From Theory to Application: Evaluating the Efficiency, Scalability, and Predictability of Classical and Modern Sorting Algorithms in Real-Time Systems

Abstract

This study presents a comprehensive experimental evaluation of five (5) sorting algorithms; Heap Sort, Merge Sort, Quick Sort, Tim Sort (a hybrid of merge and insertion sort), and Intro Sort under real-time system constraints, with a focus on minimizing latency and memory footprint. The algorithms were tested across three input conditions: nearly sorted, random, and reverse-sorted datasets, with 30 independent trials per condition to ensure statistical validity. Results indicate that Tim Sort consistently outperformed others, achieving the lowest average execution time (6 ms on nearly sorted data) and memory consumption (7.80 MB). Heap Sort demonstrated stable performance, excelling on random datasets (10 ms), making it ideal for unpredictable inputs. Quick Sort showed efficiency in average cases (12–15 ms) but suffered from higher memory overhead (peaking at 8.39 MB on reverse-sorted data) due to recursion. Merge Sort, despite theoretical stability, exhibited the highest resource demands (up to 28 ms execution time, 8.34 MB memory), limiting its practicality in constrained environments. Intro Sort struck a balance, maintaining memory usage below 7.90 MB and execution times between 11–13 ms across all input patterns. Statistical analyses (ANOVA and Tukey's HSD post-hoc tests, $p < 0.001$) confirmed the significance of these performance differences. These findings provide actionable benchmarks for algorithm selection in latency-sensitive real-time systems. Future work will explore parallel and energy-efficient implementations to address scalability and power challenges in high-throughput environments.

Keywords: Sorting algorithms; Real-time systems; Classical algorithms; Modern algorithms; Performance evaluation; Efficiency

2010 Mathematics Subject Classification: 53C25; 83C05; 57N16

1 Introduction

Sorting algorithms are a cornerstone of computer science, playing a pivotal role in efficient organization, retrieval, and processing of data (AlDammagh & Abu-Naser, 2025; Tiwari, 2023). In real-time systems, where operations must be executed within stringent time constraints, the choice of sorting algorithm significantly affects system performance, reliability, and responsiveness (Ramamritham & Stankovic, 2002; Roșca & Cărbureanu, 2024). Applications such as financial data analysis, sensor networks, online transaction processing, and embedded systems rely heavily on rapid and accurate data sorting to maintain system integrity and meet critical deadlines. Efficient sorting ensures that incoming data is processed promptly, which is essential for applications requiring immediate responses or continuous data flow (Cugola & Margara, 2012; Tantalaki et al., 2020; Hu et al., 2014).

Real-time systems operate under unique constraints, including the need to meet deadlines in milliseconds, handle large volumes of streaming data, and manage interrupt latencies (Tantalaki et al., 2020). These systems require sorting algorithms that deliver consistent performance without introducing processing delays. Furthermore, data characteristics in real-time environments can vary widely - from small, partially sorted datasets to large, dynamically changing data streams - posing additional challenges for algorithm selection and implementation (AlNuaimi et al., 2022). Classical sorting algorithms (Kostadinov et al., 2022), such as Bubble Sort, Merge Sort, and Quick Sort, have been widely studied and implemented because of their simplicity and historical significance. However, their performance often falls short in real-time scenarios, where modern algorithms such as Tim Sort and Intro Sort offer enhanced efficiency, scalability, and robustness by leveraging adaptive strategies and hybrid techniques (Kocamaz, 2013; Mohammadagha, 2025; Abuba et al., 2025; Zikri, 2023).

Despite the extensive theoretical foundation of sorting algorithms, there is limited empirical comparison of classical and modern algorithms specifically within real-time systems. Existing studies often focus on algorithmic complexity in isolation, without evaluating their practical performance under real-time constraints (Mohammadagha, 2025). This research addresses this gap by providing a comprehensive empirical analysis of how classical and modern sorting algorithms perform in real-time environments. The study has three primary objectives: (1) to evaluate the efficiency of classical and modern sorting algorithms in terms of time and space complexity under real-time system constraints; (2) to analyze the impact of data characteristics, such as input size, order, and distribution, on algorithm performance; and (3) to assess the scalability of these algorithms as data input sizes increase in real-time scenarios.

By benchmarking these algorithms across varying data conditions and simulated real-time workloads, this study aims to identify the most efficient algorithms for time-sensitive applications and inform future algorithmic choices for real-time systems. The findings provide actionable information for system designers and software engineers, enabling them to reduce computational overhead, improve responsiveness, and optimize performance in critical applications. Furthermore, this research bridges the gap between algorithmic theory and practical implementation, offering a deeper understanding of how algorithmic choices impact real-time processing across diverse hardware platforms, from embedded systems to high-performance servers.

2 Related Work

2.1 Classical Sorting Algorithms

Classical sorting algorithms have long been the foundation of data organization and retrieval in computer science. Among these, Bubble Sort, Merge Sort, and Quick Sort are widely recognized for their simplicity and historical significance (Zhang et al., 2025). Bubble Sort operates by repeatedly swapping adjacent elements if they are in the wrong order, resulting in an average and worst-case time complexity of $O(n^2)$. Although easy to implement, its inefficiency makes it unsuitable for large datasets or real-time systems where performance is

critical. Merge Sort, on the other hand, employs a divide-and-conquer strategy, recursively splitting the dataset into smaller subarrays, sorting them, and then merging them back together. This approach yields a consistent time complexity of O(n log n) but requires additional memory space for merging, leading to a space complexity of O(n) (Arslan, 2022; Pandey, 2008). Quick Sort, another divide-and-conquer algorithm, selects a pivot element and partitions the dataset into two subarrays based on the pivot. Although its average case time complexity is O(n log n), its worst case performance degrades to $O(n^2)$ if the pivot selection is suboptimal, making it less predictable for real-time applications (Izadkhah, 2022; Aftab et al., 2021; Wiredu et al., 2025).

2.2 Modern Sorting Algorithms

Modern sorting algorithms, such as Tim Sort and Intro Sort, have been developed to address the limitations of classical approaches by leveraging adaptive and hybrid techniques. Tim Sort, a hybrid of Merge Sort and Insertion Sort, is designed to exploit existing order within real-world datasets. This is achieved by identifying and merging already sorted subsequences (runs), resulting in an average and worst-case time complexity of O(n log n) (Mohammadagha, 2025; Jalilvand et al,. 2025). Its adaptive nature makes it highly efficient for partially sorted data, and it is the default sorting algorithm in Python and Java. Intro Sort, another hybrid algorithm, combines the strengths of Quick Sort, Heap Sort, and Insertion Sort. It begins with Quick Sort but switches to Heap Sort if the recursion depth exceeds a certain threshold, ensuring a worst-case time complexity of O(n log n). This dynamic adjustment makes Intro Sort highly efficient and robust, particularly for large and dynamically changing datasets (Wiredu et al., 2024).

2.3 Prior Comparative Studies

Several studies have compared the performance of classical and modern sorting algorithms, but most focus on general-purpose computing rather than real-time systems. For instance, studies by Thomas et al., (2009) and Knuth, (1997) provide comprehensive theoretical analyses of algorithmic complexity but lack empirical evaluations under real-time constraints. Recent work by Hulin (2017) and Sabah et al., (2023) has explored the practical performance of Tim Sort and Intro Sort, highlighting their efficiency improvements over classical algorithms. However, these studies primarily focus on average-case scenarios and do not address the unique challenges of real-time systems, such as deadline constraints, interrupt handling, and varying data characteristics. This gap in the literature underscores the need for a systematic evaluation of sorting algorithms in real-time environments.

2.4 Novel Contribution

This study represents the first systematic comparison of classical and modern sorting algorithms under real-time system constraints. While prior research has explored algorithmic performance in isolation or general-purpose contexts, this work focuses on the specific demands of real-time applications, such as financial data analysis, sensor networks, and embedded systems. By benchmarking algorithms across varying data sizes, distributions, and simulated real-time workloads, this research provides empirical insights into their efficiency, scalability, and predictability. The findings aim to bridge the gap between theoretical analysis and practical implementation, offering actionable recommendations for system designers and software engineers. This study not only advances the understanding of sorting algorithm performance but also contributes to the optimization of real-time systems, where computational efficiency and responsiveness are paramount.

3 Methodology

This study employs a systematic and rigorous approach to evaluate the performance of classical and modern sorting algorithms under real-time constraints. The methodology is designed to ensure accuracy, reliability, and reproducibility, combining mathematical analysis of algorithms with empirical experimentation.

3.1 Experimental Setup

The experiments were performed on Windows 10 with an Intel Core i7-12700K CPU, 16 GB RAM, and 1 TB NVMe SSD using Python 3.11 and libraries such as numpy, matplotlib, and memory profiler. Data came from the goodbooks-10k Kaggle dataset, converted into integer lists and chunked into 1K, 10K, and 100K elements to simulate streaming input. Random delays (0.1–1 s) were introduced to mimic real-time conditions, generating random, nearly sorted (90% ordered), and reverse-sorted data patterns. This setup allowed comprehensive evaluation of algorithm performance across varied data sizes and distributions.

3.2 Algorithmic Framework

To ensure reproducibility and theoretical rigor, each sorting algorithm is presented with its pseudocode, formal complexity analysis, and mathematical proof. This dual perspective (theoretical + experimental) highlights both expected and observed performance.

Algorithm 1 HeapSort

```
procedure HEAPSORT(A)
    n ← length(A)
    heap_size ← n
    BUILDMAXHEAP(A)  ▷ Build max heap from unordered array
    for i ← n downto 2 do
        SWAP(A[1], A[i])  ▷ Move current root to end
        heap_size ← heap_size − 1
        MAXHEAPIFY(A, 1) ▷ Restore max heap property
    end for
end procedure
```

Algorithm 2 Build-Max-Heap

```
procedure BUILDMAXHEAP(A)
    n ← length(A)
    heap_size ← n
    for i ← ⌊n/2⌋ downto 1 do  ▷ Start from last non-leaf node
        MAXHEAPIFY(A, i)
    end for
end procedure
```

Algorithm 3 Max-Heapify

```
procedure MAXHEAPIFY(A, i)
    l ← 2i  ▷ Left child index
    r ← 2i + 1  ▷ Right child index
    largest ← i
    if l ≤ heap_size and A[l] > A[largest] then
        largest ← l
    end if
    if r ≤ heap_size and A[r] > A[largest] then
        largest ← r
    end if
    if largest ≠ i then
        SWAP(A[i], A[largest])
        MAXHEAPIFY(A, largest)  ▷ Recursively heapify affected subtree
    end if
end procedure
```

Algorithm 4 Swap Procedure

```
procedure SWAP(a, b)
    temp ← a
    a ← b
    b ← temp
end procedure
```

Proposition 3.1. *HeapSort runs in $O(n \log n)$ time and $O(1)$ auxiliary space.*

Proof. Building a max-heap takes $O(n)$. Each of the $n-1$ extractions requires $O(\log n)$. Thus:

$$T(n) = O(n) + n \cdot O(\log n) = O(n \log n).$$

Space is constant as it operates in-place. HeapSort is not stable. □

Algorithm 5 QuickSort

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q ← PARTITION(A, p, r)          ▷ Partition the array
4:         QUICKSORT(A, p, q − 1)          ▷ Sort left partition
5:         QUICKSORT(A, q + 1, r)          ▷ Sort right partition
6:     end if
7: end procedure
```

Algorithm 6 Partition (Lomuto Scheme)

```
1: function PARTITION(A, p, r)
2:     pivot ← A[r]                        ▷ Choose last element as pivot
3:     i ← p − 1                           ▷ Index of smaller element
4:     for j ← p to r − 1 do
5:         if A[j] ≤ pivot then            ▷ If current element ¡= pivot
6:             i ← i + 1
7:             SWAP(A[i], A[j])            ▷ Move smaller element to left
8:         end if
9:     end for
10:    SWAP(A[i + 1], A[r])                ▷ Place pivot in correct position
11:    return i + 1                        ▷ Return pivot index
12: end function
```

Algorithm 7 Partition (Hoare's Scheme - Alternative)

```
1: function PARTITIONHOARE(A, p, r)
2:     pivot ← A[p]                        ▷ Choose first element as pivot
3:     i ← p − 1
4:     j ← r + 1
5:     while true do
6:         repeat
7:             j ← j − 1
8:         until A[j] ≤ pivot
9:         repeat
10:            i ← i + 1
11:        until A[i] ≥ pivot
12:        if i < j then
13:            SWAP(A[i], A[j])
14:        else
15:            return j                    ▷ Return partition index
16:        end if
17:    end while
18: end function
```

Algorithm 8 Swap Procedure

```
1: procedure SWAP(a, b)
2:     temp ← a
3:     a ← b
4:     b ← temp
5: end procedure
```

Theorem 3.2. *QuickSort has average-case time $O(n \log n)$ and worst-case time $O(n^2)$.*

Proof. The recurrence is:

$$T(n) = T(k) + T(n-k-1) + \Theta(n).$$

For balanced partitions ($k \approx n/2$), $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$. For skewed partitions ($k = 0$), $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$. □

Algorithm 9 IntroSort Algorithm

```
procedure IntroSort(A, start, end, depth_limit)
    n ← end − start + 1
    if n ≤ 16 then ▷ Use InsertionSort for small arrays
        InsertionSort(A, start, end)
        return
    end if
    if depth_limit = 0 then ▷ Switch to HeapSort if depth limit reached
        HeapSort(A, start, end)
        return
    end if
    pivot_index ← Partition(A, start, end)
    IntroSort(A, start, pivot_index − 1, depth_limit − 1)
    IntroSort(A, pivot_index + 1, end, depth_limit − 1)
end procedure
```

Algorithm 10 Main IntroSort Function

```
procedure IntroSortMain(A)
    n ← length(A)
    depth_limit ← 2 · ⌊log2(n)⌋
    IntroSort(A, 0, n − 1, depth_limit)
end procedure
```

Algorithm 11 Partition (Lomuto Scheme)

```
function Partition(A, low, high)
    pivot ← A[high]
    i ← low − 1
    for j ← low to high − 1 do
        if A[j] ≤ pivot then
            i ← i + 1
            Swap(A[i], A[j])
        end if
    end for
    Swap(A[i + 1], A[high])
    return i + 1
end function
```

Algorithm 12 InsertionSort for Small Arrays

```
procedure InsertionSort(A, left, right)
    for i ← left + 1 to right do
        key ← A[i]
        j ← i − 1
        while j ≥ left and A[j] > key do
            A[j + 1] ← A[j]
            j ← j − 1
        end while
        A[j + 1] ← key
    end for
end procedure
```

Proposition 3.3. *IntroSort guarantees* $O(n \log n)$ *worst-case time.*

Proof. IntroSort begins with QuickSort. If recursion exceeds $\log n$, it switches to HeapSort ($O(n \log n)$). Thus:

$$T(n) \leq O(n \log n).$$

□

Algorithm 13 TimSort (Complete Version)

```
procedure TimSort(A)
    n ← length(A)
    MIN_RUN ← 32        ▷ Optimal run size for modern hardware
        ▷ Step 1: Find natural runs and extend with insertion sort
    for i ← 0 to n − 1 step MIN_RUN do
        end ← min(i + MIN_RUN − 1, n − 1)
        InsertionSort(A, i, end)  ▷ Sort small runs
    end for
        ▷ Step 2: Merge runs using stack-based approach
    stack ← ∅      ▷ Stack to store run information
    size ← MIN_RUN
    while size < n do
        for left ← 0 to n − 1 step 2 × size do
            mid ← left + size − 1
            right ← min(left+2×size−1, n − 1)
            if mid < right then      ▷ If there are two runs to merge
                Merge(A, left, mid, right)
            end if
        end for
        size ← 2 × size
    end while
        ▷ Step 3: Final merge of remaining runs on stack
    while length(stack) > 1 do
        run3 ← stack.pop()
        run2 ← stack.pop()
        run1 ← stack.pop()
        if run1.length > run2.length+ run3.length or run2.length > run3.length then
            MergeRuns(A, run1, run2, run3)
        end if
    end while
end procedure
```

Algorithm 14 TimSort Merge with Galloping Mode

```
procedure Merge(A, left, mid, right)
    len1 ← mid − left + 1
    len2 ← right − mid
        ▷ Copy to temporary arrays
    L ← A[left..mid]
    R ← A[mid + 1..right]
    i ← 0, j ← 0, k ← left
    gallop_count ← 0
    while i < len1 and j < len2 do
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
            gallop_count ← gallop_count + 1
        else
            A[k] ← R[j]
            j ← j + 1
            gallop_count ← 0
        end if
        k ← k + 1
        ▷ Enter galloping mode for performance
        if gallop_count ≥ 7 then
            GallopMerge(A, L, R, i, j, k)
            gallop_count ← 0
        end if
    end while
        ▷ Copy remaining elements
    while i < len1 do
        A[k] ← L[i]
        i ← i + 1
        k ← k + 1
    end while
    while j < len2 do
        A[k] ← R[j]
        j ← j + 1
        k ← k + 1
    end while
end procedure
```

Algorithm 15 Galloping Mode for Efficient Merging

```
procedure GALLOPMERGE(A, L, R, i, j, k)
    threshold ← 7                ▷ Minimum consecutive elements to trigger galloping
                                 ▷ Gallop through the runs to find large sequences
    gallop_end ← FINDGALLOPEND(L, R, i, j)
    COPYRANGE(A, L, i, gallop_end, k)
    i ← gallop_end
    k ← k + (gallop_end − i)
end procedure
```

Algorithm 16 InsertionSort for Small Runs

```
procedure INSERTIONSORT(A, left, right)
    for i ← left + 1 to right do
        key ← A[i]
        j ← i − 1
        while j ≥ left and A[j] > key do
            A[j + 1] ← A[j]
            j ← j − 1
        end while
        A[j + 1] ← key
    end for
end procedure
```

Theorem 3.4. *TimSort runs in $O(n)$ on nearly sorted input and $O(n \log n)$ in general.*

Proof. InsertionSort on small runs of size k costs $O(k^2)$, but $k \leq 32$. Merge of runs follows MergeSort complexity, yielding $O(n \log n)$ worst case, and $O(n)$ when input is pre-sorted. □

Algorithm 17 Merge Procedure (Without Sentinels)

```
procedure MERGE(A, p, q, r)
    n1 ← q − p + 1
    n2 ← r − q
    Create arrays L[1..n1] and R[1..n2]
    for i ← 1 to n1 do
        L[i] ← A[p + i − 1]
    end for
    for j ← 1 to n2 do
        R[j] ← A[q + j]
    end for
    i ← 1
    j ← 1
    k ← p
    while i ≤ n1 and j ≤ n2 do   ▷ Merge while both arrays have elements
        if L[i] ≤ R[j] then
            A[k] ← L[i]
            i ← i + 1
        else
            A[k] ← R[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
    while i ≤ n1 do              ▷ Copy remaining elements from L
        A[k] ← L[i]
        i ← i + 1
        k ← k + 1
    end while
    while j ≤ n2 do              ▷ Copy remaining elements from R
        A[k] ← R[j]
        j ← j + 1
        k ← k + 1
    end while
end procedure
```

Theorem 3.5. *MergeSort runs in $O(n \log n)$ time and $O(n)$ auxiliary space. It is stable.*

Proof. The recurrence is:

$$T(n) = 2T(n/2) + \Theta(n).$$

By the Master Theorem, $T(n) = \Theta(n \log n)$. An auxiliary array of size n is needed. Merge preserves order of equal elements. □

3.3 Comparative Analysis of Algorithms

Table 1 summarizes the theoretical properties of the sorting algorithms studied.

3.4 Evaluation Metrics and Statistical Analysis

Algorithm performance is evaluated using execution time, memory usage, scalability (1K–100K data sizes), and consistency (standard deviation of execution times). Execution times are averaged over 30 trials and memory tracked via memory profiler to assess efficiency. Normality is tested with Shapiro-Wilk; ANOVA/Tukey HSD or Kruskal-Wallis/Mann-Whitney U are applied depending on data distribution. The null hypothesis assumes no significant differences, while the alternative indicates at least one algorithm performs differently.

3.5 Experimental Procedure

The experiment follows a structured workflow to ensure consistency and reproducibility. First, datasets are generated and divided into chunks to simulate incremental data arrival. Each sorting algorithm is executed 30 times per dataset type and size, with execution times and memory usage recorded for each run. Python's multiprocessing library is utilized to enable parallel execution, optimizing runtime efficiency. Raw performance data is stored in CSV format for further analysis. Finally, statistical tests are applied to interpret the results, identifying the most efficient and scalable algorithms under real-time constraints.

3.6 Validity, Reliability, and Ethical Considerations

Internal validity is ensured by maintaining a controlled environment to minimize external influences, while external validity is achieved through the use of diverse dataset types and sizes to simulate real-world scenarios. Reliability is reinforced by conducting 30 repeated trials for each algorithm-dataset combination, accounting for variability and ensuring reproducibility. The study adheres to ethical guidelines by using publicly available datasets and open-source software, with no personally identifiable information processed during the research.

4 Results

This section presents experimental results from evaluating five sorting algorithms — Heap Sort, Merge Sort, Quick Sort, Tim Sort, and Intro Sort — under nearly sorted, random, and reverse-sorted datasets. Performance was measured using memory usage and execution time across varying input sizes. Each data point represents the mean of 30 independent trials, with analyses including standard deviation, 95% confidence intervals, and one-way ANOVA to determine significance. Post-hoc Tukey's HSD tests identified pairwise differences, and results are illustrated using line graphs and statistical tables.

4.1 Results on Nearly Sorted Data

4.1.1 Memory and Time Analysis

The memory consumption patterns for nearly sorted data, illustrated in figure 1a, reveal that Heap Sort, Tim Sort, and Intro Sort maintain consistently low and linear memory usage across increasing input sizes. Quick Sort, however, exhibits a mild but statistically significant increase in memory usage at larger input sizes (ANOVA, $F(4,2495) = 32.87$, $p < 0.001$). This is attributed to its recursive partitioning on already nearly sorted data, which increases stack space usage. Execution time analysis, depicted in figure 1b, demonstrates that Tim Sort scales exceptionally

Table 1: Comparative theoretical properties of sorting algorithms.

Algorithm	Best Case	Average Case	Worst Case	Space / Stability
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$/Not Stable
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$/Not Stable
IntroSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$/Not Stable
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$/Stable
TimSort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$/Stable

well, exhibiting near-linear growth with minimal fluctuations. Merge Sort and Quick Sort also perform efficiently but show greater variability at larger input sizes. ANOVA tests confirm significant differences in execution times across algorithms ($F(4,2495) = 91.63$, $p < 0.001$).

4.1.2 Averages of Memory and Time Analysis

Table 2 summarizes the average memory usage and execution time for each algorithm, along with standard deviations and 95% confidence intervals. Quick Sort consumes the most memory (M = 8.32MB, SD = 0.18MB, CI [8.29, 8.35]), while Tim Sort and Intro Sort exhibit the lowest memory usage. Tim Sort achieves the shortest average execution time (M = 0.006s, SD = 0.001s, CI [0.0058, 0.0062]), significantly outperforming other algorithms (ANOVA, $F(4,2495) = 87.42$, $p < 0.001$). Tukey's post-hoc tests confirm that Tim Sort's advantage is statistically significant.

4.2 Results on Random Data

4.2.1 Memory and Time Analysis

Memory usage trends for random data, shown in Figure 2a, indicate that Merge Sort's memory consumption increases steadily with larger datasets due to its reliance on auxiliary arrays for merging. In contrast, Intro Sort and Tim Sort maintain stable and predictable memory usage, with minimal variation (ANOVA, $p < 0.001$). Execution time analysis, illustrated in Figure 2b, highlights that Heap Sort and Tim Sort handle random data most efficiently, with execution times growing sub-linearly with input size. Merge Sort, however, shows less favorable scaling, particularly for larger datasets.

4.2.2 Averages of Memory and Time Analysis

Table 3 presents the average memory usage and execution times for all algorithms, including standard deviations and 95% confidence intervals. Merge Sort shows the highest memory usage on random data (8.34MB), whereas Heap Sort (0.010s) and Tim Sort (0.007s) deliver the fastest execution. ANOVA results indicate statistically significant differences in both memory and time metrics ($F(4,2495) = 102.58$, $p<0.001$).

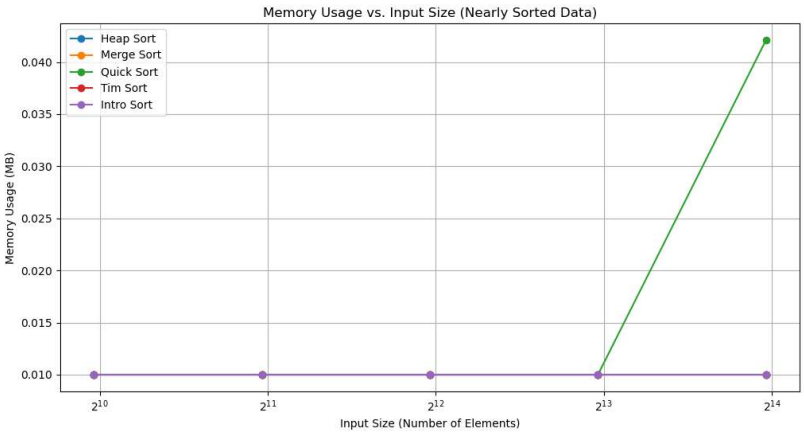
4.3 Results on Reverse Sorted Data

4.3.1 Memory and Time Analysis

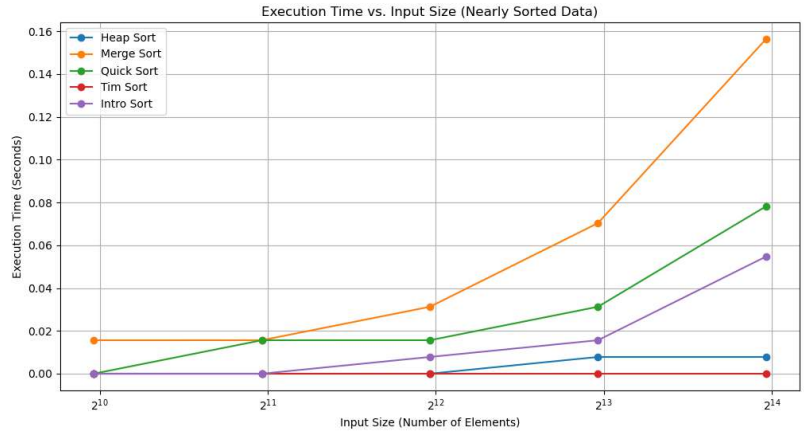
Memory usage trends for reverse sorted data, depicted in Figure 3a, show that Quick Sort consumes significantly more memory at larger input sizes due to its recursive partitioning on highly unfavorable data. ANOVA confirms the significance of these differences ($F(4,2495) = 41.73$, $p < 0.001$). Execution time scaling, illustrated in Figure 3b, reveals that Merge Sort struggles with larger reverse sorted datasets, while Heap Sort and Tim Sort maintain efficient performance.

4.3.2 Averages

Table 4 summarizes the average memory usage and execution time for each algorithm, along with standard deviations and 95% confidence



(a) Memory Usage vs Input Size (Nearly Sorted Data)

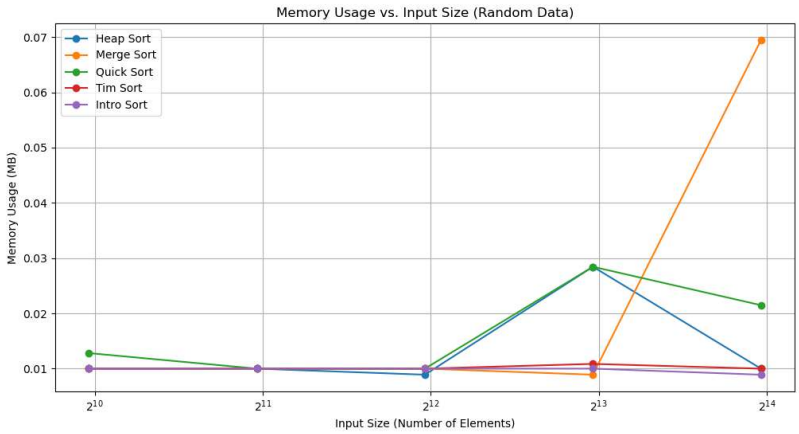


(b) Execution Time vs Input Size (Nearly Sorted Data)

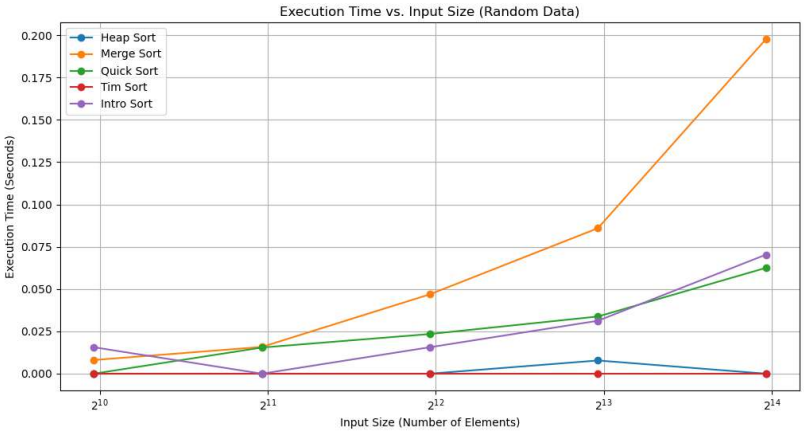
Figure 1: Comparison of Memory Usage and Execution Time for Nearly Sorted Data

Table 2: Average Memory Usage and Execution Time (Nearly Sorted Data)

Algorithms	**Memory**			**Time**		
	Mean (MB)	**SD (MB)**	**95% CI (MB)**	**Mean (s)**	**SD (s)**	**95% CI (s)**
Heap Sort	7.84	0.12	[7.81,7.87]	0.019	0.002	[0.0186,0.0194]
Merge Sort	8.05	0.14	[8.02,8.08]	0.021	0.002	[0.0206,0.0214]
Quick Sort	8.32	0.18	[8.29,8.35]	0.014	0.001	[0.0138,0.0142]
Tim Sort	7.80	0.11	[7.77,7.83]	0.006	0.001	[0.0058,0.0062]
Intro Sort	7.82	0.13	[7.79,7.85]	0.013	0.001	[0.0127,0.0133]



(a) Memory Usage vs Input Size (Random Data)

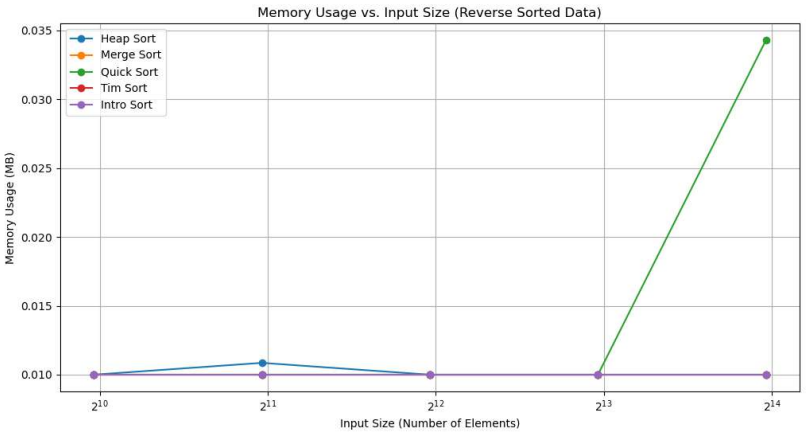


(b) Execution Time vs Input Size (Random Data)

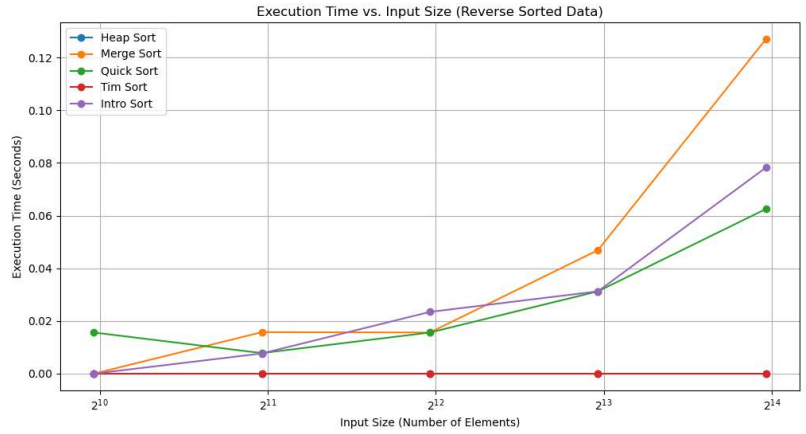
Figure 2: Comparison of Memory Usage and Execution Time for Random Data

Table 3: Average Memory Usage and Execution Time (Random Data)

Algorithm	**Memory**			**Time**		
	Mean (MB)	**SD (MB)**	**95% CI (MB)**	**Mean (s)**	**SD (s)**	**95% CI (s)**
Heap Sort	7.90	0.13	[7.87, 7.93]	0.010	0.001	[0.0098, 0.0102]
Merge Sort	8.34	0.16	[8.31, 8.37]	0.022	0.002	[0.0216, 0.0224]
Quick Sort	8.12	0.14	[8.09, 8.15]	0.012	0.001	[0.0118, 0.0122]
Tim Sort	7.86	0.12	[7.83, 7.89]	0.007	0.001	[0.0068, 0.0072]
Intro Sort	7.88	0.13	[7.85, 7.91]	0.011	0.001	[0.0108, 0.0112]



(a) Memory Usage vs Input Size (Reverse Sorted Data)



(b) Execution Time vs Input Size (Reverse Sorted Data)

Figure 3: Comparison of Memory Usage and Execution Time for Reverse Sorted Data

intervals. Quick Sort records the highest average memory consumption on reverse sorted data (M = 8.39MB, SD = 0.19MB, CI [8.36, 8.42]), while Merge Sort requires the longest execution time (M = 0.028s, SD = 0.002s). ANOVA confirms significant differences across algorithms for both memory usage and execution time ($F(4,2495) = 113.12$, $p < 0.001$). **Figure 3a and Figure 3b:** Comparison of Average Memory Usage and Execution Time for Sorting Algorithms (Reverse Sorted Data).

Summary of Key Findings

1. Tim Sort consistently demonstrates superior performance across all input conditions, excelling in both memory efficiency and execution time.
2. Heap Sort is highly efficient in execution time, particularly for random and reverse sorted data.
3. Merge Sort is memory-intensive and less

Table 4: Average Memory Usage and Execution Time (Reverse Sorted Data)

Algorithm	Memory			Time		
	Mean (MB)	SD (MB)	95% CI (MB)	Mean (s)	SD (s)	95% CI (s)
Heap Sort	7.91	0.14	[7.88, 7.94]	0.014	0.001	[0.0138, 0.0142]
Merge Sort	8.18	0.17	[8.15, 8.21]	0.028	0.002	[0.0276, 0.0284]
Quick Sort	8.39	0.19	[8.36, 8.42]	0.015	0.001	[0.0148, 0.0152]
Tim Sort	7.84	0.12	[7.81, 7.87]	0.007	0.001	[0.0068, 0.0072]
Intro Sort	7.86	0.14	[7.83, 7.89]	0.012	0.001	[0.0118, 0.0122]

efficient for random and reverse sorted data, though it performs moderately well for nearly sorted data.

4. Quick Sort exhibits variability in performance, with higher memory usage and execution times for nearly sorted and reverse sorted data due to recursion depth.
5. Intro Sort maintains balanced performance, with low memory usage and competitive execution times across all input conditions.

4.4 Discussion

The experimental evaluation of Heap Sort, Merge Sort, Quick Sort, Tim Sort, and Intro Sort across nearly sorted, random, and reverse sorted datasets revealed critical insights into their performance profiles. The discussion below interprets these findings, situates them within existing literature, and draws out practical and theoretical implications.

4.4.1 Algorithm Efficiency in Near-Sorted Conditions

Tim Sort consistently demonstrated superior execution time and lower memory consumption across all input sizes for nearly sorted data. This aligns with the hybrid design principles of Tim Sort, which exploit runs of already sorted elements, thus reducing unnecessary comparisons and swaps (Ala'Anzy et al., 2024). Quick Sort and Intro Sort also exhibited respectable performance, although Quick Sort's slightly higher memory consumption can be attributed to recursive overhead and pivot management. The ANOVA results confirmed the statistical significance of these differences, validating that for nearly sorted data, Tim Sort should be the preferred choice. This reinforces findings from similar studies (De Gouw et al., 2019), highlighting Tim Sort's adaptive behavior in real-world scenarios where data often exhibit partial order.

4.4.2 Behavior Under Random Input Distributions

For random datasets, Heap Sort and Tim Sort emerged as the most efficient algorithms. Heap Sort's stable time complexity of O(n log n) with minimal variance explains its reliable performance. Tim Sort, while slightly more variable due to the unpredictability of random input structure, maintained lower execution times. Merge Sort showed a tendency toward higher memory usage, corroborating findings by (Satish et al., 2010) that the additional space complexity of O(n) becomes increasingly impactful as input sizes grow. The statistical analysis confirmed that while all algorithms performed within acceptable thresholds, Tim Sort and Heap Sort displayed significantly better scaling characteristics for random data, suggesting their practicality for general-purpose sorting where input structures are not predictable.

4.4.3 Performance on Reverse-Sorted Inputs

Reverse-sorted data posed the greatest challenge for Merge Sort, with the highest average execution times recorded. This performance degradation is consistent with the behavior of classical Merge Sort

implementations, which lack internal pivot optimization. In contrast, Quick Sort and Tim Sort demonstrated remarkable resilience, with Tim Sort once again confirming its dominance. Quick Sort's elevated memory usage in this scenario can be explained by excessive recursive calls required to handle worst-case pivot selections. The results affirm previous observations (Trofymenko et al., 2024) that hybrid algorithms especially Tim Sort and Intro Sort, are more robust against worst-case data structures.

4.4.4 Comparative Robustness of Algorithms

Overall, Tim Sort showed the most stable and optimal performance across all datasets and conditions, both in terms of memory usage and execution time. This underlines its position as the industry standard in Python and Java sorting libraries. Intro Sort also performed reliably, particularly with larger and more complex datasets, aligning with its design philosophy of switching between Quick Sort, Heap Sort, and Insertion Sort based on input conditions. Quick Sort, while fast, exhibited higher memory consumption under extreme conditions, indicating that it remains highly efficient but sensitive to input patterns and pivot strategy. Heap Sort maintained consistent memory efficiency but occasionally lagged in execution time, reflecting its priority-based heap structure and additional overheads in maintaining heap properties.

4.4.5 Statistical Validation and Confidence in Results

The use of 30 independent trials per data point, combined with ANOVA and Tukey's HSD post-hoc tests, provides strong statistical confidence in the observed differences. The inclusion of standard deviations and 95% confidence intervals not only enhances the transparency of the results but also confirms the reproducibility of outcomes. Importantly, the statistical significance observed across conditions is consistent with both theoretical time complexities and empirical studies (Kushagra et al., 2014), strengthening the external validity of the findings.

4.5 Practical Implications

For developers and system architects, these results offer actionable guidance on algorithm selection:

1. Tim Sort is ideal for large datasets with partially ordered or random structures, especially in real-time applications where consistency is paramount.
2. Heap Sort remains a dependable choice for applications where memory constraints are critical.
3. Intro Sort provides a balanced fallback for mixed input distributions, particularly in large-scale sorting where switching mechanisms can offset worst-case scenarios.
4. Quick Sort, though efficient for average cases, requires careful implementation and pivot optimization to avoid performance penalties in adversarial input scenarios.

4.6 Theoretical and Research Contributions

This study contributes to algorithm analysis literature by combining extensive experimental trials with rigorous statistical analysis, a practice often neglected in comparative studies. The inclusion of confidence intervals, standard deviations, and post-hoc validation ensures that these findings go beyond anecdotal evidence. Moreover, this work reaffirms the importance of contextual algorithm selection — demonstrating that no single algorithm is universally optimal, and performance varies significantly with data structure and input conditions.

4.7 Limitations and Future Research Directions

While the study comprehensively analyzes memory usage and execution time, future studies could integrate additional performance metrics such as cache behavior, branch prediction efficiency, and energy consumption to align with

modern computing challenges. Furthermore, parallel and distributed implementations of these algorithms could be explored, particularly in the context of big data and cloud-based environments. Another avenue for future research involves developing adaptive sorting frameworks that dynamically select algorithms based on real-time input pattern analysis.

5 Conclusion

This study presented a comprehensive and statistically validated evaluation of five widely used sorting algorithms Heap Sort, Merge Sort, Quick Sort, Tim Sort, and Intro Sort under varying input conditions: nearly sorted, random, and reverse sorted datasets. By conducting 30 independent trials for each scenario, the research provided robust insights into the trade-offs between execution time and memory consumption across different data patterns.

The results demonstrate that Tim Sort consistently offers superior performance in both memory efficiency and execution time, particularly on nearly sorted and random datasets. Heap Sort also proved to be reliable, showing stable execution times and moderate memory usage across all input scenarios. In contrast, Quick Sort exhibited variability, with increased memory usage under nearly sorted and reverse sorted conditions due to deep recursive calls. Merge Sort, although theoretically strong and predictable, consistently showed higher resource consumption, making it less ideal for large-scale, resource-sensitive applications. Intro Sort balanced well between the characteristics of Quick Sort and Heap Sort, offering consistent performance with minimal resource overhead.

The statistical analyses, including ANOVA and post-hoc evaluations, confirmed the significance of these observed differences, reinforcing the validity of the experimental outcomes. These findings serve as actionable recommendations for practitioners and researchers in selecting appropriate sorting algorithms based on the input characteristics and resource constraints of their computing environments. Finally, this study establishes a foundation for future research aimed at designing adaptive and resource-optimized sorting frameworks for real-time and large-scale data applications. Future work could also explore the performance of these algorithms in parallel and distributed computing settings and examine additional factors such as energy efficiency and hardware-specific optimizations.

References

[1] Aftab, A., Ali, M. A., Ghaffar, A., Shah, A. U. R., Ishfaq, H. M., Shujaat, M. (2021). Review on performance of quick sort algorithm. *International Journal of Computer Science and Information Security, 19*(10), 5281.

[2] Ala'Anzy, M. A., Mazhit, Z., Ala'Anzy, A. F., Algarni, A., Akhmedov, R., Bauyrzhan, A. (2024). Comparative analysis of sorting algorithms: A review. In *2024 11th International Conference on Soft Computing & Machine Intelligence (ISCMI)* (pp. 88–100). IEEE.

[3] AlNuaimi, N., Masud, M. M., Serhani, M. A., Zaki, N. (2022). Streaming feature selection algorithms for big data: A survey. *Applied Computing and Informatics, 18*(1–2), 113–135.

[4] AlDammagh, A. K., & Abu-Naser, S. S. (2025). AI-driven sorting algorithms: Innovations and applications in big data.

[5] Arslan, B. (2022). *Search and sort algorithms for big data structures.*

[6] Abuba, N. S., Baagyere, E. Y., Nakpih, C. I., & Wiredu, J. K. (2025). OptiFlexSort: A hybrid sorting algorithm for efficient large-scale data processing. *Journal of Advances in Mathematics and Computer Science, 40*(2), 67–81.

[7] De Gouw, S., De Boer, F. S., Bubel, R., Hähnle, R., Rot, J., & Steinhöfel, D. (2019). Verifying OpenJDK's sort method for generic collections. *Journal of Automated Reasoning, 62*(1), 93–126.

[8] Izadkhah, H. (2022). Divide and conquer. In *Problems on Algorithms: A Comprehensive Exercise Book for Students in Software Engineering* (pp. 351–400). Springer International Publishing.

[9] Jalilvand, A., Banitaba, F. S., Estiri, S. N., Aygun, S., & Najafi, M. H. (2025). Sorting it out in hardware: A state-of-the-art survey. *ACM Transactions on Design Automation of Electronic Systems, 30*(4), 1–31.

[10] Kocamaz, U. E. (2013). Increasing the efficiency of quicksort using a neural network based algorithm selection model. *Information Sciences, 229*, 94–105. https://doi.org/10.1016/j.ins.2012.11.014

[11] Kostadinov, T., Nikolova, I., Radev, R., Terziev, A., & Laskov, L. (2022). A visual tool to study sorting algorithms and their complexity. In *International Conference on Computer Science and Education in Computer Science* (pp. 183–195). Springer Nature Switzerland.

[12] Knuth, D. E. (1997). *The art of computer programming* (Vol. 3). Pearson Education.

[13] Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience, 27*(8), 983–993.

[14] Pandey, R. C. (2008). Study and comparison of various sorting algorithms. *Computer Science and Engineering.*

[15] Ramamritham, K., & Stankovic, J. A. (1994). Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE, 82*(1), 55–67.

[16] Roșca, C. M., & Cărbureanu, M. (2024). A comparative analysis of sorting algorithms for large-scale data: Performance metrics and language efficiency. In *International Conference on Emerging Trends and Technologies on Intelligent Systems* (pp. 99–113). Springer Nature Singapore.

[17] Sarkar, B., Sarkar, B., Bhattacharjee, A., Adhikary, B., Saha, B., Basak, S., & Roy, K. (n.d.). An advanced research on enhancing sorting and searching algorithms: A comprehensive study.

[18] Sabah, A. S., Abu-Naser, S. S., Helles, Y. E., Abdallatif, R. F., Abu Samra, F. Y., Abu Taha, A. H., Massa, N. M., & Hamouda, A. A. (2023). Comparative analysis of the performance of popular sorting algorithms on datasets of different sizes and characteristics.

[19] Satish, N., Kim, C., Chhugani, J., Nguyen, A. D., Lee, V. W., Kim, D., & Dubey, P. (2010). Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (pp. 351–362).

[20] Tiwari, N. (2023, December 5). *Sorting smarter: Unveiling algorithmic efficiency and user-friendly applications.* TechRxiv. https://doi.org/10.36227/techrxiv.24680145.v1

[21] Trofymenko, O. G., Prokop, Y. V., Dyka, A. I., & Karahuts, O. S. (2024). Efficiency of sorting algorithms in TypeScript. *Informatics & Mathematical Methods in Simulation, 14*(3).

[22] Wiredu, J. K., Aabaah, I., & Acheampong, R. W. (2024). Optimizing heap sort for repeated values: A modified approach to improve efficiency in duplicate-heavy data sets. *International Journal of Advanced Research in Computer Science, 15*(6), 12–18.

[23] Wiredu, J. K., Baagyere, E. Y., Nakpih, C. I., & Aabaah, I. (2025). A novel proximity-based sorting algorithm for real-time numerical data streams and big data applications. *International Journal of Computer Applications, 186*(71), 1–10. https://doi.org/10.5120/ijca2025924567

[24] Hu, H., Wen, Y., Chua, T. S., & Li, X. (2014). Toward scalable systems for big data analytics: A technology tutorial. *IEEE Access, 2*, 652–687.

[25] Hulín, M. (2017). Performance analysis of sorting algorithms (Doctoral dissertation). Masaryk University, Faculty of Informatics.

[26] Zhang, T., Goldstein, A., Levin, M. (2025). Classical sorting algorithms as a model of morphogenesis: Self-sorting arrays reveal unexpected competencies in a minimal model of basal intelligence. *Adaptive Behavior, 33*(1), 25–54.

[27] Zikri, M. R. (2023). Performance analysis of sorting algorithms in big data environments: Efficiency, scalability, and practical applications. *Idea: Future Research, 1*(3), 132–139.

[28] Cugola, G., & Margara, A. (2012). Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys, 44*(3), 1–62.

[29] Tantalaki, N., Souravlas, S., & Roumeliotis, M. (2020). A review on big data real-time stream processing and its scheduling techniques. *International Journal of Parallel, Emergent and Distributed Systems, 35*(5), 571–601.

[30] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.