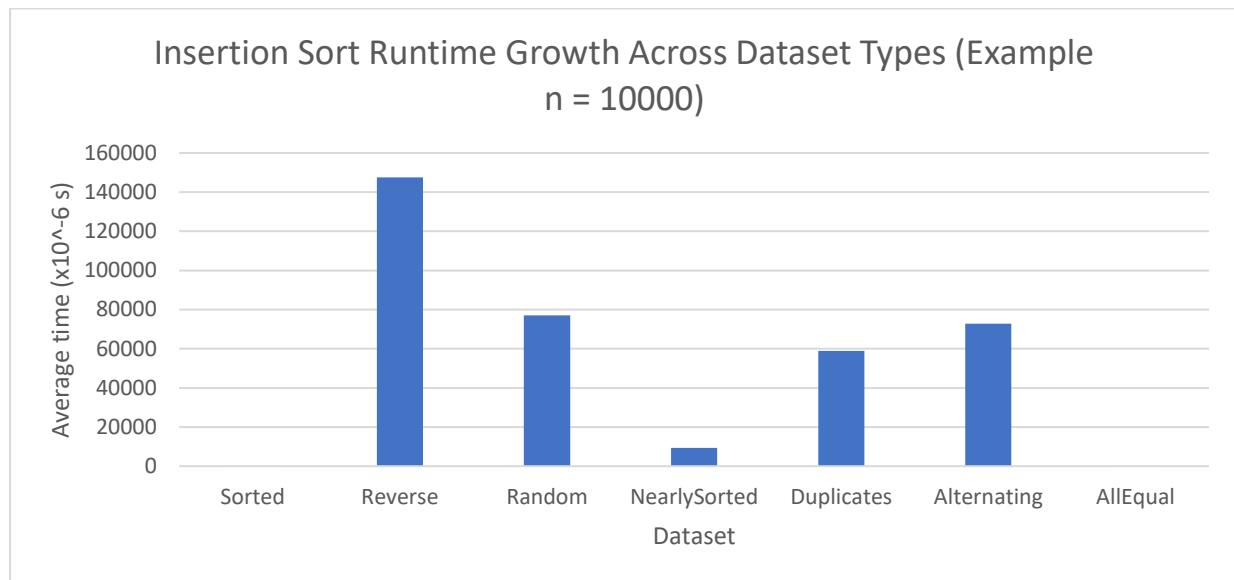


Figure 5: Execution Time of Insertion Sort Across Different Dataset Arrangements ( $n = 10000$ )

As shown in Figure 5, runtime varies dramatically across dataset types. Sorted and all-equal datasets exhibit extremely low execution times, while reverse-sorted datasets produce significantly higher runtimes. Random and duplicate-heavy datasets fall between these extremes.

This substantial variation reflects the adaptive nature of Insertion Sort. When the input is already sorted or nearly sorted, minimal element shifts are required, resulting in near-linear performance. In contrast, reverse ordering forces each element to be shifted across the entire sorted portion of the array, producing quadratic behaviour.

Unlike Bubble Sort, the runtime differences here span multiple orders of magnitude, clearly demonstrating strong input sensitivity.

### 4.2.2 Scalability Analysis

To further examine growth behaviour, execution time was analyzed across increasing input sizes for all dataset types.

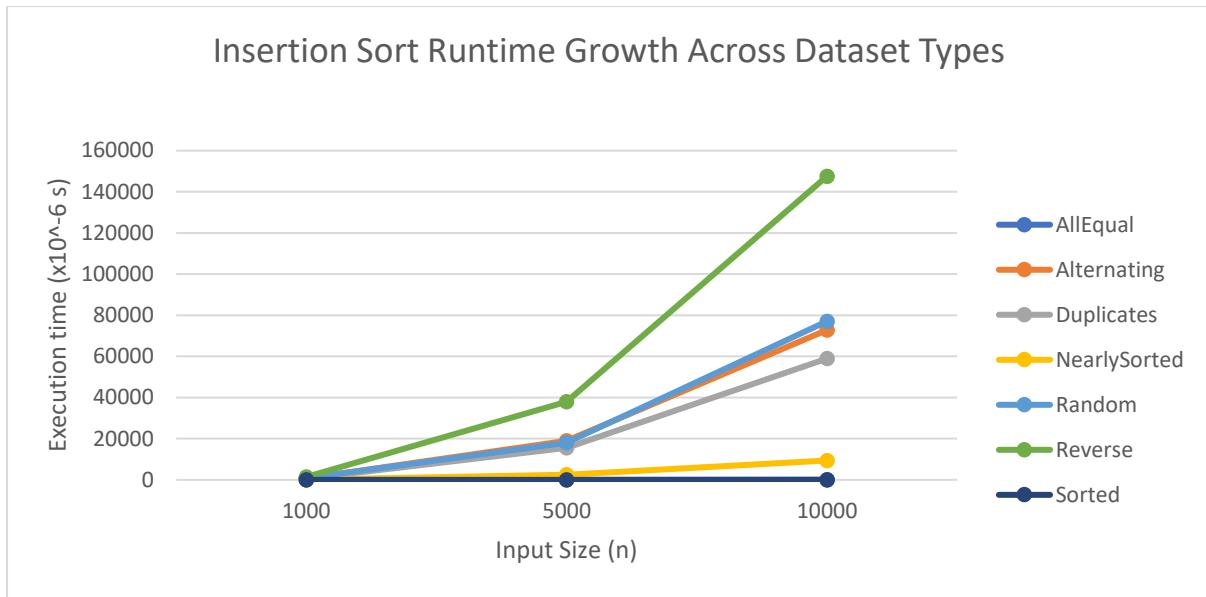


Figure 6: Runtime Growth of Insertion Sort as a Function of Input Size Across Dataset Arrangements

Figure 6 reveals distinct growth patterns depending on dataset arrangement. For sorted and all-equal datasets, the runtime increases approximately linearly with input size, consistent with the theoretical best-case complexity  $O(n)$ . However, for reverse-sorted and random datasets, runtime exhibits clear quadratic scaling.

The divergence between curves becomes more pronounced as input size increases, indicating that input arrangement not only affects constant factors but fundamentally alters the growth behavior of the algorithm.

This contrast strongly validates theoretical expectations regarding the best-case and worst-case complexity of Insertion Sort.

#### 4.2.3 Comparison Count Analysis

To validate theoretical complexity, the number of comparisons was examined across dataset types and input sizes.

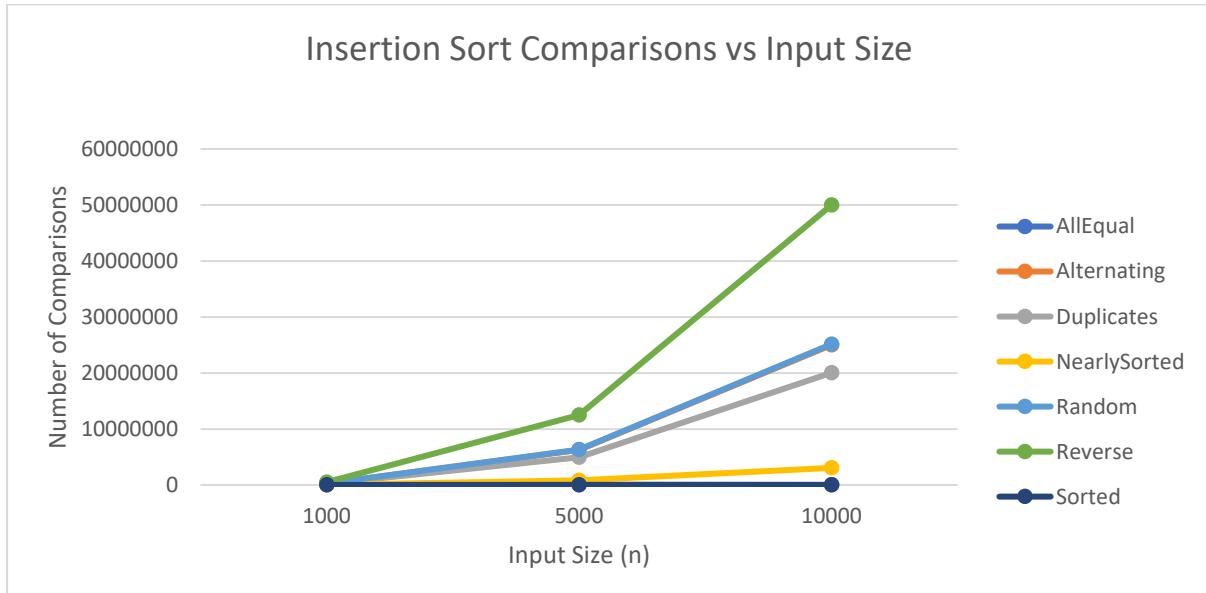


Figure 7: Number of Comparisons Performed by Insertion Sort as a Function of Input Size

As shown in Figure 7, sorted and all-equal datasets exhibit near-linear growth in comparison count, while reverse-sorted and random datasets display quadratic growth. Nearly sorted datasets fall between these extremes, further confirming the algorithm's adaptiveness.

Unlike Bubble Sort, where comparison count was independent of input arrangement, Insertion Sort's comparison count directly reflects the level of disorder in the dataset.

#### 4.2.4 Data Movement Analysis

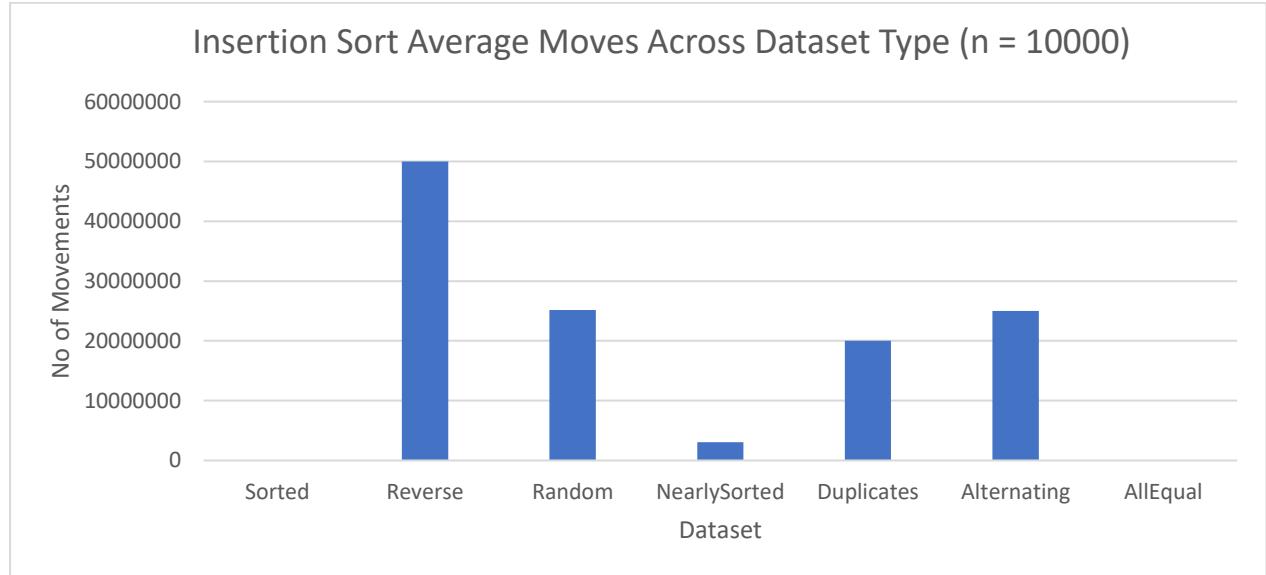


Figure 8: Data Movements of Insertion Sort Across Dataset Types (n = 10000)

The number of data movements in Insertion Sort exhibited strong dependence on input arrangement. As shown in Figure 8 (n = 10000), sorted and all-equal datasets required only 19,998 element shifts, reflecting near-linear behaviour. In contrast, reverse-sorted input resulted in approximately 50,014,998 movements, consistent with worst-case quadratic growth. Random and duplicate-heavy datasets produced intermediate movement counts, while nearly sorted input required significantly fewer shifts compared to fully disordered data. These results confirm the adaptive nature of Insertion Sort, where the number of element movements is directly influenced by the degree of presortedness in the dataset.

#### 4.2.5 Discussion

The empirical findings demonstrate that Insertion Sort is highly sensitive to input arrangement. Unlike algorithms with fixed structural behaviour, Insertion Sort adapts dynamically based on the degree of presortedness in the data.

Specifically:

- Sorted and all-equal datasets result in near-linear runtime, comparison growth, and minimal data movements.
- Reverse-sorted datasets trigger worst-case quadratic behaviour, with both comparisons and element shifts increasing dramatically.
- Nearly sorted datasets exhibit intermediate performance, reflecting reduced inversion density and lower movement frequency compared to fully disordered inputs.

These results highlight a critical insight: asymptotic complexity alone does not fully capture practical performance. In Insertion Sort, execution time is closely correlated with the number of element movements required to reposition out-of-order elements. As inversion density increases, both comparisons and shifts increase proportionally, leading to quadratic degradation.

Although Insertion Sort has a quadratic worst-case bound, it can outperform asymptotically faster algorithms such as Merge Sort or Heap Sort when applied to structured or partially ordered data. Thus, Insertion Sort represents a clear example of how input arrangement and movement density significantly influence algorithm efficiency in real-world scenarios.

### 4.3 Selection Sort

Selection Sort is a simple comparison-based sorting algorithm characterized by a fixed number of comparisons regardless of input arrangement. Unlike adaptive algorithms such as Insertion Sort, Selection Sort does not exploit existing order in the dataset. This subsection evaluates its sensitivity to input structure.

#### 4.3.1 Runtime Variation Across Dataset Types

To evaluate the impact of input arrangement at fixed scale, execution time was measured for input size  $n = 10000$  across all dataset types.

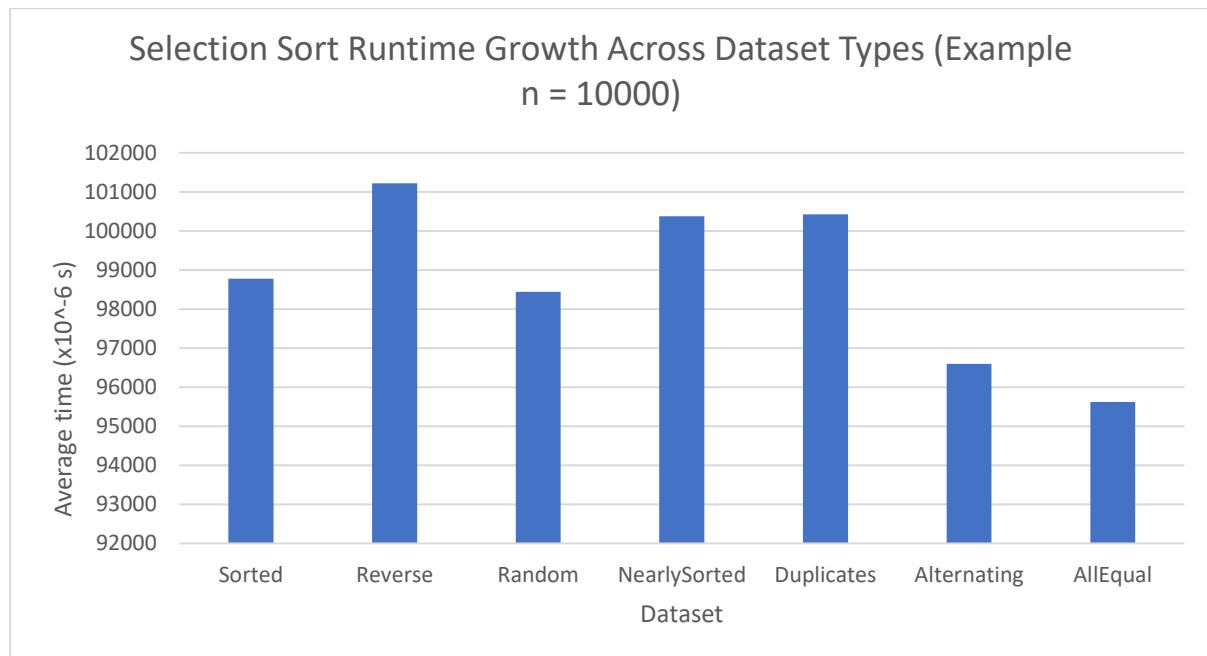


Figure 9: Execution Time of Selection Sort Across Different Dataset Arrangements ( $n = 10000$ )

As shown in Figure 9, execution time remains relatively consistent across all dataset arrangements. While minor variations are observable, the differences are significantly smaller compared to those observed in Insertion Sort.

Reverse-sorted, random, and nearly sorted datasets exhibit similar runtime behaviour, indicating that Selection Sort does not adapt to input ordering. The limited variation observed is primarily due to minor differences in swap operations and hardware-level effects rather than algorithmic structure.

#### 4.3.2 Scalability Analysis

To analyze growth behaviour, execution time was evaluated across increasing input sizes for all dataset arrangements.

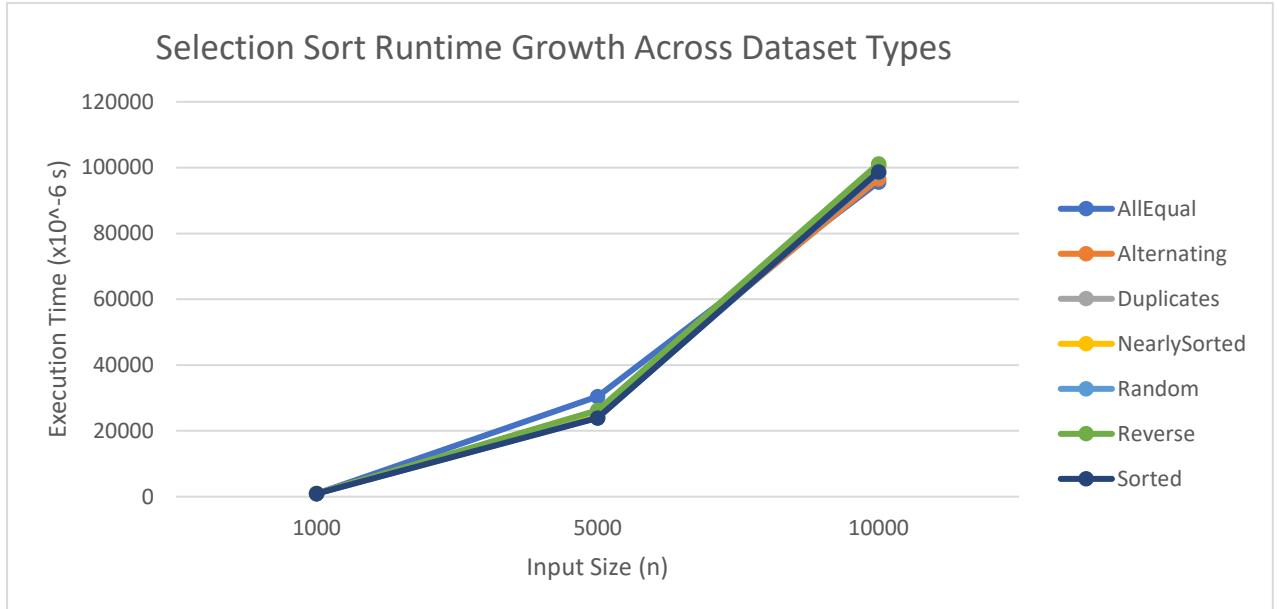


Figure 10: Runtime Growth of Selection Sort as a Function of Input Size Across Dataset Arrangements

Figure 10 illustrates that runtime increases quadratically with input size across all dataset types. The curves for different arrangements closely overlap, confirming that input structure does not influence the asymptotic growth pattern.

Doubling the input size results in approximately fourfold increases in runtime, consistent with the theoretical  $O(n^2)$  time complexity.

#### 4.3.3 Comparison Count Analysis

To validate theoretical expectations, the number of comparisons was analyzed across dataset types and input sizes.

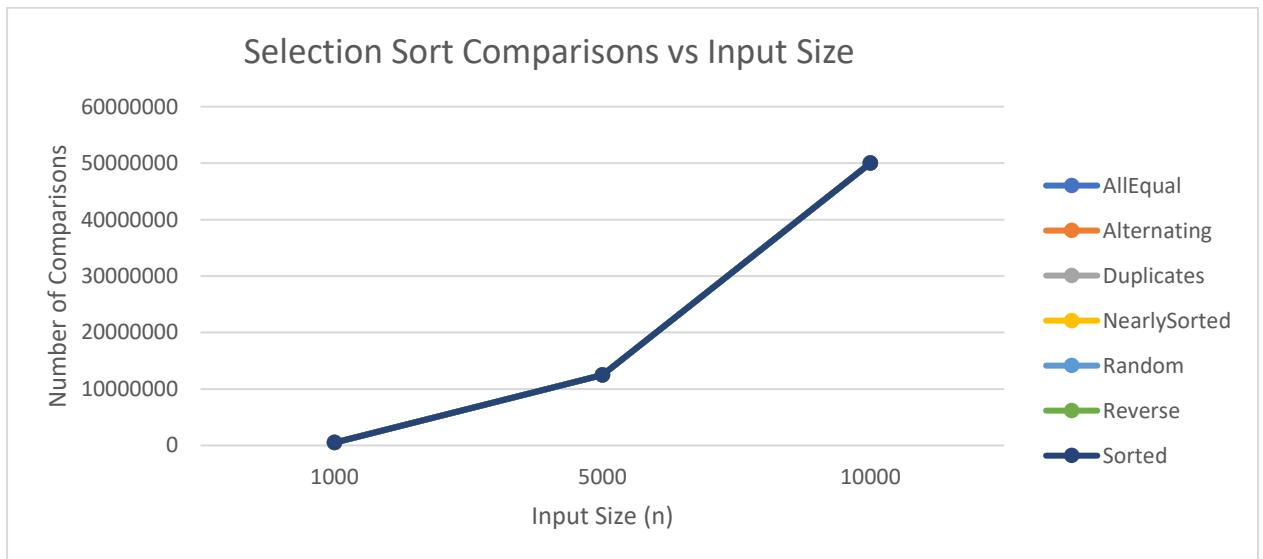


Figure 11: Number of Comparisons Performed by Selection Sort as a Function of Input Size

As shown in Figure 11, the number of comparisons is identical across all dataset arrangements for a given input size. This confirms that Selection Sort performs a fixed sequence of comparisons independent of input ordering.

Unlike Insertion Sort, which adapts based on disorder level, Selection Sort's comparison count is determined solely by input size

#### 4.3.4 Data Movement Analysis

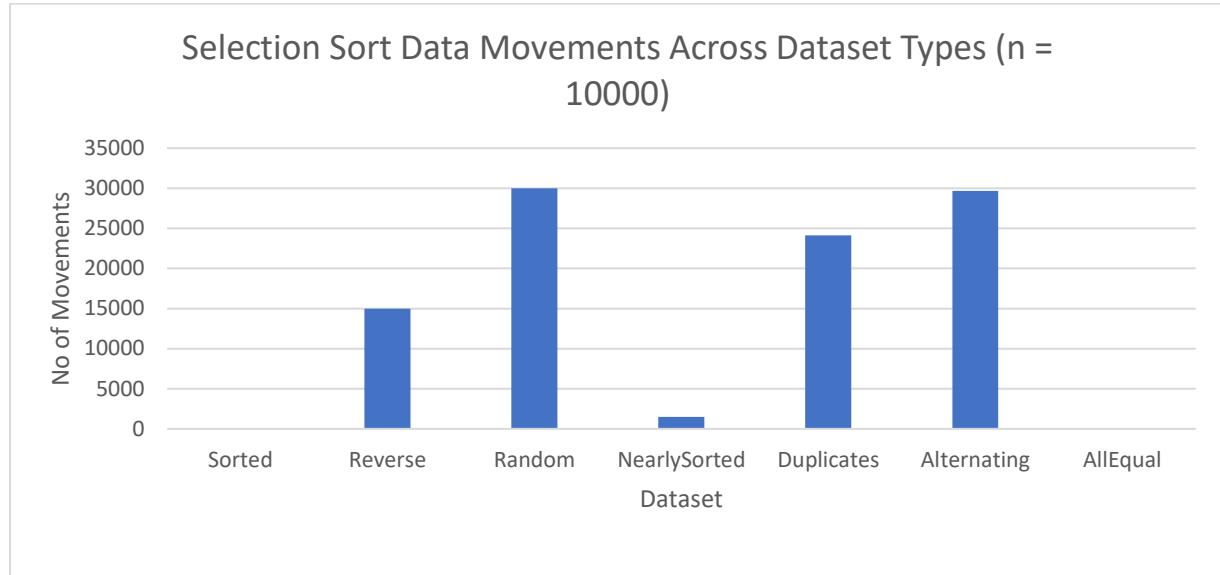


Figure 12: Data Movements of Selection Sort Across Dataset Types (n = 10000)

The number of data movements in Selection Sort remained relatively small and predictable across all dataset arrangements. As illustrated in Figure 12 (n = 10000), movement counts did not exceed 29,973 swaps even in the random case. Reverse-sorted input required approximately 15,000 swaps, while sorted and all-equal datasets required no element exchanges. Unlike Bubble and Insertion Sort, Selection Sort performs at most one swap per iteration, resulting in linear growth of movement count with respect to input size. This limited variation in swap frequency contributes to the algorithm's consistent runtime behaviour and confirms its non-adaptive structural characteristics.

#### 4.3.5 Discussion

The empirical results demonstrate that Selection Sort exhibits minimal sensitivity to input arrangement. Both runtime and comparison count remain largely unaffected by dataset structure, reflecting its fixed comparison pattern.

Although swap frequency varies slightly depending on the distribution of minimum elements, the total number of data movements remains limited and grows linearly with input size. These variations are minor compared to the quadratic comparison structure and do not significantly influence overall runtime behaviour.

Consequently, Selection Sort behaves as a fully non-adaptive algorithm, with practical performance closely aligned with its theoretical complexity predictions. Input arrangement influences constant factors only marginally and does not alter its asymptotic growth characteristics.

### 4.4 Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm with a theoretical time complexity of  $O(n \log n)$  in all cases. Unlike adaptive algorithms such as Insertion Sort, its structure does not depend on input ordering. This subsection evaluates whether practical execution reflects this theoretical stability.

#### 4.4.1 Runtime Variation Across Dataset Types

To evaluate the impact of input arrangement at fixed scale, execution time was measured for input size  $n = 10000$  across all dataset types.

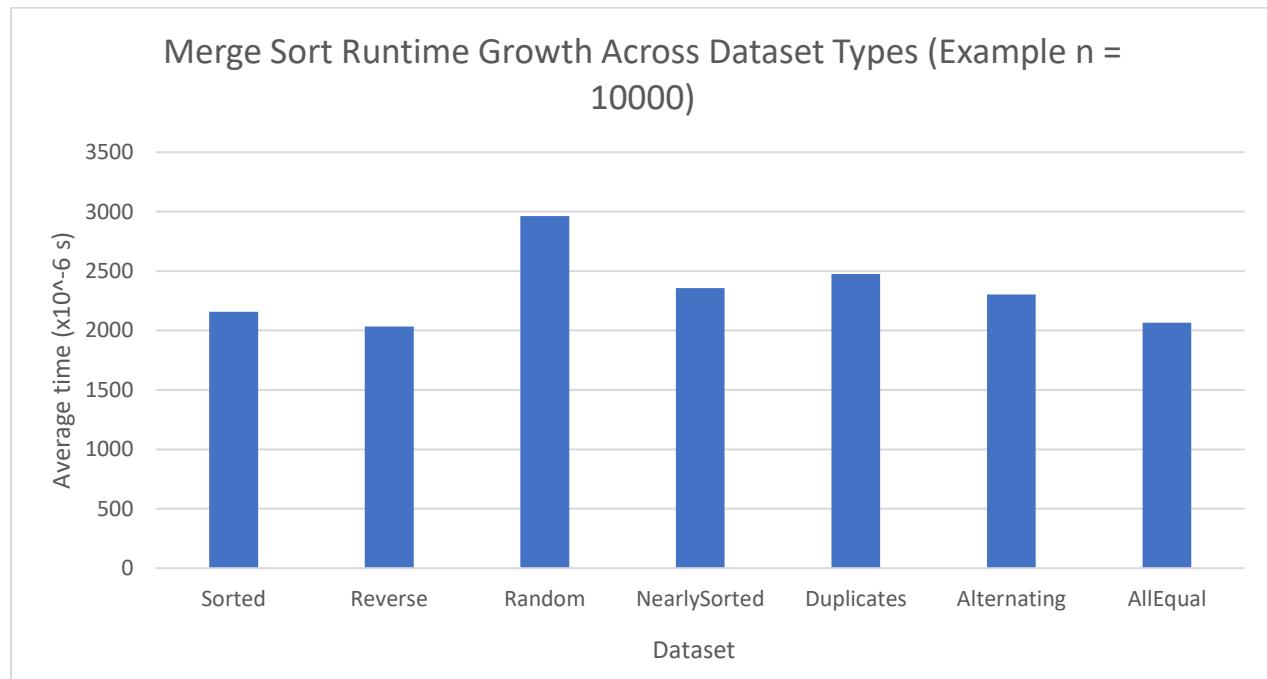


Figure 13: Execution Time of Merge Sort Across Different Dataset Arrangements ( $n = 10000$ )

As shown in Figure 13, runtime differences across dataset arrangements are minimal. Sorted, reverse-sorted, random, duplicate-heavy, and patterned datasets exhibit comparable execution times.

Unlike Insertion Sort and Quick Sort, no dataset arrangement produces dramatic degradation in performance. This stability reflects Merge Sort's structural independence from input ordering, as the algorithm recursively divides the dataset irrespective of its initial arrangement.

#### 4.4.2 Scalability Analysis

To analyze growth behaviour, execution time was evaluated across increasing input sizes for all dataset arrangements.

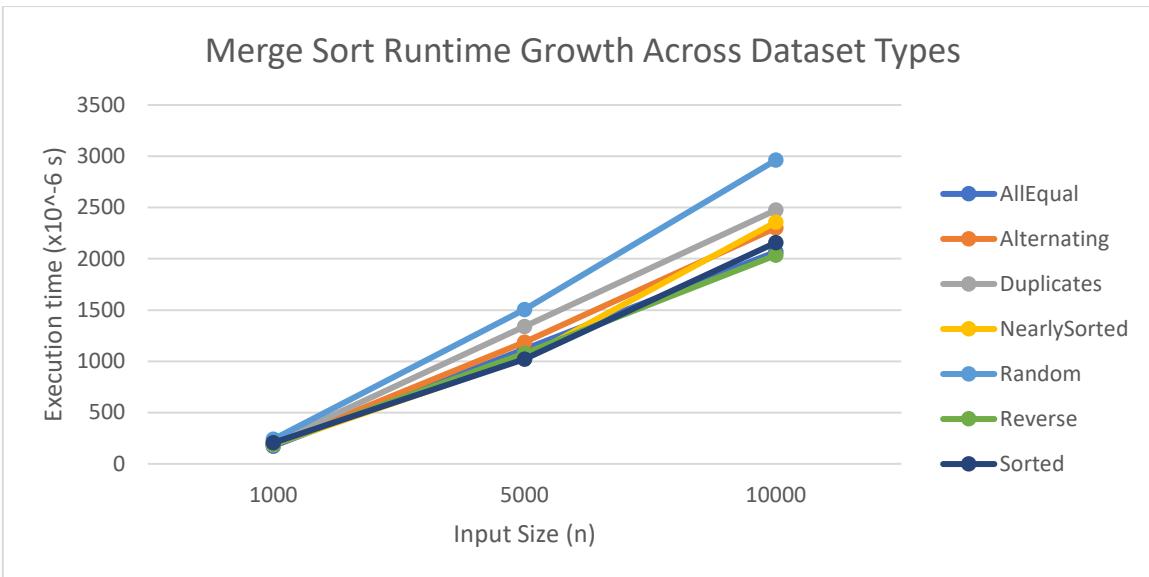


Figure 14: Runtime Growth of Merge Sort as a Function of Input Size Across Dataset Arrangements

Figure 14 illustrates that runtime increases consistently with input size across all dataset types. The growth pattern closely approximates  $n \log n$  scaling. The curves for different dataset arrangements are nearly parallel, indicating that input structure has negligible influence on asymptotic behavior.

Doubling input size results in a growth factor significantly lower than quadratic algorithms, confirming theoretical efficiency.

#### 4.4.3 Comparison Count Analysis

To further validate theoretical expectations, the number of comparisons was examined across dataset types and input sizes.

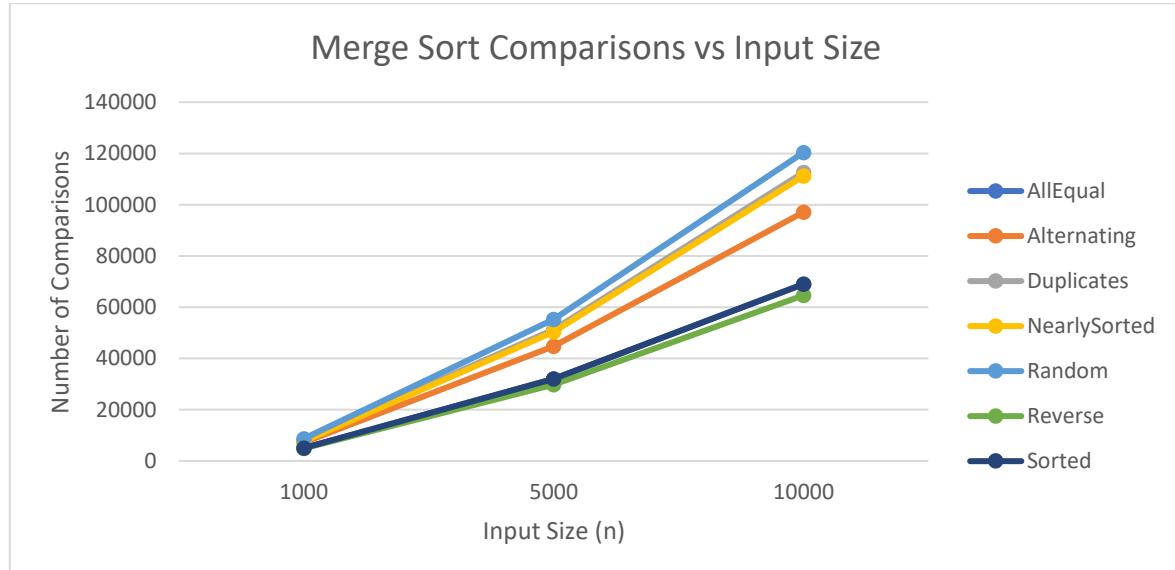


Figure 15: Number of Comparisons Performed by Merge Sort as a Function of Input Size

The comparison count grows consistently across dataset arrangements, following the expected  $n \log n$  trend. Unlike adaptive algorithms, the comparison count is not significantly influenced by input structure.

This confirms that Merge Sort maintains predictable behaviour regardless of dataset arrangement.

#### 4.4.4 Data Movement Analysis

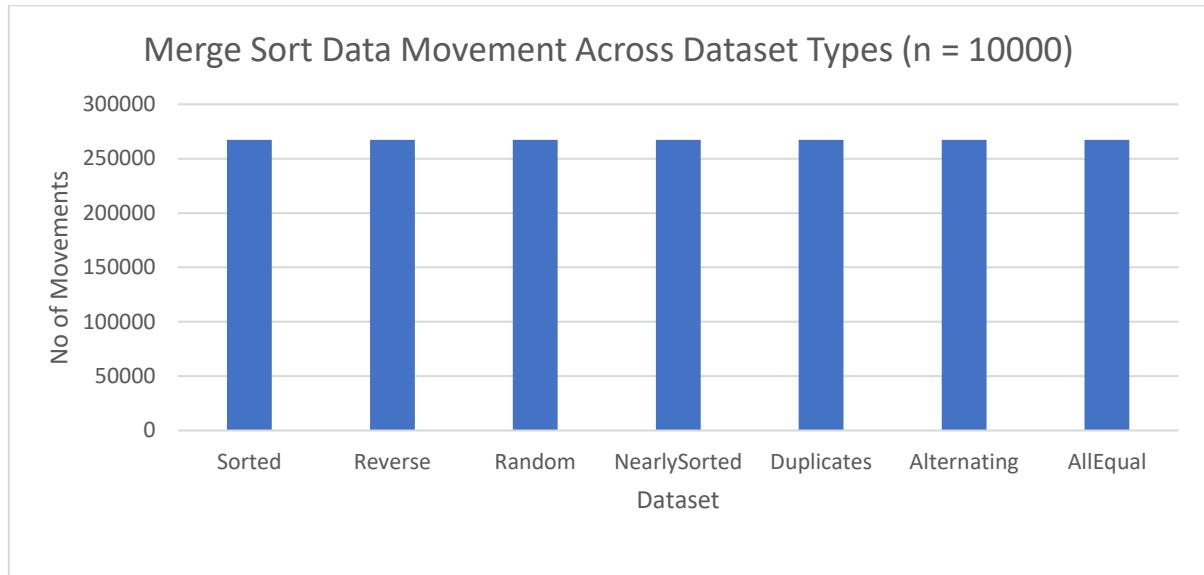


Figure 16: Data Movements of Merge Sort Across Dataset Types (n = 10000)

Merge Sort exhibited consistent data movement counts across all dataset arrangements. As shown in Figure 16 (n = 10000), the number of element movements remained constant at 267,232 regardless of whether the input was sorted, reverse-sorted, random, or duplicate-heavy. Similar consistency was observed for smaller input sizes, with 19,952 movements for n = 1000 and 123,616 for n = 5000.

Since the merge operation systematically copies elements during each recursive stage, the number of data movements depends solely on input size rather than input structure. This confirms that Merge Sort is structurally independent of presortedness and aligns closely with its theoretical  $O(n \log n)$  complexity.

#### 4.4.5 Discussion

The empirical results demonstrate that Merge Sort exhibits strong structural stability. Execution time, comparison count, and data movement frequency remain largely unaffected by dataset arrangement, with performance determined primarily by input size rather than input structure.

Unlike quadratic algorithms, Merge Sort avoids worst-case degradation under reverse-sorted or patterned inputs. Both comparison and movement counts follow predictable  $O(n \log n)$  growth across all dataset types, reflecting its divide-and-conquer structure.

However, despite its theoretical efficiency, Merge Sort does not always achieve the lowest execution time on highly structured datasets such as sorted inputs, where adaptive algorithms like Insertion Sort may outperform it due to reduced element movement requirements.

Thus, Merge Sort represents a robust and reliable algorithm whose practical behaviour closely aligns with theoretical complexity predictions while remaining largely insensitive to input arrangement.

### 4.5 Quick Sort

Quick Sort is a divide-and-conquer sorting algorithm with an average-case time complexity of  $O(n \log n)$  but a worst-case complexity of  $O(n^2)$ . Its performance depends heavily on pivot selection and

input arrangement. This subsection examines how different dataset structures influence its practical behaviour.

#### 4.5.1 Runtime Variation Across Dataset Types

To evaluate the impact of input arrangement at fixed scale, execution time was measured for input size  $n = 10000$  across all dataset types.

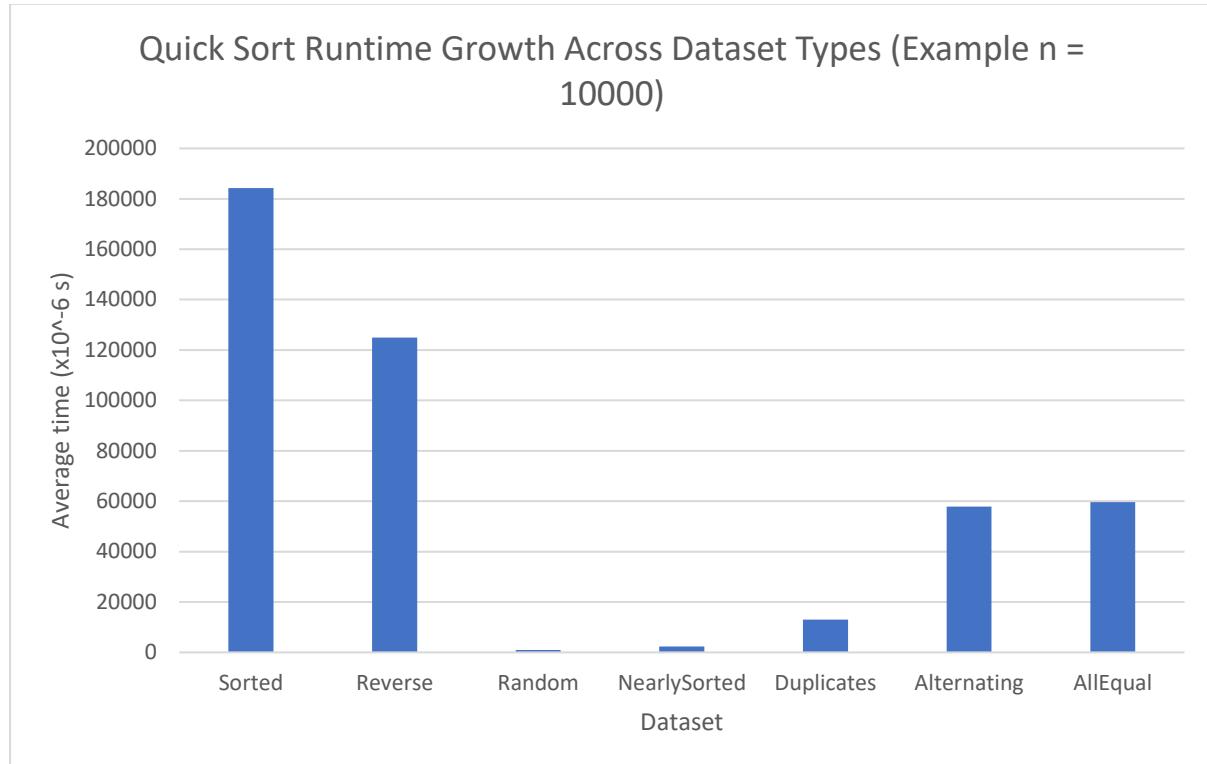


Figure 17: Execution Time of Quick Sort Across Different Dataset Arrangements ( $n = 10000$ )

As shown in Figure 17, Quick Sort exhibits significant runtime variation across dataset types. Random datasets produce the lowest execution times, while sorted and reverse-sorted datasets result in dramatically higher runtimes.

This sharp contrast is due to the pivot selection strategy used in the implementation. When the input is already sorted or nearly sorted, partitioning becomes highly unbalanced, leading to deep recursion and quadratic behaviour.

Unlike Merge Sort, Quick Sort demonstrates strong sensitivity to input structure, with runtime differences spanning multiple orders of magnitude.

#### 4.5.2 Scalability Analysis

To analyze growth behaviour, execution time was evaluated across increasing input sizes for all dataset arrangements.

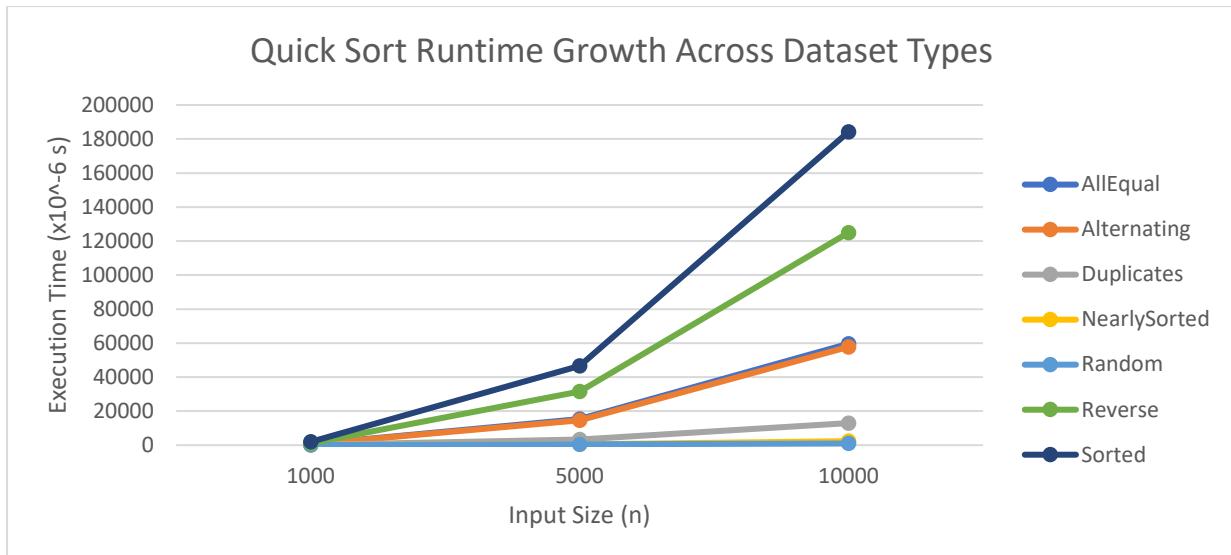


Figure 18: Runtime Growth of Quick Sort as a Function of Input Size Across Dataset Arrangements

Figure 18 reveals two distinct growth patterns. For random datasets, runtime follows an approximately  $n \log n$  trend, consistent with theoretical average-case complexity. However, for sorted and reverse-sorted datasets, runtime exhibits near-quadratic scaling.

As input size increases, the divergence between dataset curves becomes more pronounced. This demonstrates that input arrangement does not merely affect constant factors but can fundamentally alter the growth class of the algorithm in practice.

#### 4.5.3 Comparison Count Analysis

To further validate theoretical behaviour, the number of comparisons was examined across dataset types and input sizes.

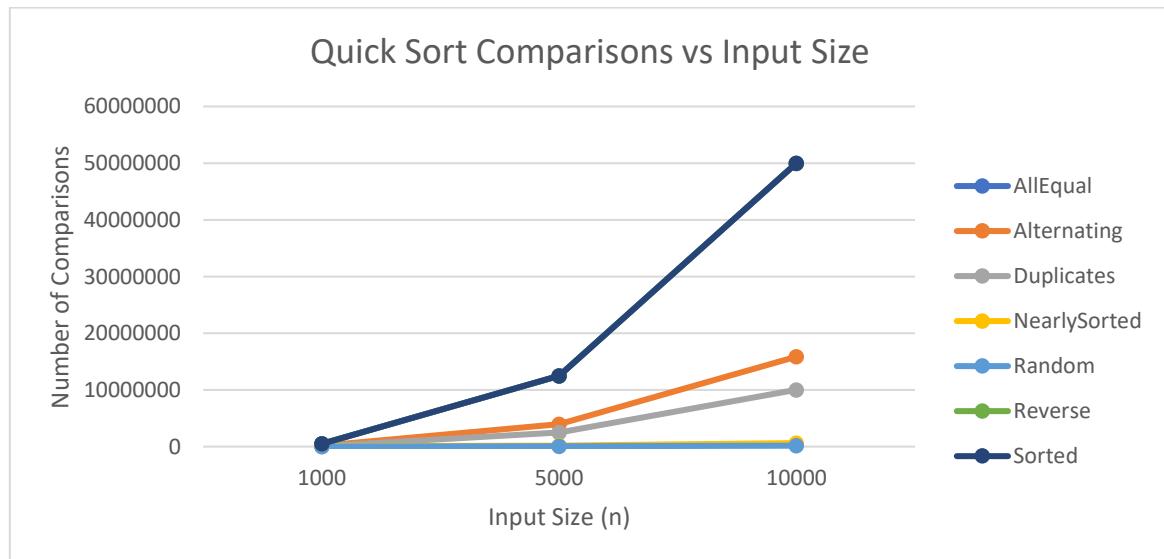


Figure 19: Number of Comparisons Performed by Quick Sort as a Function of Input Size

The comparison count confirms that random datasets exhibit  $n \log n$  growth, while sorted and reverse-sorted datasets approach quadratic comparison counts. Duplicate-heavy datasets may also show increased comparisons depending on pivot handling.

This highlights Quick Sort's structural dependence on partition balance.

#### 4.5.4 Data Movement Analysis

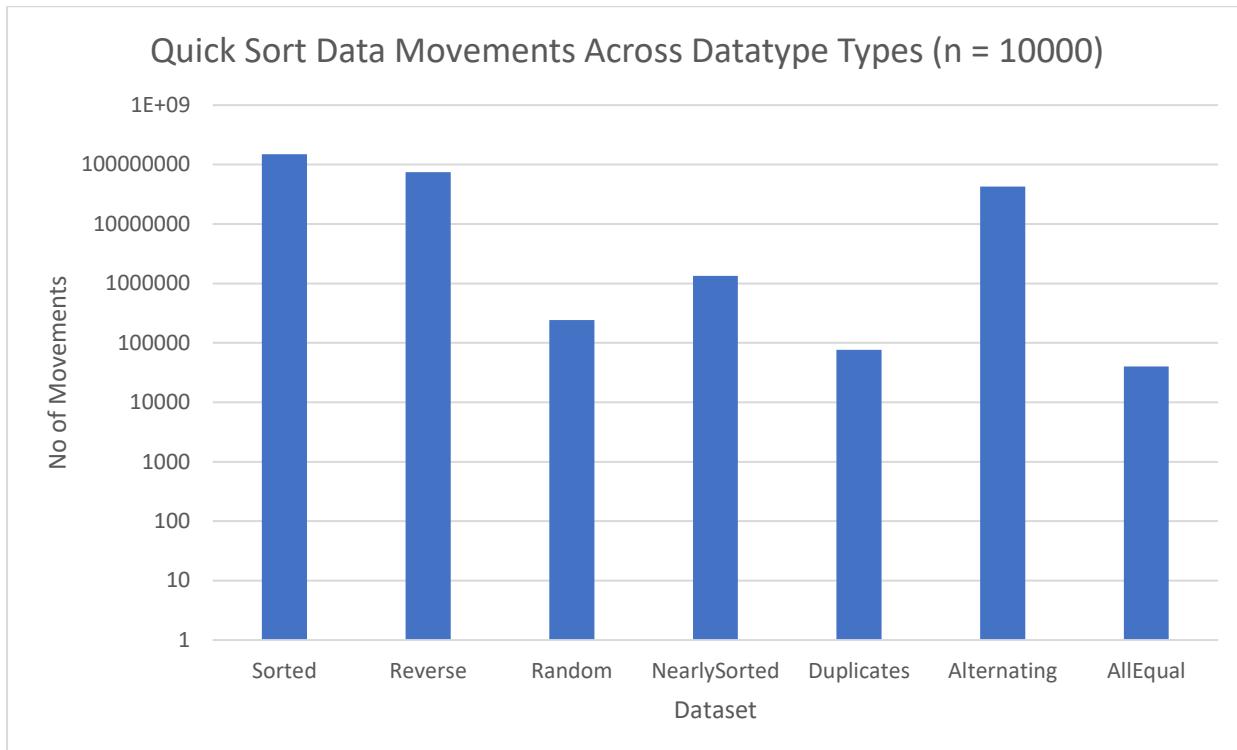


Figure 20: Data Movements of Quick Sort Across Dataset Types (n = 10000, Logarithmic Scale)

Quick Sort exhibited substantial variation in data movement counts across dataset arrangements. As illustrated in Figure 20 (logarithmic scale, n = 10000), sorted input resulted in approximately 150 million element movements, representing the highest observed value. Reverse-sorted and alternating datasets also produced significantly elevated movement counts. In contrast, random input required only 243,701 movements, reflecting balanced partitioning behavior. Duplicate-heavy and all-equal datasets resulted in comparatively low movement counts due to reduced partition exchanges.

The logarithmic representation highlights the multi-order magnitude differences between structured and unstructured inputs. These findings confirm that Quick Sort is highly sensitive to input arrangement, with data movement directly influenced by partition balance and pivot effectiveness.

#### 4.5.5 Discussion

The empirical results demonstrate that Quick Sort is highly sensitive to input arrangement. While it performs exceptionally well on random datasets, its performance degrades significantly under structured inputs when a non-random pivot strategy is used.

Both execution time and data movement frequency increase dramatically under sorted and reverse-sorted inputs due to unbalanced partitioning. In such cases, partition sizes become highly skewed, leading to deep recursion and near-quadratic growth in comparisons and element exchanges. In contrast, random inputs produce balanced partitions, resulting in efficient O(n log n) behaviour and moderate movement counts.

These findings strongly support the central thesis of this study: input data arrangement can substantially influence practical algorithm performance. Although Quick Sort maintains excellent average-case complexity, its real-world efficiency depends heavily on pivot selection and input structure.

Quick Sort therefore exemplifies how an algorithm with strong theoretical guarantees may exhibit severe performance degradation in structured or adversarial scenarios without careful implementation choices.

## 4.6 Heap Sort

Heap Sort is a comparison-based sorting algorithm that operates by constructing a binary heap and repeatedly extracting the maximum element. Its theoretical time complexity is  $O(n \log n)$  in all cases, independent of input arrangement. This subsection evaluates whether empirical results reflect this theoretical stability.

### 4.6.1 Runtime Variation Across Dataset Types

To evaluate the impact of input arrangement at fixed scale, execution time was measured for input size  $n = 10000$  across all dataset types.

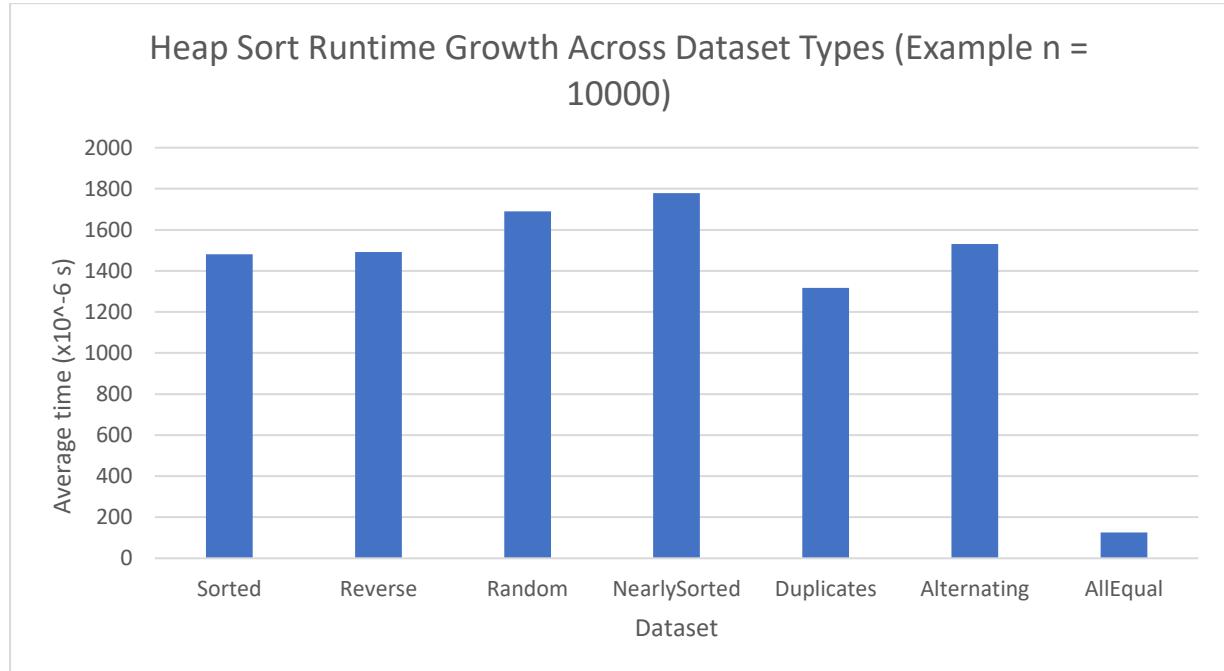


Figure 21: Execution Time of Heap Sort Across Different Dataset Arrangements ( $n = 10000$ )

As shown in Figure 21, runtime variation across dataset arrangements is relatively small compared to algorithms such as Insertion Sort and Quick Sort. Although minor differences are observable, no dataset produces catastrophic performance degradation.

Reverse-sorted, random, duplicate-heavy, and patterned datasets exhibit comparable execution times. Sorted and all-equal datasets may show slightly lower runtimes due to fewer structural adjustments during heap construction, but the differences remain limited.

This indicates that Heap Sort is largely insensitive to input ordering.

### 4.6.2 Scalability Analysis

To examine growth behaviour, execution time was analyzed across increasing input sizes for all dataset arrangements.

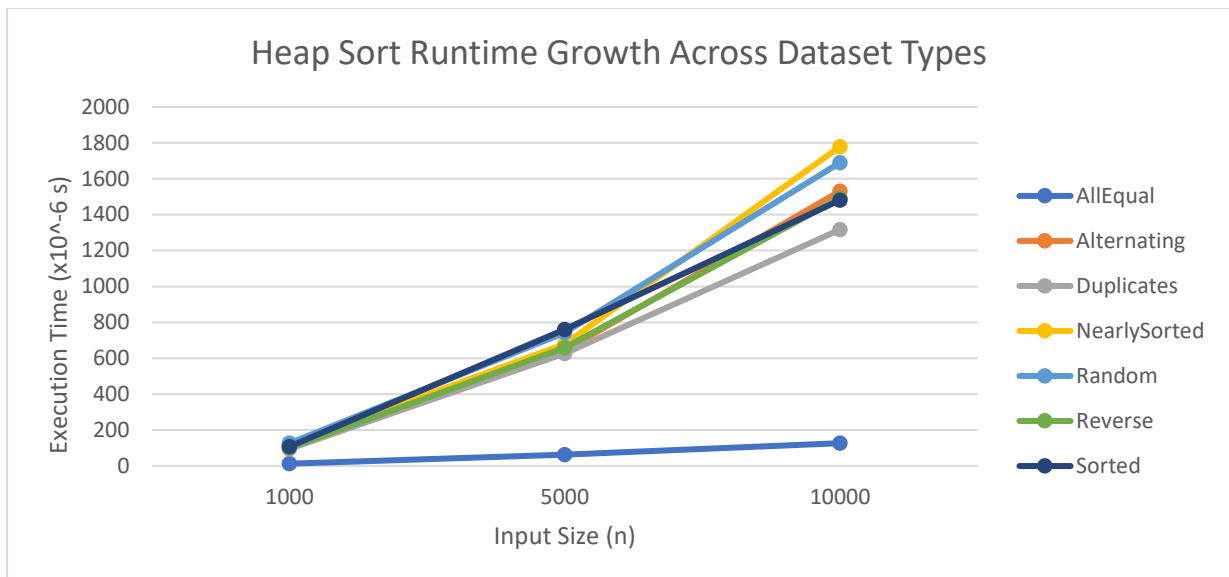


Figure 22: Runtime Growth of Heap Sort as a Function of Input Size Across Dataset Arrangements

Figure 22 shows consistent growth across all dataset types. The curves follow an approximately  $n \log n$  trend, with minimal divergence between dataset arrangements.

Unlike Quick Sort, Heap Sort does not exhibit worst-case quadratic behaviour under sorted or reverse-sorted inputs. As input size increases, growth remains predictable and stable.

#### 4.6.3 Comparison Count Analysis

To validate theoretical expectations, the number of comparisons was analyzed across dataset types and input sizes.

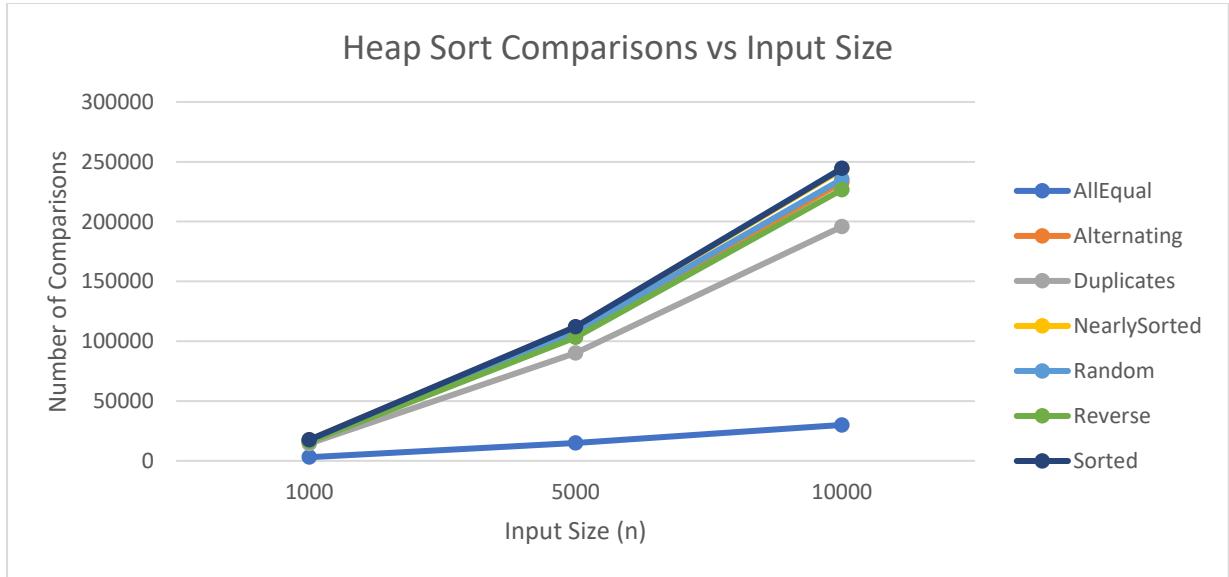


Figure 23: Number of Comparisons Performed by Heap Sort as a Function of Input Size

The comparison count grows consistently with input size across all dataset arrangements, aligning closely with theoretical  $O(n \log n)$  behaviour. Variations between datasets are minimal and do not alter the overall growth pattern.

This confirms that Heap Sort maintains structural independence from input ordering.

#### 4.6.4 Data Movement Analysis

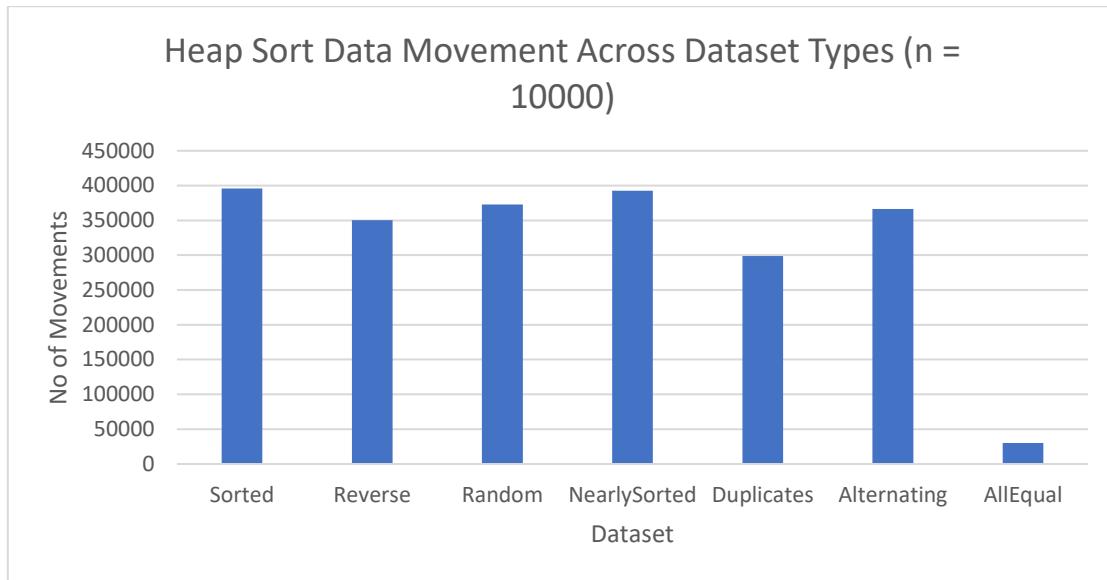


Figure 24: Data Movements of Heap Sort Across Dataset Types (n = 10000)

Heap Sort demonstrated relatively stable data movement counts across dataset arrangements. As illustrated in Figure 24 (n = 10000), most datasets required between approximately 350,000 and 395,000 element movements, indicating minimal variation due to input structure. Random, sorted, reverse-sorted, nearly sorted, and alternating datasets exhibited comparable movement counts. Duplicate-heavy datasets showed slightly reduced movements, while all-equal input resulted in the lowest count at approximately 29,997 movements.

Since Heap Sort relies on systematic heap construction and re-heapification operations, data movements depend primarily on heap restructuring rather than initial ordering. Unlike Quick Sort, Heap Sort does not exhibit extreme movement escalation under adversarial inputs, reinforcing its structural robustness and input insensitivity.

#### 4.6.5 Discussion

The empirical findings demonstrate that Heap Sort exhibits strong structural stability and predictable performance. Execution time, comparison count, and data movement frequency remain largely consistent across dataset arrangements, indicating minimal sensitivity to input structure.

While Heap Sort does not always achieve the fastest runtime on random datasets, it avoids the severe degradation observed in Quick Sort under structured or adversarial inputs. Data movement counts remain within a narrow range for most dataset types, reflecting the algorithm's reliance on systematic heap construction and re-heapification rather than presortedness.

Compared to Merge Sort, Heap Sort performs in-place sorting without additional memory overhead, though it may incur slightly higher constant factors due to heap maintenance operations.

Overall, Heap Sort represents a robust and input-insensitive algorithm whose practical behaviour closely aligns with theoretical  $O(n \log n)$  complexity bounds.

### 5. Discussion

The experimental results demonstrate that input data arrangement plays a critical role in determining practical sorting performance. While theoretical time complexity accurately predicts asymptotic

growth trends, it does not fully capture performance variations caused by structured or adversarial inputs.

A clear distinction emerges between adaptive and non-adaptive algorithms. Insertion Sort exhibited strong adaptiveness, achieving near-linear performance on sorted and nearly sorted datasets while degrading to quadratic behaviour under reverse ordering. In contrast, Selection Sort and Bubble Sort showed minimal sensitivity to input arrangement in terms of comparison count, which remained fixed across all datasets.

Among advanced algorithms, Merge Sort and Heap Sort demonstrated structural stability, maintaining consistent  $O(n \log n)$  behaviour across all dataset types. Their runtime growth remained predictable and largely unaffected by input structure.

Quick Sort, however, displayed pronounced sensitivity to dataset arrangement. While it achieved optimal performance on random datasets, it degraded significantly under sorted and reverse-sorted inputs due to unbalanced partitioning. This underscores the importance of pivot selection strategy in practical implementations.

An important observation across all algorithms is that execution time variation correlates more strongly with data movement frequency than with comparison count. For algorithms such as Bubble Sort and Selection Sort, comparison counts remained largely fixed across dataset arrangements, yet runtime differences were evident due to variation in swap frequency. In adaptive algorithms such as Insertion Sort, both comparisons and data movements scaled with inversion density, reinforcing sensitivity to presortedness. Quick Sort demonstrated extreme movement escalation under unbalanced partitioning, while Merge Sort and Heap Sort maintained stable movement patterns. These findings indicate that element movement is a critical practical factor influencing runtime beyond asymptotic comparison complexity.

Furthermore, the results suggest that adaptive algorithms can outperform asymptotically superior algorithms when applied to structured data. For example, Insertion Sort frequently surpassed Merge Sort and Heap Sort on nearly sorted datasets despite its quadratic worst-case bound.

Overall, the study confirms that input arrangement influences not only constant factors but, in some cases, the effective growth behaviour observed in practice.

## 6. Conclusion and Future Work

This study presented a comprehensive experimental analysis of the impact of input data arrangement on the practical performance of sorting algorithms. By systematically evaluating both elementary and advanced sorting techniques across diverse dataset structures, the research sought to bridge the gap between theoretical time complexity analysis and real-world execution behaviour.

The results confirm that asymptotic complexity remains a reliable predictor of growth trends; however, it does not fully capture the influence of input structure on practical efficiency. Algorithms such as Bubble Sort and Selection Sort demonstrated minimal sensitivity to input arrangement in terms of comparison structure, maintaining consistent quadratic behaviour across datasets. In contrast, Insertion Sort exhibited strong adaptiveness, achieving near-linear performance on sorted and nearly sorted inputs while degrading under reverse ordering.

Among advanced algorithms, Merge Sort and Heap Sort maintained stable  $O(n \log n)$  performance across all dataset types, demonstrating structural independence from input arrangement. Quick Sort, however, displayed significant sensitivity to dataset structure, performing optimally on random inputs but degrading toward quadratic behaviour under sorted or adversarial conditions due to unbalanced partitioning.

Importantly, the study revealed that execution time variation is closely associated with data movement frequency in addition to comparison count. Algorithms exhibiting large movement escalation under disorder, such as Bubble Sort and Quick Sort, showed corresponding runtime increases, whereas

structurally stable algorithms maintained consistent movement patterns. This highlights that practical performance is influenced not only by asymptotic comparison complexity but also by element exchange behaviour.

These findings reinforce the central thesis of this study: input data arrangement plays a critical role in determining practical sorting performance. While theoretical complexity defines upper and lower bounds, real-world efficiency is strongly influenced by dataset characteristics such as presortedness, duplication, and structural patterning.

From a practical perspective, the results suggest that algorithm selection should consider input properties rather than relying solely on asymptotic complexity. Adaptive algorithms may outperform theoretically superior algorithms when applied to structured data, while structurally stable algorithms provide robustness under unpredictable input conditions.

## Future Work

This study can be extended in several directions:

1. Evaluating hybrid and adaptive industrial sorting algorithms such as TimSort to analyze how modern implementations exploit input structure.
2. Investigating the impact of different pivot selection strategies on Quick Sort performance.
3. Incorporating hardware-level metrics such as cache misses and branch prediction effects for deeper performance analysis.
4. Expanding dataset sizes further to analyze scalability at larger magnitudes.
5. Comparing comparison-based algorithms with non-comparison-based techniques such as Counting Sort and Radix Sort under controlled value distributions.

Such extensions would provide deeper insight into the interaction between algorithm design, data structure, and system architecture.

## 7. References

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.
- [2] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Reading, MA, USA: Addison-Wesley, 1998.
- [3] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani, *Algorithms*. New York, NY, USA: McGraw-Hill, 2008.
- [4] J. L. Bentley and M. D. McIlroy, “Engineering a Sort Function,” *Software: Practice and Experience*, vol. 23, no. 11, pp. 1249–1265, 1993.
- [5] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Boston, MA, USA: Addison-Wesley, 2011.
- [6] P. Sanders and S. Winkel, “Super Scalar Sample Sort,” in *European Symposium on Algorithms (ESA)*, 2004, pp. 784–796.
- [7] M. A. Weiss, *Data Structures and Algorithm Analysis in C++*, 4th ed. Pearson, 2014.

- [8] T. Peters,  
“Timsort: A Stable, Adaptive, Natural Mergesort,” 2002. [Online]. Available:  
<https://bugs.python.org/file4451/timsort.txt>
- [9] A. LaMarca and R. E. Ladner,  
“The Influence of Caches on the Performance of Sorting,” *Journal of Algorithms*, vol. 31, no. 1, pp. 66–104, 1999.
- [10] J. Hennessy and D. Patterson,  
*Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.