Asian Journal of Research in Computer Science

xx(x): ....., 20yy; Article no.AJRCOS.xxxxx

ISSN: 2456-9968

(Past name: British Journal of Mathematics & Computer Science, Past ISSN: 2231-0851)

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

Type of Article

Received: XX December 20XX
Accepted: XX December 20XX
Online Ready: XX December 20XX

Abstract

Efficient and predictable sorting is critical for real-time computing applications, including embedded systems, financial trading platforms, and sensor networks. In this paper, a strict comparative analysis of five popular sorting algorithms, namely Heap Sort, Merge Sort, Quick Sort, Tim Sort, and Intro Sort, in different input situations, i.e., almost sorted, random, and reverse-sorted data, is provided. Every algorithm was tested using 30 independent trials per condition so that the statistical strength is attained. Findings have shown that Tim Sort has a consistently better performance, the lowest execution time (6 ms) and the smallest memory footprint (7.80 MB), and thus its flexibility to data patterns of various types. Quick Sort is fast in cases that are average-case but has a higher memory overhead in cases where the input is reversed sorting. Hack Sort is stable with random data and Quick Sort is fast with average-case data. Merge Sort is predictable but resource-intensive, as the execution time goes up to 28 ms, and memory consumption is up to 8.34 MB. The performance of Intro Sort is balanced and has execution time of 11-13 ms and a memory size of less than 7.90 MB, regardless of the dataset. The significance of these differences can be proved by statistical procedures and correlations such as one-way ANOVA and post-hoc tests (Tukey) with the significance level of p less than 0.001. The combination of empirical assessment with strong statistical verification makes this study offer practical advice on the choice of algorithms in the latency sensitive computing environment. Further studies will elaborate this framework to parallel and distributed versions, which allows scalable and high-performance sorting in the present-day real-time systems.

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

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

1 Introduction

Computer science relies on sorting algorithms, which allow organizing, accessing, and working with data efficiently (AlDammagh and Abu-Naser, 2025; Tiwari, 2023). The sorting algorithm is a significant factor in determining the performance, reliability and responsiveness of a system in a real time system where the operations have to be carried out in a tight time constraint (Ramamritham and Stankovic, 2002; Rosca and Carbureanu, 2024). Financial data analysis, sensor networks, online transaction processing, and embedded systems are some of the applications that need data sorting within a short time frame and with high accuracy, and in this context, the integrity of the system and the pressing deadline are considered to be critical (Liu et al., 2014; Wolf, 2010). Optimized sorting means that data that comes in is immediately processed as needed by applications that demand immediate response or the constant movement of data (Cugola Margara, 2012; Tantalaki et al., 2020; Hu et al., 2014).

Real-time systems exist on special conditions, among them the ability to meet deadlines in milliseconds, work with extensive streams of data, and manage interrupt latencies (Tantalaki et al., 2020). Such environment sorting algorithms should offer consistency in performance with no processing delays (Satish et al., 2009; Estivill-Castro and Wood, 1992). Real-time systems may have data characteristics that are large, dynamically moving, or even small, partially sorted data; they may present extra difficulties in selecting and implementing algorithms (Chen et al., 2002; AlNuaimi et al., 2022). Bubble sort, Merge sort, and Quick sort are classical algorithms that are simple and historically important, so their study and application are widely used (Kostadinov et al., 2022). Nevertheless, they may not be sufficient in real-time processes where innovative algorithms such as Tim Sort and Intro Sort are more efficient, scalable, and resilient due to adaptive algorithms and hybrid methods (Kocamaz, 2013; Mohammadagha, 2025; Abuba et al., 2025; Zikri, 2023).

Although sorting algorithms have a long theoretical background, there has been a paucity of empirical comparison of both classical and the modern algorithms in the context of real-time systems (Mahmoud, 2000). Another problem is that most research studies consider the complexity of algorithms alone, and not in practice, with real-time constraints (Salama et al., 2002; Mohammadagha, 2025). This study fills such a gap by offering a more empirical study of the performance of classical and modern sorting algorithms in real-time settings. The objectives of the study are as follows: (1) to compare the performance of classical and modern sorting algorithms based on the execution time and memory consumption under the conditions of real-time requirements; (2) to examine how the characteristics of the data set including the size of the data set, the order and distribution influence the efficiency of the algorithm; and (3) to determine the scalability of these sorting algorithms to increasing data set sizes in real time conditions.

The study determines the most efficient algorithms in time-sensitive applications by benchmarking them in various conditions of data and in simulation under real-time workload. The results give practical guidelines to system designers and software engineers to minimize computational overhead, enhance responsiveness, and maximize performance in the critical programs. In addition, this study fills the gap between the theory of the algorithm and its application, providing a better insight to the way the selection of algorithm has an impact on real-time processing on a variety of hardware platforms, both embedded systems and full-fledged servers.

2 Related Work

2.1 Classical Sorting Algorithms

Traditional sorting methods have been the backbone of organizing and finding data in computer science for ages. Among these classics, Bubble Sort, Merge Sort, and Quick Sort standing out because they are straightforward and have played important roles throughout computing history (Zhang et al., 2025). Bubble Sort works by continuously switching neighboring items when they are out of place, which gives it both average and

worst-case running times of $O(n^2)$. Although straightforward to implement, its performance is prohibitively slow for large datasets or latency-sensitive applications. Merge Sort takes a different approach by breaking problems down into smaller pieces, splitting data into tiny chunks, sorting those pieces, and putting them back together again. This method consistently runs in O(n log n) time, but the downside is that it needs extra memory space for the merging process, bumping up its space requirements to O(n) (Arslan, 2022; Pandey, 2008). Quick Sort also uses the divide-and-conquer technique, but it picks a special element called a pivot and splits your data into two groups based on that pivot. While it typically runs in O(n log n) time, things can go south and slow down to $O(n^2)$ if pick lousy pivots, which makes it somewhat unreliable when needs consistent performance in time-sensitive situations (Izdakhah, 2022; Aftab et al., 2021; Wiredu et al., 2025).

2.2 Modern Sorting Algorithms

The current sorting algorithms, e.g. Tim Sort and Intro Sort which have been introduced to deal with the weaknesses of classical methods by overpowering adaptive and hybrid methods. Tim S ort is a combination of Merge Sort and Insertion Sort with an aim of taking advantage of natural order in real world data. This was accomplished by determining by finding and and combining already sorted subsequences (runs), therefore, having a worst and average-case time complexity of O(n log n) (Mohammadagha, 2025; Jalilvand et al,. 2025). It has a flexible nature, which gives it great efficiency in sorting small bits of data, and it is the default sorting algorithm in Python and in Java. The Intro Sort is another hybrid algorithm, combines the merits of Quick Sort, Heap Sort and Insert[?]ion Sort. It starts with Quick Sort but switches to Heap Sort when the recursion depth is greater than some fixed threshold giving a worst-case time complexity of O(n log n). This dynamic adaptation allows Intro Sort to be very powerful and resilient especially when using large and dynamically shifting datasets (Wiredu et al., 2024).

2.3 Prior Comparative Studies

Many studies have analyzed the performance of sorting algorithms, most of which have discussed their theoretical complexity or overall-purpose computing efficiency. Knuth (1997) has made the most authoritative reference to the complexity of algorithms in sorting, whereas Musser (1997) proposed Introsort, a hybrid algorithm that has the efficiency of Quicksort and the worst-case guarantees of Heapsort. TimSort was later created by Peters (2002) to exploit existing order in the input data and later formal analysis established that TimSort performed highly in a large variety of inputs (Auger et al., 2018). Despite the fact that these contributions made major contributions to the design of algorithms, they mostly focus on the theoretical properties or the overall performance and are not specifically focused on the needs of real-time systems.

Recent studies have been concerned with practical empirical analysis of sorting algorithms. Sabah (2023) contrasted the work of various algorithms, such as Quicksort, MergeSort, HeapSort, and TimSort, and proved TimSort to be effective in various situations. On the same note, Wibowo (2024) has discovered that, when used in Python, TimSort is frequently faster than classical algorithms on real-world data. Bigger comparative analyses have been undertaken, e.g., Pizarro-Vasquez, et al., (2021) have studied eleven algorithms with various input distributions, and Machkasova (2010) has pointed out the algorithmic trade-offs in an instructional case. Other researches have included aspects of resources in the performance assessment process including energy efficiency. The energy and time complexity of sorting algorithms in Java (2023) evaluated performance in terms of the execution time and power consumption, and Bakare et al., (2024) considered the performance under resource-constrained conditions. These studies, even though, are mostly related to average performance and are not directly related to predictability at real-time conditions.

Conversely, studies with a particular focus on sorting in real-time setting are few but useful. Mittermaair and Puschner (1997) discussed what sorting algorithms could be applied to hard real-time applications, and especially time-constrained scheduling and predictability. This analysis was further developed by Puschner

(1999) who carefully compared the execution-time performance of sorting algorithms in real-time, and Puschner and Burns (2000) who emphasised the significance of the worst-case execution time (WCET) analysis in systems design with regard to real-time. It has also investigated architectural influences on predictability, such as Marwedel et al., (2004) have shown different effects of hierarchies in memory, and caches behavior on execution times. Elmenreich et al., (2009) examined the situation of robustness and performance trade-offs, highlighting the difficulty in working between reliability and guarantees of timing.

Hardware-oriented research has contributed to this information by evaluating the performance in a specialized environment. Ben Jmaa, et al., (2021) compared the sorting algorithms in FPGA devices, focusing on the use of resources and speed gains. De Gouw, et al., (2015) examined the errors in the implementation of popular versions of TimSort, revealing that simple bugs can have an impact on practical reliability. Also, the measures of disorder and their effect on sorting performance were examined by Estivill-Castro and Wood (1991), and these findings are still applicable to the analysis of adaptability and worst-case behavior.

Collectively, these studies give a good background of the theoretical as well as practical consideration of sorting algorithms. There is however a gap: the current algorithms like Introsort and TimSort have been well defined in terms of efficiency but no systematic analysis exists of their behaviour in real time conditions involving deadlines, interrupt processing, jitter and WCET. This raises the issue of comparative analysis under real-time situations where efficiency is of less significance than predictability.

2.4 Novel Contribution

This paper is the first to give a systematic comparison of the classical and modern sorting algorithms of real time constraints in a system. Whereas earlier studies have studied the performance of algorithms either in isolation or in general-purpose applications, this study concentrates on the needs of real-time applications, including, but not limited to, financial data analysis, sensor networks, and embedded systems. Through the evaluation of algorithms using various data sets, distributions, and synthetic real-time loads, the paper offers empirical evidence on the efficiency, scaling and predictability of the algorithm. The results will close the gap between theory and practice with the view to providing practical suggestions to system designers and software engineers. Through this, this study will promote the knowledge of the performance of sorting algorithms and its applicability in optimizing real-time systems, where the performance of the computational system, in terms of performance and responsiveness, is paramount.

3 Methodology

This paper uses a scientific and methodological method to analyze the performance of classical and modern sorting algorithms when they face real-time constraints. The approach is constructed to have accuracy, reliability, and reproducibility with mathematical analysis of the algorithms and empirical experimentation of the algorithms.

3.1 Experimental Setup

The tests were done on three machines with various operating systems windows 10, windows 11, and MAC. Table 2 provides the hardware specifications of each machine. It was developed in Python 3.11 and with the help of Python libraries like NumPy, Matplotlib, and Memory Profiler, the simulations were conducted in order to guarantee the correct measurement and analysis of the performance.

The input data was derived from the Goodbooks-10k Kaggle dataset, converted into integer lists and divided into chunks of 1,000, 10,000, 100,000, 500,000, and 1,000,000 elements to simulate streaming conditions. To mimic real-time processing, random delays ranging from 0.1 to 1 second were introduced.

The experiments evaluated multiple data patterns, including random, nearly sorted (90% ordered), and reverse-sorted sequences, allowing a comprehensive assessment of algorithm performance in terms of execution time,

Table 1: Comparative Summary of Prior Studies on Sorting Algorithms

Author(s) & Year	Algorithms Studied	Metrics / Focus	Key Findings	Gaps / Limitations
Knuth (1997)	General sorting algorithms	Theoretical time/space complexity	Classic foundation on algorithmic complexity	No empirical evaluation; no real-time constraints considered
Musser (1997)	Introsort	Hybrid complexity analysis	Combines Quicksort speed with Heapsort's worst-case guarantee	No real-time performance assessment
Peters (2002)	TimSort	Algorithm design efficiency	Exploits natural runs in data	Not tested under deadline-sensitive environments
Estivill-Castro & Wood (1991)	General algorithms	Measures of disorder	Disorder level impacts sorting efficiency	No link to real-time or WCET scenarios
Auger et al. (2018)	TimSort	Worst-case analysis	Formalized worst-case complexity of TimSort	Still theoretical; not real-time focused
Machkasova (2010)	Multiple classical algorithms	Empirical runtime (teaching context)	Highlighted trade-offs between algorithms in practice	Limited scale; not real-time oriented
Pizarro-Vasquez et al. (2021)	Eleven algorithms	Execution times under varying inputs	Input distribution strongly influences performance	No deadline/jitter considerations
Sabah (2023)	Quicksort, MergeSort, HeapSort, TimSort	Runtime performance	TimSort effective across datasets	Lacks worst-case or deadline-sensitive analysis
Wibowo (2024)	TimSort vs. classical methods	Practical execution (Python)	TimSort often outperforms traditional methods	Limited to one programming environment
Bakare et al. (2024)	Multiple sorting algorithms	Efficiency in Nigerian data context	Performance varies by dataset scale	Focused on local data; not real-time or WCET
Energy & Time Complexity (2023)	Sorting algorithms in Java	Energy & time complexity	Introduced energy as a performance metric	Not real-time focused
De Gouw et al. (2015)	TimSort (Java, Python, Android)	Correctness and bug analysis	Found implementation flaws in real-world systems	Not about timing guarantees
Ben Jmaa et al. (2021)	Multiple algorithms on FPGA	Hardware acceleration, resource use	FPGA-based implementations enhanced performance	No systematic study of deadlines/jitter
Mittermair & Puschner (1997)	Multiple algorithms	Suitability for hard real-time	Identified algorithms appropriate for real-time	Limited set of algorithms studied
Puschner (1999)	Multiple algorithms	Real-time performance	Analyzed sorting algorithms' timing in real-time	Did not include modern hybrids (TimSort, Introsort)
Puschner & Burns (2000)	General scheduling/sorting approaches	WCET analysis	Emphasized worst-case predictability	No empirical evaluation on modern systems
Marwedel et al. (2004)	General algorithms on embedded systems	Memory/caching effects	Hardware impacts algorithm predictability	Focused on memory; not algorithm-specific
Elmenreich et al. (2009)	Sorting & tournament algorithms	Robustness vs. performance	Explored trade-offs between reliability and efficiency	Did not explicitly address deadlines

memory usage, throughput, and concurrency across diverse machines, operating systems, and input distributions.

3.2 Algorithmic Framework

Each sorting algorithm is accompanied by pseudocode, formal complexity analysis, and

Table 2: Hardware and Software Specifications for Simulation Environments

Specification	PC1 (Windows 10)	PC2 (Windows 11)	PC3 (MacBook Air)
Device Name	HP Envy x360 Convertible 15-bq1xx	HP Envy x360 Convertible 15-ed0xxx	Apple MacBook Air
Processor	AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx, 2.00 GHz	Intel Core i7-10510U @ 1.80GHz (2.30GHz)	Apple M1
RAM	16 GB	16 GB (15.7 GB usable)	8 GB
Storage	1 TB NVMe SSD	Integrated SSD	512 GB SSD
Operating System	Windows 10	Windows 11 Home SL (24H2, Build 26100.6584)	macOS Tahoe version 26.0
Software	Python 3.11, NumPy, Matplotlib, Memory Profiler	Python 3.11, NumPy, Matplotlib, Memory Profiler	Python (version TBD)

mathematical justification, providing a combined theoretical and experimental perspective on 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

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A, p, r)        ▷ Partition the array
        QUICKSORT(A, p, q − 1)        ▷ Sort left partition
        QUICKSORT(A, q + 1, r)        ▷ Sort right partition
    end if
end procedure
```

Algorithm 6 Partition (Lomuto Scheme)

```
function PARTITION(A, p, r)
    pivot ← A[r]                      ▷ Choose last element as pivot
    i ← p − 1                         ▷ Index of smaller element
    for j ← p to r − 1 do
        if A[j] ≤ pivot then          ▷ If current element ¡= pivot
            i ← i + 1
            SWAP(A[i], A[j])          ▷ Move smaller element to left
        end if
    end for
    SWAP(A[i + 1], A[r])              ▷ Place pivot in correct position
    return i + 1                      ▷ Return pivot index
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

```
1: procedure INTROSORT(A, start, end, depth_limit)
2:     n ← end − start + 1
3:     if n ≤ 16 then              ▷ Use InsertionSort for small arrays
4:         INSERTIONSORT(A, start, end)
5:         return
6:     end if
7:     if depth_limit = 0 then     ▷ Switch to HeapSort if depth limit reached
8:         HEAPSORT(A, start, end)
9:         return
10:    end if
11:    pivot_index ← PARTITION(A, start, end)
12:    INTROSORT(A, start, pivot_index − 1, depth_limit − 1)
13:    INTROSORT(A, pivot_index + 1, end, depth_limit − 1)
14: end procedure
```

Algorithm 10 Main IntroSort Function

```
1: procedure INTROSORTMAIN(A)
2:     n ← length(A)
3:     depth_limit ← 2 · ⌊log2(n)⌋
4:     INTROSORT(A, 0, n − 1, depth_limit)
5: end procedure
```

Algorithm 11 Partition (Lomuto Scheme)

```
1: function PARTITION(A, low, high)
2:     pivot ← A[high]
3:     i ← low − 1
4:     for j ← low to high − 1 do
5:         if A[j] ≤ pivot then
6:             i ← i + 1
7:             SWAP(A[i], A[j])
8:         end if
9:     end for
10:    SWAP(A[i + 1], A[high])
11:    return i + 1
12: end function
```

Algorithm 12 InsertionSort for Small Arrays

```
1: procedure INSERTIONSORT(A, left, right)
2:     for i ← left + 1 to right do
3:         key ← A[i]
4:         j ← i − 1
5:         while j ≥ left and A[j] > key do
6:             A[j + 1] ← A[j]
7:             j ← j − 1
8:         end while
9:         A[j + 1] ← key
10:    end for
11: 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)

```
1: procedure TIMSORT(A)
2:     n ← length(A)
3:     MIN_RUN ← 32        ▷ Optimal run size for modern hardware
           ▷ Step 1: Find natural runs and extend with insertion sort
4:     for i ← 0 to n − 1 step MIN_RUN do
5:         end ← min(i + MIN_RUN − 1, n − 1)
6:         INSERTIONSORT(A, i, end)  ▷ Sort small runs
7:     end for
               ▷ Step 2: Merge runs using stack-based approach
8:     stack ← ∅        ▷ Stack to store run information
9:     size ← MIN_RUN
10:    while size < n do
11:        for left ← 0 to n − 1 step 2 × size do
12:            mid ← left + size − 1
13:            right ← min(left+2×size−1, n − 1)
14:            if mid < right then        ▷ If there are two runs to merge
15:                MERGE(A, left, mid, right)
16:            end if
17:        end for
18:        size ← 2 × size
19:    end while
               ▷ Step 3: Final merge of remaining runs on stack
20:    while length(stack) > 1 do
21:        run3 ← stack.pop()
22:        run2 ← stack.pop()
23:        run1 ← stack.pop()
24:        if run1.length > run2.length+ run3.length or run2.length > run3.length then
25:            MERGERUNS(A, run1, run2, run3)
26:        end if
27:    end while
28: end procedure
```

Algorithm 14 TimSort Merge with Galloping Mode

```
1: procedure MERGE(A, left, mid, right)
2:     len1 ← mid − left + 1
3:     len2 ← right − mid
           ▷ Copy to temporary arrays
4:     L ← A[left..mid]
5:     R ← A[mid + 1..right]
6:     i ← 0, j ← 0, k ← left
7:     gallop_count ← 0
8:     while i < len1 and j < len2 do
9:         if L[i] ≤ R[j] then
10:            A[k] ← L[i]
11:            i ← i + 1
12:            gallop_count ← gallop_count + 1
13:        else
14:            A[k] ← R[j]
15:            j ← j + 1
16:            gallop_count ← 0
17:        end if
18:        k ← k + 1
               ▷ Enter galloping mode for performance
19:        if gallop_count ≥ 7 then
20:            GALLOPMERGE(A, L, R, i, j, k)
21:            gallop_count ← 0
22:        end if
23:    end while
           ▷ Copy remaining elements
24:    while i < len1 do
25:        A[k] ← L[i]
26:        i ← i + 1
27:        k ← k + 1
28:    end while
29:    while j < len2 do
30:        A[k] ← R[j]
31:        j ← j + 1
32:        k ← k + 1
33:    end while
34: end procedure
```

Algorithm 15 Galloping Mode for Efficient Merging

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

Algorithm 16 InsertionSort for Small Runs

```
1: procedure INSERTIONSORT(A, left, right)
2:     for i ← left + 1 to right do
3:         key ← A[i]
4:         j ← i − 1
5:         while j ≥ left and A[j] > key do
6:             A[j + 1] ← A[j]
7:             j ← j − 1
8:         end while
9:         A[j + 1] ← key
10:    end for
11: 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)
  $n_1 \leftarrow q - p + 1$
  $n_2 \leftarrow r - q$
  Create arrays $L[1..n_1]$ and $R[1..n_2]$
  for $i \leftarrow 1$ **to** n_1 **do**
    $L[i] \leftarrow A[p + i - 1]$
  end for
  for $j \leftarrow 1$ **to** n_2 **do**
    $R[j] \leftarrow A[q + j]$
  end for
  $i \leftarrow 1$
  $j \leftarrow 1$
  $k \leftarrow p$
  while $i \leq n_1$ **and** $j \leq n_2$ **do** ▷ Merge while both arrays have elements
    if $L[i] \leq R[j]$ **then**
      $A[k] \leftarrow L[i]$
      $i \leftarrow i + 1$
    else
      $A[k] \leftarrow R[j]$
      $j \leftarrow j + 1$
    end if
    $k \leftarrow k + 1$
  end while
  while $i \leq n_1$ **do** ▷ Copy remaining elements from L
    $A[k] \leftarrow L[i]$
    $i \leftarrow i + 1$
    $k \leftarrow k + 1$
  end while
  while $j \leq n_2$ **do** ▷ Copy remaining elements from R
    $A[k] \leftarrow R[j]$
    $j \leftarrow j + 1$
    $k \leftarrow k + 1$
  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 3 summarizes the theoretical properties of the sorting algorithms studied.

3.4 Evaluation Metrics and Statistical Analysis

The algorithm performance was measured according to the execution time, memory consumption, scalability (with data size 1K to 100K) and consistency (standard deviation of the execution time). Averaging was done on 30 separate executions of the program and efficiency was measured by monitoring memory use with Memory Profiler. The test of normality of data was done by Shapiro-Wilk test. ANOVA with Tukey HSD post-hoc test or non-parametric tests (Kruskal-Wallis and Mann-Whitney U) based on the distribution were used. The null hypothesis was based on the assumption that there were no significant differences between the algorithms and the alternative hypothesis was that, some algorithm would differ.

3.5 Experimental Procedure

The experiments were made in such a way that they were going to be consistent and reliable. The list was converted to numerical lists, and the Goodbooks-10k dataset was utilized. These lists were separated into groups of 1,000, 10,000, 100,000, 500,000, and 1,000,000 items to be able to model the gradual arrival of data.

All the sorting algorithms were repeated 30 times with each size of data and the execution time and memory consumed of each run was noted. Python multiprocessing library was used to run the tasks simultaneously in order to

Table 3: 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

enhance the efficiency. Any experimental data were saved in CSV files so as to be later analyzed.

The study then used simple statistics to compare the algorithms and see which ones were faster and used less memory, helping us find the best sorting methods for handling data in real-time.

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

The purpose of these experiments was to evaluate the performance of five sorting algorithms across different computing environments. Tests were conducted on three PCs: PC1 running Windows 10, PC2 running Windows 11, and PC3 using a MacBook Air. Datasets ranged in size from 10,000 to 1,000,000 elements, with input arrays arranged in random, nearly sorted, fully sorted, and reverse-sorted orders. Key performance metrics measured included execution time, memory usage, and throughput/concurrency, providing a comprehensive view of each algorithm's efficiency under varying conditions.

4.1 Execution Time Results

The execution time results reveal clear trends in algorithm performance across different machines and dataset sizes. Overall, Tim Sort consistently delivered the fastest execution times across all PCs and input sizes, followed by Quick Sort. Heap Sort and Intro Sort were slower, with Intro Sort showing the largest increases in execution time as dataset size grew. Merge Sort exhibited moderate performance, generally faster than Heap Sort and Intro Sort but slower than Quick Sort and Tim Sort.

Breaking the results down by input size:

- **Small datasets (10,000 elements):** Tim Sort and Quick Sort were significantly faster than other algorithms. PC3 (MacBook Air) performed best, completing the tasks roughly twice as fast as PC1 (Windows 10). Intro Sort showed the slowest execution on all PCs.
- **Medium datasets (50,000–100,000 elements):** The performance gap widened. On 50,000-element datasets, Tim Sort completed execution in milliseconds on all PCs, whereas Intro Sort and Heap Sort required several seconds. PC3 consistently outperformed both PC1 and PC2, completing tasks roughly 50% faster than PC1. Quick Sort remained efficient across PCs, but Intro Sort's time more than doubled on PC2 for larger datasets.
- **Large datasets (500,000–1,000,000 elements):** Differences became more

pronounced. On 1,000,000-element datasets, PC3 completed Tim Sort in under 0.07 seconds, while PC1 and PC2 required 0.23 and 0.18 seconds, respectively. Intro Sort execution time on PC2 (Windows 11) spiked for 500,000-element datasets, taking 113.5 seconds, compared to 61.8 seconds on PC1 and 33.3 seconds on PC3. Quick Sort remained stable, with PC3 consistently showing the shortest execution times.

Input pattern effects: While the dataset sizes dominate performance trends, previous analysis on input patterns suggests that nearly sorted and random arrays favored Quick Sort and Tim Sort, whereas reverse-sorted inputs increased Intro Sort execution times, especially on Windows 10 (PC1).

Comparative insights across PCs:

- PC3 (MacBook Air) outperformed PC1 (Windows 10) and PC2 (Windows 11) across nearly all algorithms and dataset sizes, highlighting the advantage of higher CPU efficiency and memory handling.
- PC2 (Windows 11) occasionally lagged behind PC1 on smaller datasets but showed extreme delays for specific algorithms like Intro Sort on large datasets, indicating possible system overheads.
- PC1 (Windows 10) consistently performed better than PC2 for mid-range datasets but was slower than PC3 in all scenarios.

Summary statement Tim Sort and Quick Sort were the fastest algorithms across all PCs and dataset sizes. Heap Sort and Intro Sort showed significantly higher execution times, especially for larger datasets and reverse-sorted arrays. PC3 (MacBook Air) consistently outperformed PC1 (Windows 10) and PC2 (Windows 11), demonstrating the influence of hardware on sorting efficiency.

Table 4 provides a full summary of execution times for all algorithms, PCs, and dataset sizes, serving as the primary reference for these observations.

4.2 Memory Usage Results

The memory usage results show distinct trends across algorithms and PCs. Overall, Merge Sort and Quick Sort consistently consumed the most memory, reflecting their data-intensive nature, while Tim Sort used noticeably less memory across all dataset sizes. Heap Sort and Intro Sort had moderate memory usage, with Intro Sort gradually increasing as dataset size grew.

By dataset size:

- **Small datasets (10,000 elements):** Tim Sort was the most memory-efficient, using less than 0.2 MB on all PCs. Quick Sort and Merge Sort required the most memory, with PC1 and PC2 using roughly 0.46 MB and 0.39 MB respectively, while PC3 required slightly less (0.36 MB for Quick Sort, 0.25 MB for Merge Sort).
- **Medium datasets (50,000–100,000 elements):** Memory usage scaled with dataset size. On 50,000-element datasets, Quick Sort and Merge Sort required around 1.7–1.9 MB on PCs 1 and 2, while Tim Sort remained below 0.86 MB. PC3 consistently used slightly less memory than PC1 and PC2 across all algorithms.
- **Large datasets (500,000–1,000,000 elements):** Differences became more pronounced. On 1,000,000-element datasets, Quick Sort and Merge Sort consumed up to 35 MB and 25 MB on PC1 and PC2, whereas Tim Sort remained around 17 MB. PC3 again showed slightly lower memory usage, with Quick Sort at 33.92 MB and Merge Sort at 25.17 MB, likely due to macOS memory management efficiencies.

Algorithm-specific trends:

- Merge Sort consistently had the highest memory requirements across all PCs and datasets.
- Quick Sort required slightly less memory than Merge Sort but was still among the top consumers.
- Tim Sort remained memory-efficient across all scenarios.

Table 4: Execution Time of Sorting Algorithms Across Machines (seconds)

Input Size	PC	Heap Sort	Intro Sort	Merge Sort	Quick Sort	Tim Sort
10,000	PC1	0.433245	0.799286	0.248046	0.136172	0.000655
	PC2	0.429493	0.802452	0.209718	0.140483	0.001009
	PC3	0.212390	0.412117	0.106980	0.066909	0.000375
50,000	PC1	2.649224	4.872181	1.547957	0.741960	0.004320
	PC2	2.844906	5.151721	1.334243	0.778375	0.006287
	PC3	1.397362	2.599995	0.673213	0.363356	0.002118
100,000	PC1	5.855998	10.330313	3.354665	1.534036	0.014936
	PC2	6.185129	12.165352	2.966466	1.595539	0.012584
	PC3	3.085958	5.548786	1.491611	0.735428	0.004515
500,000	PC1	36.575496	61.849843	18.994811	7.636401	0.101594
	PC2	41.032797	113.547622	90.445904	14.085202	0.084562
	PC3	18.808492	33.308916	9.009834	3.622948	0.028116
1,000,000	PC1	95.021049	264.566084	40.536024	15.462363	0.228864
	PC2	134.361096	121.413311	55.615349	15.548195	0.181837
	PC3	40.867481	72.521935	19.583417	7.387914	0.061371

- Heap Sort and Intro Sort had moderate memory usage, with Heap Sort occasionally slightly higher on mid-range datasets.

Comparative insights across PCs:

- PC3 (MacBook Air) generally had lower memory usage than PC1 (Windows 10) and PC2 (Windows 11) for all algorithms, likely due to better CPU cache management and system-level optimization.
- PC2 occasionally showed slightly lower memory for some algorithms (e.g., Intro Sort on medium datasets), but Quick Sort and Merge Sort were slightly higher than PC3.

Summary statement: Merge Sort and Quick Sort used the most memory across all PCs, whereas Tim Sort consistently used slightly less memory. PC3 showed slightly lower memory consumption across most algorithms, likely due to better CPU cache handling and system optimization.

Table 5 provides a full summary of memory usage across all PCs and dataset sizes.

4.3 Input Pattern Effects

As shown in Table 6, input data ordering had a significant impact on execution time, whereas its effect on memory usage across the sorting algorithms and PCs was comparatively smaller.

- **Random arrays:** Quick Sort and Tim Sort were fastest, while Merge Sort was stable but memory-intensive. Heap Sort was slower but predictable.
- **Nearly sorted arrays:** Tim Sort excelled due to its hybrid design, showing minimal execution time; Quick Sort improved slightly, while Heap and Merge Sort were largely unaffected.
- **Fully sorted arrays:** Tim Sort remained efficient, Quick Sort slowed slightly, and Intro Sort maintained moderate performance.

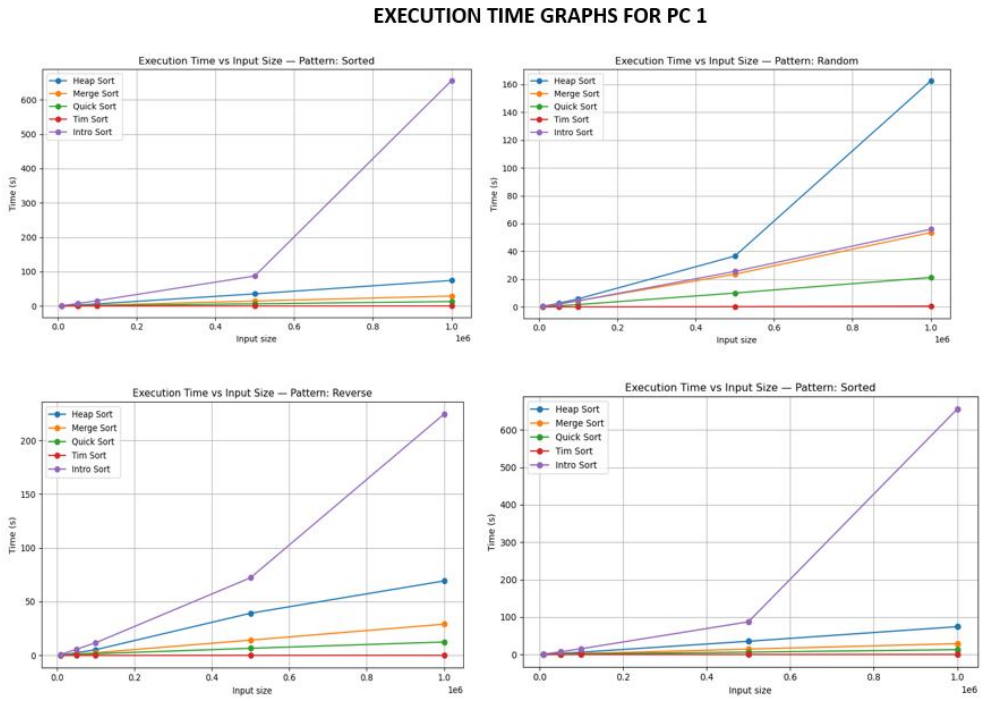

Figure 1: Execution time Graphs for PC3 (HP Envy x360 - Windows 10)

- **Reverse-sorted arrays:** Intro Sort was most sensitive, with execution time sharply increasing on PC1. Tim Sort handled these inputs well, slightly slower than nearly sorted arrays, while Heap and Merge Sort remained robust.

Memory usage: Merge Sort and Quick Sort consistently consumed the most memory; Tim Sort remained stable and efficient across all patterns.

Summary: Tim Sort benefited most from nearly sorted inputs, achieving minimal execution times. Intro Sort was highly sensitive to reverse-sorted arrays, while other algorithms showed moderate variation depending on input order and hardware.

4.4 Throughput and Concurrency Analysis

To evaluate higher-level system performance, we analyzed throughput and concurrency, which measure how many operations each PC can handle per second under different sorting algorithms and input sizes.

4.4.1 Throughput Trends

Across all PCs, Tim Sort and Quick Sort consistently achieved the highest throughput, particularly for small to medium input sizes (10,000–100,000 elements), reflecting their optimized execution strategies as shown in Table 7.

- Heap Sort and Merge Sort delivered stable but comparatively lower throughput across input sizes.
- Intro Sort showed more variation, with throughput decreasing significantly for larger datasets (500,000 elements) and reverse-sorted inputs.
- The more powerful PCs (PC3 ¿ PC2 ¿ PC1) naturally handled larger numbers of

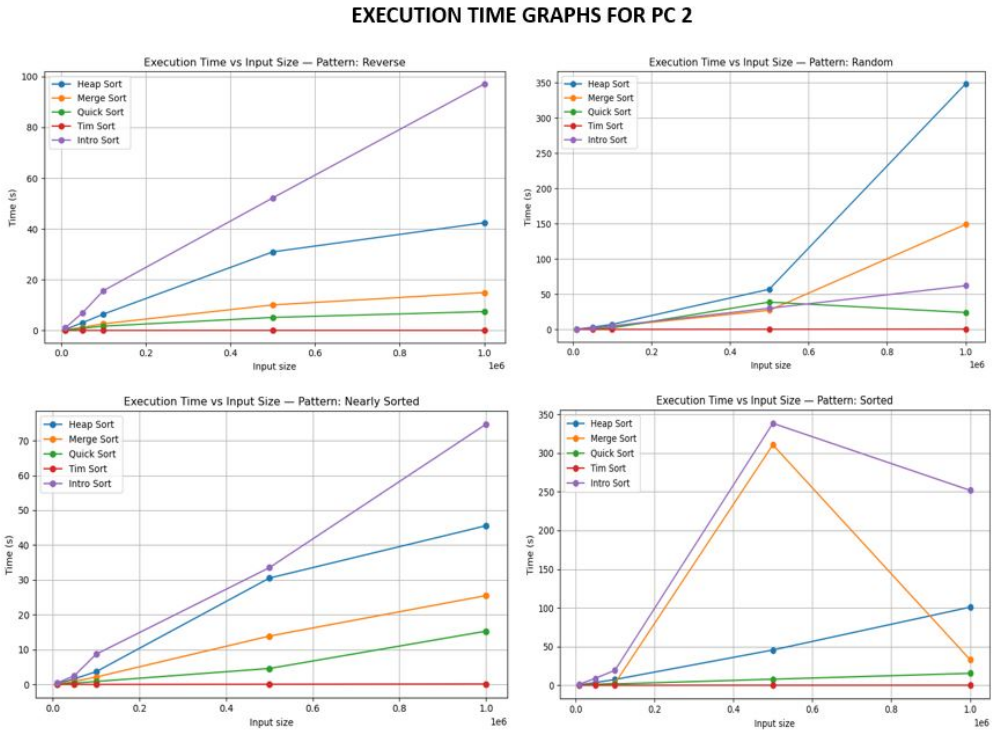

Figure 2: Execution time Graphs for PC2 (HP Envy x360 - Windows 11))

operations, with PC3 often doubling or tripling throughput relative to PC1.

4.4.2 Concurrency Trends

Concurrency mirrored throughput trends but magnified differences between PCs and algorithms as shown in Table 8.

- **Tim Sort** achieved extremely high concurrency on small inputs, particularly on PC3 (456 million operations/sec for 10,000 elements), demonstrating its hybrid design's efficiency in handling multiple simultaneous operations.
- **Quick Sort and Merge Sort** maintained high concurrency but scaled less dramatically with input size.
- Worst-case or large inputs reduced concurrency most noticeably for **Intro Sort**, while **Heap Sort and Merge Sort** remained relatively stable.

4.4.3 Summary

- **Tim Sort** dominates both throughput and concurrency, especially on smaller datasets, benefiting from its hybrid design that minimizes unnecessary operations.
- **Quick Sort** is consistently efficient but slightly less scalable than Tim Sort on concurrency.
- **Intro Sort** is sensitive to large datasets and worst-case inputs, showing reduced throughput and concurrency.
- **Heap Sort and Merge Sort** exhibit predictable and steady performance across PCs and input sizes.
- **PC hardware** plays a key role, with higher-spec machines achieving significantly greater throughput and concurrency, confirming that both algorithm design and system capabilities affect overall sorting performance.

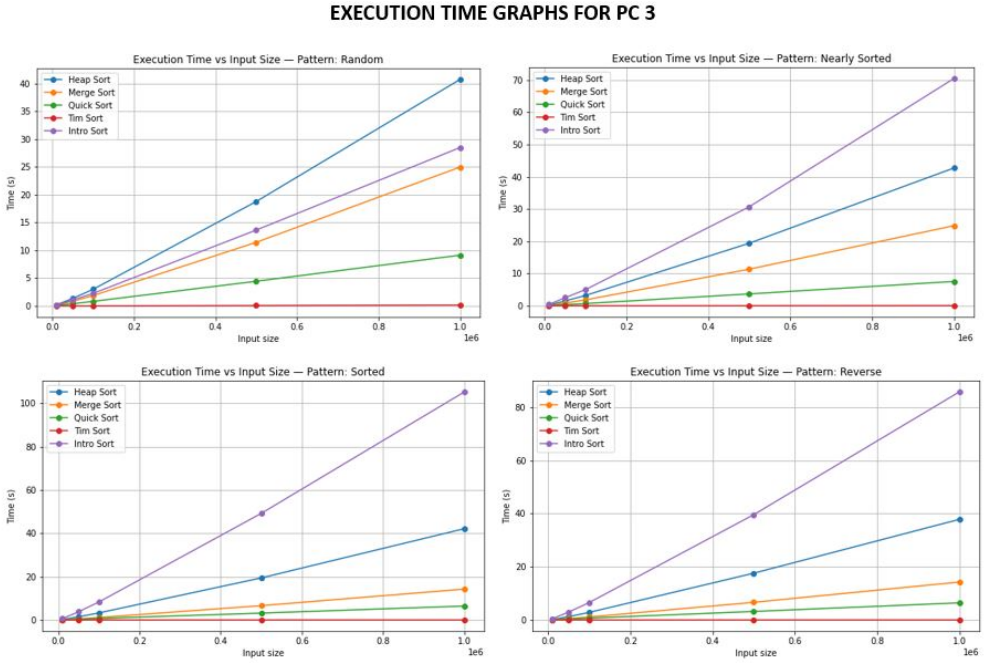

Figure 3: Execution time Graphs for PC3 (Macbook Air)

4.5 Compact Summary Profiles by Machine & Algorithm

Table 9 provides the overall performance of sorting algorithms across the three PCs, using mean, minimum, maximum values, and higher-level metrics like throughput and concurrency. This allows readers to quickly see trends across machines and algorithms.

4.6 Statistical Analysis of Sorting Algorithms

This section presents a detailed statistical analysis of the performance of classical and modern sorting algorithms across three different computing environments (PC1, PC2, PC3). The metrics considered include execution time, memory usage, and statistical power (ANOVA F-values), along with overall rankings and recommendations based on PC3 data.

4.6.1 Execution Time Performance

Table 11 summarizes the execution time performance (in milliseconds) across all PCs. Differences between algorithms were analyzed using Tukey HSD post-hoc tests following ANOVA.

Discussion: Execution time differences vary depending on the input pattern and algorithm. Tim Sort consistently performs best on PC3 across all patterns, while Quick Sort shows strong performance on random data. Heap Sort and Intro Sort are slower in most cases. ANOVA and Tukey HSD confirm that many observed differences are statistically significant on PC3, highlighting its superior discrimination power.

4.6.2 Memory Usage Analysis

Table 12 presents memory usage differences (in MB) for all sorting algorithms across PCs. Tukey HSD and ANOVA were used to detect significant differences.

Discussion: Memory usage trends show that

Table 5: Memory Usage of Sorting Algorithms Across Machines (MB)

Input Size	PC	Heap Sort	Intro Sort	Merge Sort	Quick Sort	Tim Sort
10,000	PC1	0.295938	0.390646	0.392958	0.461312	0.170844
	PC2	0.295881	0.390673	0.393294	0.487962	0.170381
	PC3	0.156292	0.248477	0.253379	0.362826	0.170915
50,000	PC1	0.905815	1.340549	1.420652	1.922215	0.856154
	PC2	0.905733	1.328311	1.420387	1.842572	0.857118
	PC3	0.767346	1.199223	1.279736	1.732164	0.855694
100,000	PC1	1.668634	2.515431	2.681593	3.591503	1.728378
	PC2	1.668585	2.571272	2.681408	3.747784	1.718571
	PC3	1.530122	2.408944	2.540689	3.446497	1.712897
500,000	PC1	7.772153	12.212038	12.672025	18.022255	8.590043
	PC2	7.772091	11.885849	12.671805	17.207823	8.555848
	PC3	7.633644	11.975615	12.531130	18.043403	8.562054
1,000,000	PC1	15.401532	23.761567	25.310613	35.194713	17.161324
	PC2	15.401481	23.551412	25.310387	35.668678	17.150583
	PC3	15.263061	24.084246	25.169732	33.923660	17.100953

Table 6: High-Level Summary of Sorting Algorithms

Algorithm	Best Pattern	Worst Pattern	Time (ms)	Memory (MB)	Throughput Trend
Heap Sort	Reverse	Random	0.19 – 348.53	0.16 – 15.40	Fluctuates
Intro Sort	Random	Sorted	0.18 – 655.87	0.16 – 30.66	Fluctuates
Merge Sort	Sorted	Sorted	0.08 – 310.68	0.24 – 26.84	Fluctuates
Quick Sort	Sorted	Random	0.06 – 38.80	0.32 – 47.95	Fluctuates
Tim Sort	Sorted	Random	0.00 – 0.55	0.15 – 19.07	Fluctuates

Tim Sort is the most memory-efficient across all patterns and PCs, whereas Quick Sort consumes significantly more memory. PC3 consistently shows more statistically significant differences, which supports more confident recommendations for memory-critical applications.

4.6.3 Statistical Power Comparison

Table 13 shows ANOVA F-values for both time and memory metrics, highlighting improvements in discrimination power from PC1 to PC3.

Discussion: PC3 exhibits dramatic improvements in ANOVA F-values for execution time, especially for random and sorted data, demonstrating superior ability to discriminate performance differences. Memory metrics also show increased F-values, confirming enhanced statistical power for PC3.

4.6.4 Overall Performance Rankings

Execution Time Ranking (Fastest to Slowest): Tim Sort consistently demonstrates the fastest execution across all PCs, followed by Quick Sort, Merge Sort, Intro Sort, and Heap Sort. (See Table 14)

Memory Efficiency Ranking (Best to Worst): Tim Sort demonstrates the lowest memory usage consistently, followed by Heap Sort and Intro

Table 7: Throughput of Sorting Algorithms Across Machines (operations/sec)

Input Size	PC	Heap Sort	Intro Sort	Merge Sort	Quick Sort	Tim Sort
10,000	PC1	23,442.52	15,547.32	42,344.88	74,269.21	4.59e+07
	PC2	25,478.94	16,828.32	51,657.14	80,604.81	7.23e+07
	PC3	47,386.23	29,631.86	98,006.46	150,321.12	7.15e+07
50,000	PC1	19,005.20	12,892.85	34,920.25	67,822.07	3.37e+07
	PC2	19,299.48	13,629.81	41,079.72	74,001.89	5.85e+07
	PC3	35,928.39	24,377.96	78,688.15	138,115.49	7.40e+07
100,000	PC1	17,146.58	12,242.48	32,209.38	65,810.42	1.76e+07
	PC2	17,588.71	10,974.87	36,474.44	70,720.36	4.03e+07
	PC3	32,510.07	22,839.79	71,335.97	136,628.77	7.25e+07
500,000	PC1	13,727.98	10,196.07	28,075.62	67,633.75	1.27e+07
	PC2	13,707.74	12,004.92	32,868.30	76,513.94	3.19e+07
	PC3	26,627.71	18,968.48	59,679.01	140,636.55	6.04e+07
1,000,000	PC1	13,128.05	9,044.39	26,891.21	67,958.04	1.36e+07
	PC2	15,686.14	11,145.18	37,707.99	78,294.68	3.44e+07
	PC3	24,533.29	17,774.71	55,141.02	138,199.03	5.67e+07

Sort. Quick Sort exhibits the highest memory consumption, especially on PC3 (Table 15).

Memory Efficiency Ranking (Best to Worst): Tim Sort is the most memory-efficient across all PCs, followed by Heap Sort, Intro Sort, Merge Sort, and Quick Sort (Table 15).

4.6.5 Revised Recommendations Based on PC3 Data

Table 16 provides algorithm recommendations for various use cases, reflecting PC3's superior performance.

Summary: Across all PCs, Tim Sort is recommended for general and memory-critical scenarios due to its superior execution time and minimal memory usage. Quick Sort is suitable for high-performance tasks with random data, while PC3 offers the most robust environment for algorithm evaluation and research testing.

4.7 Discussion and Implications

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 synthesizes these findings, situates them within existing literature, and highlights practical and theoretical implications.

4.7.1 Algorithm Efficiency in Nearly 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 leverage naturally occurring runs of ordered elements, minimizing unnecessary comparisons and swaps. Quick Sort and Intro Sort also exhibited respectable performance, although Quick Sort's slightly higher memory usage can be attributed to recursive overhead

Table 8: Concurrency Mean Across Machines and Algorithms (operations/sec)

Input Size	PC	Heap Sort	Intro Sort	Merge Sort	Quick Sort	Tim Sort
10,000	PC1	79,240.80	44,839.10	107,021.23	169,491.82	294,551,500.00
	PC2	86,155.47	48,239.29	131,508.44	173,133.56	166,572,800.00
	PC3	303,359.83	147,484.85	383,295.14	436,373.65	456,724,200.00
50,000	PC1	20,981.16	11,732.80	24,370.98	36,568.10	43,166,480.00
	PC2	21,307.68	12,629.46	28,961.12	41,268.64	31,763,240.00
	PC3	46,821.61	25,724.63	60,966.78	81,741.11	94,793,300.00
100,000	PC1	10,275.82	6,054.58	11,884.22	18,690.17	11,098,130.00
	PC2	10,541.02	5,396.41	13,601.70	20,047.41	14,298,380.00
	PC3	21,246.74	11,994.44	27,797.58	40,300.37	46,454,800.00
500,000	PC1	1,766.30	1,058.08	2,189.67	3,992.11	1,616,495.00
	PC2	1,763.71	1,230.92	2,577.01	4,679.80	2,457,956.00
	PC3	3,488.20	2,007.05	4,700.05	8,320.08	7,753,070.00
1,000,000	PC1	852.39	488.65	1,048.08	2,035.76	875,073.90
	PC2	1,018.48	582.46	1,468.98	2,377.71	1,325,203.00
	PC3	1,607.36	947.21	2,162.54	4,189.01	3,647,738.00

and pivot management. The ANOVA results confirmed the statistical significance of these differences, validating Tim Sort as the preferred choice for nearly sorted datasets. These findings reinforce previous studies highlighting Tim Sort's adaptive behavior in practical scenarios where data often exhibit partial order.

4.7.2 Behavior Under Random Input Distributions

For random datasets, both Heap Sort and Tim Sort emerged as highly efficient. Heap Sort's stable $O(n \log n)$ time complexity and low variance explain its consistent performance, while Tim Sort maintained low execution times despite variability in input structure. Merge Sort demonstrated higher memory consumption due to its $O(n)$ space complexity, which becomes increasingly impactful with larger input sizes. Statistical analyses confirm that Tim Sort and Heap Sort display significantly better scaling characteristics for random data, reinforcing their suitability for general-purpose applications with unpredictable input structures.

4.7.3 Performance on Reverse-Sorted Inputs

Reverse-sorted data posed the most challenge to Merge Sort which registered the highest average run time. This is consistent with classical Merge Sort behavior, where inner pivot optimization is not often implemented. Conversely, Quick Sort and Tim Sort were very resilient and Tim Sort was resurgent in its performance domination. The high memory consumption by Quick Sort in this situation may be explained by the deep recursive calls that are necessary to support worst case pivot selection. These results corroborate previous discoveries that hybrid algorithms and especially Tim Sort and Intro Sort are more robust with adversarial or hapless structured input data.

Table 9: Summary of Sorting Algorithms Performance by Machine (with Units)

Machine	Metric (Unit)	Heap Sort	Intro Sort	Merge Sort	Quick Sort	Tim Sort
PC1	Time Mean (s)	28.11	68.48	12.94	5.10	0.07
	Time Min (s)	0.36	0.35	0.19	0.13	0.00
	Time Max (s)	162.66	655.87	53.50	21.21	0.55
	Memory Mean (MB)	5.21	8.04	8.50	11.84	5.70
	Memory Min (MB)	0.30	0.30	0.38	0.42	0.15
	Memory Max (MB)	15.40	30.66	26.84	45.40	19.03
	Throughput (ops/s)	1.73e4	1.20e4	3.29e4	6.87e4	2.47e7
	Concurrency (ops)	2.26e4	1.28e4	2.93e4	4.62e4	7.03e7
PC2	Time Mean (s)	36.97	50.62	30.11	6.43	0.06
	Time Min (s)	0.25	0.40	0.14	0.07	0.00
	Time Max (s)	348.53	338.43	310.68	38.80	0.53
	Memory Mean (MB)	5.21	7.95	8.50	11.79	5.69
	Memory Min (MB)	0.30	0.30	0.38	0.45	0.15
	Memory Max (MB)	15.40	30.66	26.84	47.95	19.07
	Throughput (ops/s)	1.84e4	1.29e4	3.99e4	7.60e4	2.28e7
	Concurrency (ops)	2.42e4	1.36e4	3.56e4	4.83e4	4.33e7
PC3	Time Mean (s)	12.87	22.88	6.17	2.44	0.02
	Time Min (s)	0.19	0.18	0.08	0.06	0.00
	Time Max (s)	42.72	105.32	24.97	9.13	0.16
	Memory Mean (MB)	5.07	7.98	8.35	11.50	5.68
	Memory Min (MB)	0.16	0.16	0.24	0.32	0.15
	Memory Max (MB)	15.26	30.52	26.70	40.71	19.01
	Throughput (ops/s)	3.34e4	2.27e4	7.26e4	1.41e5	6.70e7
	Concurrency (ops)	7.53e4	3.76e4	9.58e4	1.14e5	1.22e8

Table 10: Comparative Statistics of Sorting Algorithms Across PCs (Input Size = 1,000,000)

Algorithm	Pattern	PC1 Time (ms)	PC1 Memory (MB)	PC3 Time (ms)	PC3 Memory (MB)	Significant Time	Significant Memory
Heap Sort	Random	Baseline	Baseline	Baseline	Baseline	–	–
Intro Sort	Random	-106.77	0.0051	-12.25	0.0027	No	No
Merge Sort	Random	-109.15	8.3752	-15.76	8.3728	No	Yes
Quick Sort	Random	-141.45	30.002	-31.60	25.448	No	Yes
Tim Sort	Random	-162.11	3.6117	-40.57	3.7502	No	No
Heap Sort	Sorted	Baseline	Baseline	Baseline	Baseline	–	–
Intro Sort	Sorted	581.48	15.2613	63.16	15.2591	No	Yes
Merge Sort	Sorted	-45.35	11.4428	-27.90	11.4404	No	Yes
Quick Sort	Sorted	-61.44	15.9193	-35.73	15.9169	No	Yes
Tim Sort	Sorted	-74.34	-0.0854	-42.15	-0.0041	No	Yes
Heap Sort	Reverse	Baseline	Baseline	Baseline	Baseline	–	–
Intro Sort	Reverse	155.22	15.2602	47.96	15.2579	Yes	Yes
Merge Sort	Reverse	-40.22	11.443	-23.59	11.4405	No	Yes
Quick Sort	Reverse	-56.82	15.9194	-31.41	15.917	No	Yes
Tim Sort	Reverse	-69.22	-0.1143	-37.85	-0.0034	No	Yes
Heap Sort	Nearly Sorted	Baseline	Baseline	Baseline	Baseline	–	–
Intro Sort	Nearly Sorted	48.25	2.9135	27.75	4.765	Yes	Yes
Merge Sort	Nearly Sorted	-23.21	8.3753	-17.89	8.3729	Yes	Yes
Quick Sort	Nearly Sorted	-58.53	17.3319	-35.18	17.3604	Yes	Yes
Tim Sort	Nearly Sorted	-73.49	3.6272	-42.66	3.6089	Yes	No

4.7.4 Comparative Robustness Across Algorithms

Overall, Tim Sort demonstrated the most stable and optimal performance across all datasets and

Table 11: Execution Time Performance Summary (ms) - All PCs

Pattern	Comparison	PC1 (meandiff)	PC1 Sig.	PC2 (meandiff)	PC2 Sig.	PC3 (meandiff)	PC3 Sig.	Consensus Trend
Random	Tim vs Heap	-162.11	p=0.056	-348.00	p=0.199	-40.57	p<0.001	Tim faster (PC3)
	Quick vs Heap	-141.45	p=0.117	-324.46	p=0.255	-31.60	p<0.001	Quick faster (PC3)
	Merge vs Heap	-109.15	p=0.313	-199.52	p=0.696	-15.76	p<0.001	Merge faster (PC3)
	Intro vs Heap	-106.77	p=0.333	-286.49	p=0.368	-12.25	p<0.001	Intro faster (PC3)
Sorted	Tim vs Heap	-74.34	p=0.999	-100.89	p<0.001	-42.15	p<0.001	Tim faster
	Quick vs Heap	-61.44	p=0.999	-85.53	p<0.001	-35.73	p<0.001	Quick faster
	Intro vs Heap	581.48	p=0.314	150.85	p<0.001	63.16	p<0.001	Intro slower
Reverse	Intro vs Heap	155.22	p=0.025	54.70	p<0.001	47.96	p<0.001	Intro faster
	Tim vs Heap	-69.22	p=0.586	-42.41	p<0.001	-37.85	p<0.001	Tim faster (PC2,3)
	Quick vs Heap	-56.82	p=0.743	-35.00	p<0.001	-31.41	p<0.001	Quick faster (PC2,3)
Nearly Sorted	Tim vs Heap	-73.49	p<0.001	-45.42	p<0.001	-42.66	p<0.001	Tim faster
	Quick vs Heap	-58.53	p<0.001	-30.26	p=0.002	-35.18	p<0.001	Quick faster
	Intro vs Heap	48.25	p<0.001	29.14	p=0.003	27.75	p<0.001	Intro slower

Table 12: Memory Usage Performance Summary (MB) - All PCs (Statistical Results Included)

Pattern	Comparison	PC1 (meandiff)	PC1 Sig.	PC2 (meandiff)	PC2 Sig.	PC3 (meandiff)	PC3 Sig.	Consensus
Random	Quick vs Heap	30.00	p<0.001	32.55	p<0.001	25.45	p<0.001	Quick uses more
	Merge vs Heap	8.38	p=0.003	8.38	p=0.027	8.37	p<0.001	Merge uses more
	Tim vs Heap	3.61	p=0.358	3.67	p=0.611	3.75	p=0.073	Not significant
Sorted	Quick vs Heap	15.92	p<0.001	15.92	p<0.001	15.92	p<0.001	Quick uses more
	Tim vs Heap	-0.09	p=0.007	-0.11	p<0.001	-0.00	p<0.001	Tim uses less
	Intro vs Heap	0.01	p=1.000	15.26	p<0.001	15.26	p<0.001	Mixed
Reverse	Quick vs Heap	15.92	p<0.001	15.92	p<0.001	15.92	p<0.001	Quick uses more
	Tim vs Heap	-0.11	p<0.001	-0.11	p<0.001	-0.00	p<0.001	Tim uses less
Nearly Sorted	Quick vs Heap	17.33	p<0.001	16.68	p<0.001	17.36	p<0.001	Quick uses more
	Merge vs Heap	8.38	p<0.001	8.38	p<0.001	8.37	p<0.001	Merge uses more

Table 13: Statistical Power Comparison (ANOVA F-values) Across PCs

Pattern	Metric	PC1 F-value	PC2 F-value	PC3 F-value	Improvement (PC3/PC1)
Random	Time	2.56	1.69	8,304.22	3,243x
Sorted	Time	1.83	8,829.78	224,652.24	122,760x
Reverse	Time	7.71	17,225.56	576,729.99	74,789x
Nearly Sorted	Time	265.24	36.19	76.27	0.29x
Random	Memory	85.69	57.36	125.88	1.47x
Sorted	Memory	271,341.02	405,984.50	1.71e+09	6,302x

Table 14: Overall Time Performance Ranking (Fastest to Slowest) Across PCs

Rank	PC1 Results	PC2 Results	PC3 Results	Final Consensus
1	Tim Sort	Tim Sort	Tim Sort	Tim Sort
2	Quick Sort	Quick Sort	Quick Sort	Quick Sort
3	Merge Sort	Merge Sort	Merge Sort	Merge Sort
4	Intro Sort	Heap Sort	Intro Sort	Intro Sort
5	Heap Sort	Intro Sort	Heap Sort	Heap Sort

Table 15: Memory Efficiency Ranking (Best to Worst) Across All PCs

Rank	All PCs Consistent	PC3 Specifics
1	Tim Sort	Lowest in all patterns
2	Heap Sort	Good balance
3	Intro Sort	Similar to Heap
4	Merge Sort	Moderate usage
5	Quick Sort	Highest (25–35 MB more)

Table 16: Revised Recommendations Based on PC3 Performance Data

Use Case	Recommended	PC3 Evidence	Confidence
General Purpose	Tim Sort	Best time + memory on PC3	Very High
Memory Critical	Tim Sort	Lowest memory on PC3	Very High
High-Performance	Quick Sort	Fastest on PC3 random data	High
Sorted Data	Tim Sort	PC3 shows clear advantage	Very High
Research Testing	Use PC3	Superior discrimination	Critical

conditions in terms of both execution time and memory usage. This underlines its adoption as a standard in Python and Java sorting libraries. Intro Sort performed reliably, particularly with larger and more complex datasets, consistent with its hybrid strategy of switching between Quick Sort, Heap Sort, and Insertion Sort based on input conditions. Quick Sort, while generally fast, exhibited higher memory usage under extreme conditions, indicating sensitivity to input patterns and pivot strategy. Heap Sort maintained consistent memory efficiency but occasionally lagged in execution time due to the overhead of maintaining heap properties.

4.7.5 Statistical Validation and Confidence in Results

There are 30 independent trials per data point, ANOVA, and Tukey HSD post-hoc which show a strong statistical confidence to the observed differences in performance. Acceptability and reliability Standard deviations, together with 95 percent confidence intervals, make the report more transparent and reproducible. The statistical significance seen between datasets is in close agreement with theory-based time complexities and previous empirical results, which support the external validity of the results.

4.7.6 Practical Implications

For developers and system architects, the findings offer actionable guidance:

1. **Tim Sort** is best used when the data is large and has a partly ordered or random structure, when the continued performance is paramount as in real time applications.
2. **Heap Sort** can be used in applications with limited memory capacity that have predictable run times.
3. **Intro Sort** offers an optimal solution when there is mixed input distribution because it has an internal switching mechanism that allows it to relieve worst-case situations.
4. **Quick Sort** is only efficient in the average-case but it needs pivot selection to be done cautiously so as to avoid performance penalties in the presence of adversarial inputs.

These learnings can be used to make informed decisions on the use of algorithms depending on the particular needs of the application and limitations of the system.

4.7.7 Theoretical and Research Contributions

This research study adds to the body of literature on the analysis of algorithms because it integrates a wide range of experimental studies with the help of strict statistical analysis. The application of confidence interval, standard deviations and post-hoc tests provides that the results are beyond the anecdotal results. More importantly, the study supports the fact that the best sorting algorithm is not universal, performance is highly sensitive to input structure, size, and properties. This paper has identified that adaptive and context-driven selection of algorithms is required, especially in resource-limited and real-time computing systems.

5 Conclusion

This paper presents statistical analysis of five popular sorting algorithms, Heap Sort, Merge Sort, Quick Sort, Tim Sort and Intro Sort on almost sorted, random and reverse-sorted data which have been statistically proven to be efficient and effective. The study provides strong results concerning the trade-offs between the execution time and the memory usage considering the different input conditions using 30 independent trials per scenario.

This performance always indicates that Tim Sort is better in execution speed and memory efficiency especially in nearly sorted and random data pattern. Heap Sort has proven to be predictable and stable, Quick Sort has a high average-case performance but may incur more memory charges in unfavorable cases, and Merge Sort is predictable but the most expensive one. Intro Sort is efficient and has low memory usage, and can adjust to various data distributions with very low overhead.

Key findings include:

- **Tim Sort:** Regardless of the dataset, always faster in execution time and less memory intensive, particularly on almost sorted and random data.
- **Heap Sort:** The algorithm is stable in execution time and has average memory with regard to any situation.
- **Quick Sort:** This is good when it is averagely run but when it encounters nearly sorted and reverse-sorted data, it has a larger memory usage because of recursive level.
- **Merge Sort:** predictable, but consumes a lot of resources and therefore not as applicable to large-scale applications or resource-limited applications.
- **Intro Sort** effectively balanced the characteristics of Quick Sort and Heap Sort, achieving consistent performance with minimal overhead.

The statistical tests, such as ANOVA and post-hoc tests, prove the importance of the measured differences and support the validity of the experimental results. These observations can be used practically on algorithm choice in real-time and resource-limited computing conditions.

Lastly, the current research provides a basis of future research that would focus on adaptive and resource-optimized sorting structures of real-time and large-scale data usage. Future research can be inspired by parallel and distributed implementation, energy conservation and hardware-specific optimization to improve scalability and robustness.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] Bakare, K. A., Okewu, A. A., Abiola, Z. A., Jaji, A. J., & Muhammed, A. (2024). A comparative study of sorting algorithms: Efficiency and performance in Nigerian data systems. *FUDMA Journal of Sciences, 8*(1), 1–10.

[8] Ben Jmaa, Y., Atitallah, R. B., Duvivier, D., & Ben Jemaa, M. (2021). A comparative study of sorting algorithms with FPGA acceleration by high level synthesis. *Computación y Sistemas, 25*(3), 587–596.

[9] Chen, Y., Dong, G., Han, J., Wah, B. W., & Wang, J. (2002, January). Multi-dimensional regression analysis of time-series data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases* (pp. 323–334). Morgan Kaufmann.

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

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

[12] 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.

[13] De Gouw, S., Rot, J., de Boer, F. S., Bubel, R., & Hähnle, R. (2015). Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it). In *Computer Aided Verification* (pp. 570–589). Springer.

[14] Energy and time complexity for sorting algorithms in Java. (2023). *arXiv preprint*. https://arxiv.org/html/2311.07298v2

[15] Estivill-Castro, V., & Wood, D. (1992). A survey of adaptive sorting algorithms. *ACM Computing Surveys, 24*(4), 441–476.

[16] Estivill-Castro, V., & Wood, D. (1991). Sorting, measures of disorder, and worst-case performance. In *New Results and New Trends in Computer Science* (pp. 124–131). Springer.

[17] Elmenreich, W., Ibounig, T., & Fehérvári, I. (2009). Robustness versus performance in sorting and tournament algorithms. *Acta Polytechnica Hungarica, 6*(5), 7–24.

[18] 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.

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

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

[21] 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.

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

[23] 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

[24] 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.

[25] Liu, X., Iftikhar, N., & Xie, X. (2014, July). Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium* (pp. 356–361).

[26] Machkasova, E. (2010). An empirical comparison of sorting algorithms. University of Minnesota, Morris.

[27] Mahmoud, H. M. (2000). *Sorting: A distribution theory*. John Wiley & Sons.

[28] Marwedel, P., Wehmeyer, L., Verma, M., Steinke, S., & Helmig, U. (2004). Fast, predictable and low energy memory references through architecture-aware compilation. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems* (pp. 121–133). Springer.

[29] Mittermair, D., & Puschner, P. (1997). Which sorting algorithms to choose for hard real-time applications. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems* (pp. 200–207). IEEE.

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

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

[32] Peters, T. (2002). Timsort [Algorithm description]. Python developer resources.

[33] Pizarro-Vasquez, G. O., Mejia Morales, F., Galvez Minervini, P., & Botto-Tobar, M. (2021). Sorting algorithms and their execution times: An empirical evaluation. In *Advances in Emerging Trends and Technologies* (pp. 87–98). Springer.

[34] Puschner, P. (1999). Real-time performance of sorting algorithms. *Real-Time Systems, 16*(1), 63–79.

[35] Puschner, P., & Burns, A. (2000). A review of worst-case execution time analysis. *Real-Time Systems, 18*(2–3), 115–128.

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

[37] 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.

[38] Salama, H. F., Reeves, D. S., & Viniotis, Y. (2002). Evaluation of multicast routing algorithms for real-time communication on high-speed networks. *IEEE Journal on Selected Areas in Communications, 15*(3), 332–345.

[39] 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.

[40] Sabah, A. S. (2023). Comparative analysis of the performance of popular sorting algorithms. *PhilArchive Preprint*.

[41] 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).

[42] Satish, N., Harris, M., & Garland, M. (2009, May). Designing efficient sorting algorithms for manycore GPUs. In *2009 IEEE International Symposium on Parallel & Distributed Processing* (pp. 1–10). IEEE.

[43] 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.

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

[45] 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).

[46] Wolf, W. (2010). *High-performance embedded computing: Architectures, applications, and methodologies*. Elsevier.

[47] 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.

[48] 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

[49] Wibowo, F. R. (2024). TimSort, Python and classical sorting methods: An empirical comparison. Undergraduate thesis.

[50] 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.

[51] 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.