Vol. 7, No. 4, December 2025

ISI Journal of Information Systems and Informatics

Published By Asosiasi Doktor Sistem Informasi Indonesia

A Unified Framework for Theoretical and Experimental Evaluation of Classical and Modern Sorting Algorithms in Real-Time Systems

Stephen Akobre [1], Japheth Kodua Wiredu [2,*], Iven Aabaah [3] and Umar Adam Wumpini [4]

[1]Department of Cyber Security and Computer Engineering Technology, C. K. Tedam University of Technology and Applied Sciences, Navrongo, Ghana

[2]Department of Computer Science, Regentropfen University College, Bolgatanga, Ghana.

[3]Department of Information Systems & Technology, C. K. Tedam University of Technology and Applied Sciences, Navrongo, Ghana

[4]Department of Mathematics and ICT, Gambaga College of Education, Ghana

Email: [1]sakobre@cktutas.edu.gh, [2]wiredujapheth130@gmail.com, [3]iaabaah@cktutas.edu.gh, [4]adamnayib30@gmail.com

Received: October 10, 2025
Revised: October 23, 2025
Accepted: Nov 11, 2025
Published: Dec 9, 2025

Corresponding Author:
Author Name*:
Japheth Kodua Wiredu
Email*:
wiredujapheth130@gmail.com

DOI:
10.63158/journalisi.v7i4.1287

Abstract. This paper presents a theoretical and experimental evaluation of eight popular sorting algorithms HeapSort, QuickSort, MergeSort, Parallel MergeSort, TimSort, IntroSort, Bitonic Sort, and MSD Radix Sort—assessing their suitability for real-time computing environments. The study combines algorithmic analysis with large-scale benchmarks across various input distributions (random, almost sorted, reverse-sorted) and data scales, focusing on execution time and memory usage. Results show that hybrid and adaptive algorithms outperform classical ones. TimSort had the shortest execution times (as low as 1.0 ms on sorted data), and IntroSort showed consistent performance across data types (11-13 ms on random inputs) with minimal memory (<7.90 MB). HeapSort maintained predictable O (n log n) behavior, suitable for hard real-time constraints, while QuickSort and MergeSort had lower latency but higher memory usage. These findings are significant for latency-sensitive applications like high-frequency trading and sensor data processing. The study recommends using hybrid algorithms like TimSort and IntroSort for general-purpose workloads, providing evidence-based guidance for real-time system design.

Keywords: Sorting algorithms, Real-time systems, Hybrid algorithms, Performance evaluation

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

1. INTRODUCTION

Sorting algorithms are regarded as one of the basic pillars of computer science, which allows to organize, locate, and process data economically in various computational fields [1, 2]. The efficiency of sorting algorithms directly influences overall computational performance, particularly in domains such as database management, search optimization, high-frequency trading, cloud computing, and embedded real-time systems [3, 4]. The growing prevalence of data-intensive applications from IoT sensor networks and real-time financial transactions to edge computing has further elevated the importance of predictable, low-latency sorting in constrained environments. The current computing environment, which is typified by the massive increase in data, high real-time performance demands, and the need to operate at larger scales than ever before has put a greater strain on the need to find sorting algorithms that can marry theory with practical implementation than ever before [2, 5].

The classical algorithms, such as Heapsort, Quicksort, or MergeSort have been used for a long time as a benchmark both in theory and practice [1, 3]. Although these algorithms can be remarkably successful in offering great strong asymptotic guarantees and fully elegant analytical properties [3, 6], they tend to be limited in the modern environment that involves the need to be adaptable, cache-aware, and run concurrently [2, 4]. Subsequently, adaptive, memory-hierarchy, and parallelist approaches like TimSort, IntroSort, Parallel Merge Sort, Bitonic Sort, and MSD Radix Sort have been developed, which aim to provide significant improvements in scalability and throughput, particularly in real-time systems where predictable latency and determinism are the most important factors [2, 4, 5].

Regardless of all the research, there is still a major gap: that of an evaluative system which will systematically combine theory with practical verification of sorting algorithms with real-time constraints. Specifically, there is a lack of a unified framework that cohesively integrates theoretical analysis (e.g., pseudocode, complexity proofs) with empirical benchmarking for both classical and modern algorithms under real-time conditions. The current literature generally pays attention to one of formal complexity or experimental benchmarking without systematic combination [1, 2]. This partitioning

does not provide practitioners and researchers with full guidance in terms of algorithm choice in latency-critical, data-intensive setting [3, 4].

This paper fills that gap by proposing a unified framework on the evaluation of sorting algorithms in both the classical and modern paradigms. Both algorithms are given in pseudocode, formal complexity analysis and with feasible mathematical proofs of correctness. These theoretical bases are further complemented by strict experimental verification in real-time scenarios, which guarantees reproducible and holistic performance understanding [1, 2]. The framework makes predictions and observations consistent at the theoretical level and empirical evidence to come up with strong conclusions regarding the efficiency of the algorithm, its scalability, and reliability.

The work is based on the recent developments in the area, such as the proximity-based sorting algorithm of real-time numeric data streams by [7] and the adaptations to the HeapSort algorithm in the case of duplicate-heavy datasets by [8]. Such contributions highlight the importance of adapting algorithms to the characteristics of data, a principle that serves as the central foundation of our methodology. Also, [9] introduced OptiFlexSort, is an example of the trend of adoption of hybrid solutions that are more effective in processing large data volumes.

This study will contribute to the scientific literature and practice of sorting algorithms, as it will bridge the gap between the theoretical analysis of the algorithms and the application of existing algorithms to practical implementation. The results have direct consequences to scalable systems like large-scale data analytics, artificial intelligence, high-performance computing, and edge-based real time systems where efficiency, scalability, and predictable performance continue to be critical to system integrity.

2. RELATED WORKS

Sorting algorithms are one of the most widely researched fields in computer science, and research into them has included both theoretical background and practical implementation in addition to new paradigms of computation. It has developed much beyond a pioneering emphasis on comparison-based algorithms, to include parallel,

adaptive, and specialized solutions to the needs of contemporary hardware architectures and applications [10], [11], [12].

2.1. Foundational Sorting Algorithms and Theoretical Frameworks

The mathematical foundations of sorting algorithms are fully described in classical literature. The original landmark works on inflexible methods of sorting data is still the Art of Computer Programming by [10], which is rigorously analyzed mathematically and frames computational complexity concepts. This was furthered by [11] that with empirical case studies of various systems and [12] gave algorithmic blueprints that are common in the academia. Quicksort, MergeSort, and Heapsort are occasionally known as classical algorithms because of their predictable complexity and sound analytical guarantees [13], [3], [6]. Beyond these developments, theoretical and computational models have offered valuable insights. As an example, [3] focused on sorting in a graph grammar model, and [3] offered the worst-case execution time (WCET) analysis needed in real-time applications. They are used as reference points against which new practices are compared [1], [2].

2.2. Parallel and Distributed Sorting Architectures

The emergence of multicore processors, graphics cards and distributed systems have made parallel sorting solutions to be prioritized. Regular sampling was first used to perform load-balanced parallel sorting by [14], and radix sort was first performed with dramatic speedups on a CUDA architecture by [15]. [16] generalized radix sorting to in-place, to cache-efficient parallel designs and [17] studied parallel Quicksort designs on distributed memory systems. Bitonic sorting networks, which were previously viewed as theoretical, have acquired a new meaning in terms of hardware and GPU scenarios [19]. Recent publications show that they are applicable to high-performance systems in which deterministic parallelism is vital [20]. Parallel Merge sort has also experienced a lot of development, where [21] and [4] demonstrate optimized heterogeneous and cloud-based implementations. These works highlight the relevancy of load distribution, communication overhead and memory efficiency in parallel sorting.

2.3. Hybrid and Adaptive Algorithmic Strategies

Hybrid sorting algorithms were developed as a reaction to the inadequacy of single methods. IntroSort [18], a variant of Quicksort with the addition of Heapsort guarantees to avoid the worst case and the strengths of InsertionSort to small datasets, was proposed. An algorithm known as TimSort by [23], is a combination of MergeSort and Insertion Sort that builds on the order present and has since been the default algorithm in Python and Java. These hybrids are robust, which is proven by empirical research. This study [23] optimized integer-data radix-based hybrids and [2] gave comparative research based on heterogeneous computing systems. The new hybrid construction (as shown by [24] in real-time hybrids and [9] in OptiFlexSort) also exhibits scalability in large-scale and real-time applications. All of these works point to the effectiveness of hybridization in embedding the theoretical assurances with the efficiency of practice.

2.4. Non-Comparative Sorting and Domain-Specific Adaptations

Radix and Counting Sort are related to non-comparative methods that can take advantage of data properties in order to perform in linear time in application to a specific type [21], [25]. This research [16] demonstrated that radix methods scale linearly to parallel systems, whereas [23] presented optimizations to increase integer sorting throughput. Beyond extensions to domain-specific sorting, the extensibility of sorting can also be seen in a proximity-based sorting algorithm on numerical data streams by [7] and an adaptation of Heapsort to duplicate heavy data sets by [8]. The innovations demonstrate that obtaining performance benefits on the order of magnitude through adapting algorithms to data distributions and environments are possible.

2.5. Empirical Evaluation and Real-Time Performance

Although the theoretical properties are well known, the performance of it in the conditions of the real world has become a research priority. This study [1] reviewed systematically the concept of scalability in big data environments, and paper [2] compared the classical algorithms and modern algorithms on heterogeneous architecture. [4] have placed an accent on real-time data streams, with latency as one of the bottlenecks. [8] empirically validated the modifications of heaps and showed that it was efficient with duplicate-heavy data. Hardware architecture and algorithm design interaction is also an area that has had intensive research. [5] proposed the cache-aware

sorting algorithms that can be optimized with modern processors with [24] showing that hybrids could be accelerated using a GPU with real-time applications. Taken together, these studies highlight that performance cannot be considered to be fully explained in terms of asymptotic complexity empirical benchmarking is still essential.

2.6. Emerging Paradigms and Future Directions

New computing paradigms continue to expand the scope of sorting research. Cloud-based distributed sorting frameworks have been investigated in Hadoop/MapReduce contexts [21], while edge computing applications require lightweight, cache-efficient algorithms [5], [26]. Hardware acceleration has also gained traction: [18] and [15] showed how GPU-specific designs achieve massive throughput improvements, while [27] addressed predictability in embedded and real-time systems.

2.7. Research Gap and Contribution

Despite extensive progress across theory, hybrid strategies, and empirical studies, a critical research gap persists: the lack of a unified framework that integrates theoretical analysis, mathematical correctness, and empirical benchmarking for sorting algorithms in real-time contexts. Most prior studies emphasize either complexity theory [10], [3] or empirical validation [1], [4], but rarely both in a cohesive manner. This study addresses that gap by introducing a unified evaluation framework encompassing pseudocode specifications, complexity analysis, correctness proofs, and experimental validation. Building on prior innovations [7], [8], [9], the framework facilitates reproducible, rigorous assessment of sorting algorithms across real-time computing scenarios. By bridging theoretical rigor with empirical grounding, it advances understanding of sorting performance in data-intensive, latency-sensitive environments.

3. METHODOLOGY

This study adopts a unified methodological framework that integrates theoretical analysis and experimental validation to evaluate the performance of classical and modern sorting algorithms under real-time computing constraints. The methodology is organized into five key phases: algorithm selection, theoretical specification, experimental design, performance evaluation, and validation.

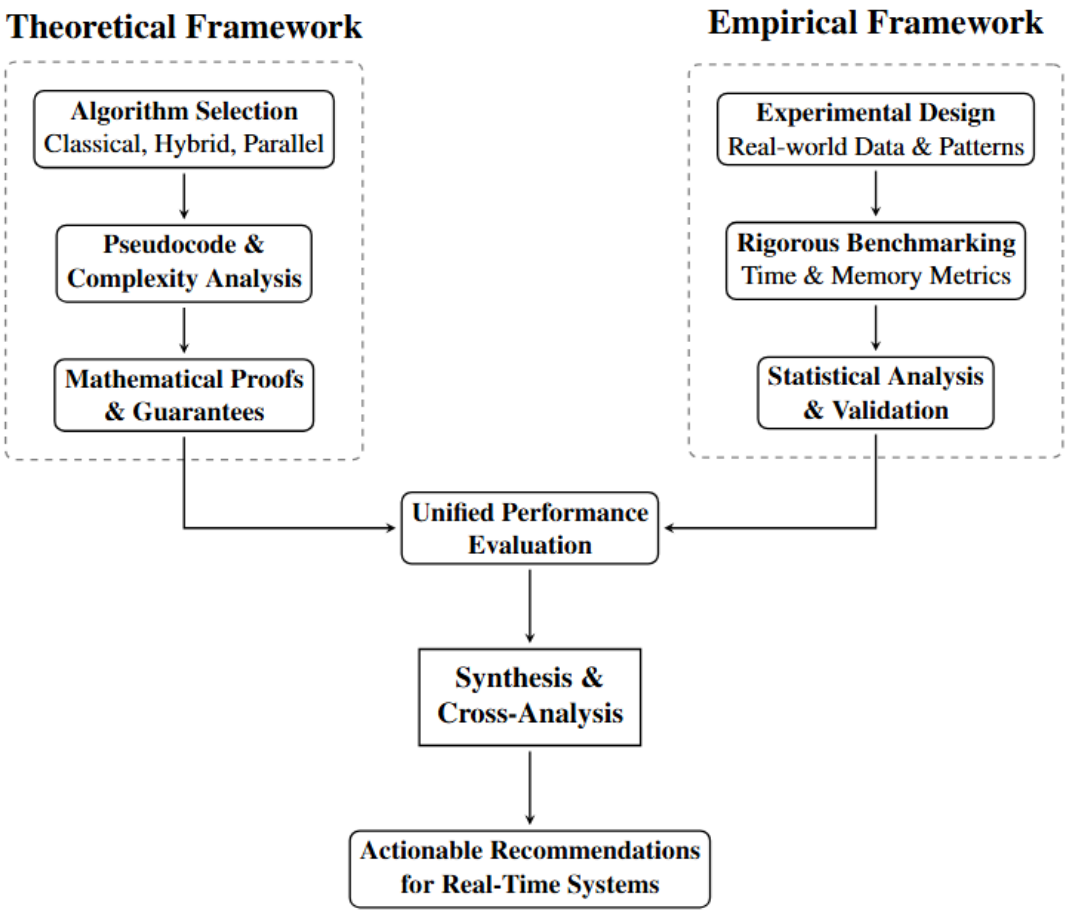

Figure 1. Compact unified framework for theoretical and empirical algorithm evaluation in real-time systems.

3.1 Experimental Setup

The experiments were conducted on a workstation running Windows 10 equipped with an Intel Core i7-12700K CPU, 16 GB RAM, and a 1 TB NVMe SSD. Algorithms were implemented and benchmarked using Python 3.11, leveraging libraries such as numpy, matplotlib, and memory_profiler.

1) Algorithm Selection Criteria

The eight algorithms were selected to provide a comprehensive comparison across different algorithmic paradigms and eras. The selection criteria were designed to include:

a. Classical Comparison-Based Algorithms: HeapSort, QuickSort, and MergeSort were chosen as foundational benchmarks with well-understood theoretical properties.

b. Modern Hybrid Algorithms: TimSort and IntroSort were included as state-of-the-art examples that combine multiple strategies (e.g., MergeSort/InsertionSort, QuickSort/ HeapSort) for robust performance.
c. Parallel Algorithms: Parallel MergeSort and Bitonic Sort were selected to represent architectures that leverage concurrency, with the latter being particularly relevant for GPU-based systems.
d. Non-Comparison-Based Algorithms: MSD Radix Sort was chosen to represent a linear-time algorithm that exploits data structure (digits/characters).

This diverse selection ensures the evaluation covers in-place vs. out-of-place, stable vs. unstable, comparison-based vs. non-comparison-based, and classical vs. modern adaptive strategies.

2) Dataset Selection, Input Patterns, and Assumptions

Input data was derived from the goodbooks-10k dataset, a publicly available collection of book ratings. This dataset was selected for its real-world provenance and representativeness of typical integer-formatted data encountered in real-time systems, such as sensor IDs, transaction amounts, or user priority scores.

The four input patterns namely random, nearly sorted, reverse-sorted, and fully sorted were specifically chosen to model real-world data characteristics:

a. Random Data: Serves as a standard baseline for average-case performance.
b. Nearly Sorted Data: Models common real-time scenarios like incremental data updates (e.g., log files with new entries, sensor readings with minor fluctuations, or maintaining a sorted list after minor modifications).
c. Reverse-Sorted Data: Represents a worst-case scenario for many algorithms (e.g., naive QuickSort) and tests algorithmic resilience to adversarial input.
d. Fully Sorted Data: Tests an algorithm's ability to optimally handle best-case conditions, which is crucial for systems processing already-ordered data streams.

A key assumption made during testing was the uniformity of data types; all inputs were composed of 32-bit integers. This controlled for variability and allowed a focused comparison of the algorithmic logic itself, independent of complex data comparison

functions. Furthermore, random delays (0.1–1 s) were introduced between data chunks to mimic the intermittent arrival of data in real-time conditions.

3.2. Algorithmic Framework

To guarantee generalisability and retain theoretical rigour, each sorting algorithm, (ie. Heapsort, Quicksort, MergeSort, Parallel Merge Sort, TimSort, IntroSort, Bitonic Sort and MSD Radix Sort), is described with its own pseudo-code, formal complexity, and in some cases with a mathematical proof of its correctness and efficiency. This is a composite approach that integrates both the theoretical foundations and experimental validations. This dual approach to emphasize the anticipated performance (analytically determined) as well as the experimental performance (empirically determined).

3.2.1. HeapSort

HeapSort is a classical comparison-based sorting algorithm that leverages the heap data structure to achieve optimal worst-case performance while operating in-place. The algorithm transforms the input array into a binary heap, then repeatedly extracts the maximum element to construct the sorted array in ascending order. Mathematical Analysis of HeapSort as shown in Equation 1 to 10.

1) Time Complexity Analysis:

HeapSort consists of two phases:

Phase 1: Build-Max-Heap

$$T_{\text{build}}(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \quad (1)$$

Simplifying:

$$T_{\text{build}}(n) = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right) \quad (2)$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \quad (\text{Knownseriesidentity}) \quad (3)$$

$$T_{\text{build}}(n) = O(2n) = O(n) \quad (4)$$

Phase 2: Heap Extraction

$$T_{\text{extract}}(n) = \sum_{i=1}^{n-1} O(\log i) = O\left(\sum_{i=1}^{n} \log i\right) \quad (5)$$

Using properties of logarithms and Stirling's approximation:

$$\sum_{i=1}^{n} \log i = \log(n!) \quad (6)$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

$$\log(n!) = n\log n - n\log e + \Theta(\log n) \quad \text{(Stirling)} \tag{7}$$

$$T_{\text{extract}}(n) = O(n\log n) \tag{8}$$

Total Time Complexity:

$$T(n) = T_{\text{build}}(n) + T_{\text{extract}}(n) = O(n) + O(n\log n) = O(n\log n) \tag{9}$$

2) Space Complexity Analysis

HeapSort is an in-place algorithm:

$$S(n) = O(1) \quad \text{(excludinginputarray)} \tag{10}$$

Algorithm 1 Heapsort

```
1:  procedure HEAPSORT(A)                          ▷ Build max heap from unordered array
2:      n ← length(A)
3:      for i ← ⌊n/2⌋ − 1 down to 0 do
4:          current ← i
5:          while true do                          ▷ MaxHeapify
6:              largest ← current
7:              l ← 2 × current + 1, r ← l + 1
8:              if l < n and A[l] > A[largest] then
9:                  largest ← l
10:             end if
11:             if r < n and A[r] > A[largest] then
12:                 largest ← r
13:             end if
14:             if largest = current then break
15:             end if
16:             SWAP(A[current], A[largest])
17:             current ← largest
18:         end while
19:     end for                                    ▷ Extract elements from heap one by one
20:     for i ← n − 1 down to 1 do
21:         SWAP (A [0], A[i])
22:         current ← 0, heapSize ← i
23:         while true do                          ▷ MaxHeapify root
24:             largest ← current
25:             l ← 2 × current + 1, r ← l + 1
26:             if l < heapSize and A[l] > A[largest] then
27:                 largest ← l
28:             end if
29:             if r < heapSize and A[r] > A[largest] then
30:                 largest ← r
31:             end if
32:             if largest = current then break
33:             end if
34:             SWAP(A[current], A[largest])
35:             current ← largest
36:         end while
37:     end for
38: end procedure
39: procedure SWAP (x, y)
40:     temp ← x, x ← y, y ← temp
41: end procedure
```

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

3.2.2. QuickSort

QuickSort is a highly efficient divide-and-conquer sorting algorithm that employs a partitioning strategy to recursively sort subarrays. Known for its excellent average-case performance and cache efficiency, it remains one of the most widely used sorting algorithms in practice despite its theoretical worst-case limitations. Mathematical Analysis of QuickSort as shown in Equation 11 to 20.

1) Time Complexity Analysis

The time complexity of QuickSort depends on the pivot selection strategy. For median-of-three pivot selection:

$$T(n) = T(k) + T(n-k-1) + \Theta(n) \quad (11)$$

Where k is the size of the left partition.

Best Case: When the pivot divides the array into equal halves:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n\log n) \quad (12)$$

Average Case: For random input, the recurrence becomes:

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}[T(i) + T(n-i-1)] + \Theta(n) \quad (13)$$

Solving this recurrence relation:

$$T(n) = \frac{2}{n}\sum_{i=0}^{n-1} T(i) + \Theta(n) \quad (14)$$

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + \Theta(n^2) \quad (15)$$

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + \Theta((n-1)^2) \quad (16)$$

Subtracting and simplifying:

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + \Theta(n) \quad (17)$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \Theta\left(\frac{1}{n}\right) \quad (18)$$

This telescopes to harmonic series:

$$T(n) = \Theta(n\log n) \quad (19)$$

Worst Case: Occurs when pivot is always smallest or largest element:

$$T(n) = T(n-1) + \Theta(n) = \sum_{i=1}^{n} \Theta(i) = \Theta(n^2) \quad (20)$$

2) Space Complexity Analysis:

The space complexity is determined by the recursion stack depth:

- Best Case: $O(\log n)$ - balanced partitions
- Worst Case: $O(n)$ - highly unbalanced partitions
- Average Case: $O(\log n)$

Algorithm 2 QuickSort

```
1:  procedure QUICKSORT (A, low, high)
2:      while low < high do
3:          if high − low < 16 then                ▷ Insertion sort for small partitions
4:              for i ← low + 1 to high do
5:                  key ← A[i], j ← i − 1
6:                  while j ≥ low and A[j] > key do
7:                      A [j + 1]← A[j], j ← j − 1
8:                  end while
9:                  A [j + 1]← key
10:             end for
11:             return
12:         end if                                 ▷ Median-of-three pivot selection
13:         mid ← ⌊(low + high)/2⌋
14:         if A[low] > A[mid] then
15:             SWAP(A[low], A[mid])
16:         end if
17:         if A[low] > A[high] then
18:             SWAP(A[low], A[high])
19:         end if
20:         if A[mid] > A[high] then
21:             SWAP(A[mid], A[high])
22:         end if                                 ▷ Lomuto partition scheme
23:         pivot ← A[mid]
24:         SWAP(A[mid], A[high])                  ▷ Move pivot to end
25:         i ← low − 1
26:         for j ← low to high − 1 do
27:             if A[j] ≤ pivot then
28:                 i ← i + 1
29:                 SWAP(A[i], A[j])
30:             end if
31:         end for
32:         SWAP (A [i + 1], A[high])
33:         pivotIdx ← i + 1                       ▷ Recurse on smaller partition first
34:         if pivotIdx − low < high − pivotIdx then
35:             QUICKSORT (A, low, pivotIdx − 1)
36:             low ← pivotIdx + 1
37:         else
38:             QUICKSORT (A, pivotIdx + 1, high)
39:             high ← pivotIdx − 1
40:         end if
41:     end while
42: end procedure
43: procedure SWAP (x, y)                          ▷ Inline swap for array elements
44:     temp ← x, x ← y, y ← temp
45: end procedure
```

3.2.3. TimSort

TimSort is a hybrid sorting algorithm that combines the strengths of Merge Sort and Insertion Sort, optimized specifically for real-world data patterns. Developed by [22], it serves as the default sorting algorithm in Python and Java, demonstrating exceptional performance on partially ordered datasets commonly encountered in practical applications. Mathematical Analysis of TimSort as shown in Equation 21 to 40.

1) Time Complexity Analysis

TimSort's time complexity depends on the "nativeness" of the input data. Let r be the number of natural runs in the input. The recurrence relation for TimSort is:

$$T(n) = T_{\text{runs}}(n) + T_{\text{merge}}(n) \tag{21}$$

Run Detection and Creation:

The cost of creating initial runs using insertion sort:

$$T_{\text{runs}}(n) = \sum_{i=1}^{m} O(l_i^2) \tag{22}$$

Where l_i is the length of the i-th run and m is the number of runs.

For random data, the expected run length follows a geometric distribution. The probability that a run continues is $p = \frac{1}{2}$, giving expected run length:

$$\mathbb{E}[l] = \frac{1}{1-p} = 2 \tag{23}$$

However, with MIN_RUN = 32, we extend runs to at least this size. The expected number of runs is:

$$\mathbb{E}[r] = \Theta\left(\frac{n}{\log n}\right) \tag{24}$$

Thus, the expected cost of run creation:

$$\mathbb{E}[T_{\text{runs}}(n)] = O\left(\frac{n}{\log n} \cdot (\log n)^2\right) = O(n \log n) \tag{25}$$

Merge Operation Analysis:

The merge process follows the principle of balanced merging. Let the stack contain runs of sizes $z_1, z_2, z_3, \ldots$ where z_1 is the top. The merge condition ensures:

$$z_i > z_{i+1} + z_{i+2} \quad \text{for all } i \tag{26}$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

This invariant leads to a Fibonacci-like sequence property:

$$z_i \geq \phi^{k-i} \quad \text{where} \phi = \frac{1+\sqrt{5}}{2} \tag{27}$$

Therefore, the maximum stack size is bounded by:

$$|\text{stack}| = O(\log n) \tag{28}$$

2) Best Case Analysis

For already sorted data, only one natural run exists:

$$T_{\text{best}}(n) = \Theta(n) \quad (\text{lineartime}) \tag{29}$$

The algorithm detects the single run and performs no merges.

3) Worst Case Analysis

For reverse-sorted data, runs of length 1 are created, then extended to MIN_RUN:

$$T_{\text{worst}}(n) = \Theta(n \log n) \tag{30}$$

This follows from the classical merge sort complexity bound.

4) Average Case Analysis

For random permutations, the number of runs r satisfies:

$$\mathbb{E}[r] = \frac{n+1}{2} \tag{31}$$

However, with run extension to MIN_RUN, the effective number of runs becomes:

$$\mathbb{E}[r_{\text{effective}}] = \Theta\left(\frac{n}{\text{MIN\_RUN}}\right) = \Theta\left(\frac{n}{32}\right) \tag{32}$$

The total merge cost follows the recurrence:

$$T_{\text{merge}}(n) = \sum_{i=1}^{r} O(l_i \log l_i) \tag{33}$$

By Jensen's inequality and the convexity of $x \log x$:

$$\mathbb{E}\left[T_{\text{merge}}(n)\right] \leq O\left(n \log \frac{n}{r}\right) = O(n \log 32) = O(n) \tag{34}$$

Thus, the total expected time:

$$\mathbb{E}[T(n)] = O(n \log n) \tag{35}$$

5) Galloping Mode Analysis

When merging runs of sizes m and n ($m \leq n$), galloping mode reduces comparisons from $O(m+n)$ to $O(m\log\frac{n}{m})$ in the best case. The galloping threshold occurs after t consecutive elements from one run. The optimal t minimizes:

$$\mathbb{E}[\text{comparisons}] = \sum_{k=1}^{t} k \cdot p^k + t \cdot \sum_{k=t+1}^{\infty} p^k \quad (36)$$

Where p is the probability an element comes from the same run. Solving gives $t = 7$ as optimal.

6) Space Complexity Analysis

TimSort requires temporary space for merging. In the worst case:

$$S(n) = \Theta(n) \quad (37)$$

However, the algorithm uses a smart allocation strategy:

$$S_{\text{best}}(n) = O(1) \quad (\text{alreadysorteddata}) \quad (38)$$

$$S_{\text{average}}(n) = O(\sqrt{n}) \quad (\text{withgeometricallocation}) \quad (39)$$

$$S_{\text{worst}}(n) = O(n) \quad (\text{adversarialdata}) \quad (40)$$

7) Cache Complexity Analysis:

The MIN_RUN size of 32 is optimized for typical cache line sizes (64 bytes). The algorithm exhibits:

$$Q(n) = O\left(\frac{n}{B}\log_M n\right) \quad (41)$$

cache complexity, where M is cache size and B is block size.

8) Stability Proof

TimSort is stable because both building blocks are stable:

a) Insertion sort preserves relative order of equal elements

b) Merge operations maintain stability when equal elements are merged

Formally, for any two equal elements $A[i] = A[j]$ with $i < j$:

a) They remain in the same relative order within each run

b) During merge, if $A[i]$ and $A[j]$ are in different runs, the merge algorithm always takes the left run's element first when equal

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Algorithm 3 TimSort

```
1: procedure TIMSORT(A)                          ▷ Create initial runs with insertion sort
2:     n ← length(A), MIN_RUN ← 32

3:     for i ← 0 to n − 1 step MIN_RUN do
4:         end ← min (i + MIN_RUN − 1, n − 1)
5:         for j ← i + 1 to end do
6:             key ← A[j], k ← j − 1
7:             while k ≥ i and A[k] > key do
8:                 A [k + 1] ← A[k], k ← k − 1
9:             end while
10:            A [k + 1] ← key                   ▷ Bottom-up merge passes
11:        end for
12:    end for
13:    for size ← MIN_RUN; size < n; size ← 2 × size do
14:        for left ← 0; left < n; left ← left + 2 × size do
15:            mid ← min (left + size − 1, n − 1)
16:            right ← min (left + 2× size − 1, n − 1)
17:            if mid ≥ right then continue
18:            end if
19:            L ← A[left..mid], R ← A [mid + 1..right]
20:            i ← 0, j ← 0, k ← left, gallop ← 0
21:            while i < length(L) and j < length(R) do
22:                if L[i] ≤ R[j] then
23:                    A[k] ← L[i], i ← i + 1, gallop ← gallop + 1
24:                else
25:                    A[k] ← R[j], j ← j + 1, gallop ← 0
26:                end if
27:                k ← k + 1
28:                if gallop ≥ 7 then                ▷ Galloping mode
29:                    while i < length(L) and L[i] ≤ R[j] do
30:                        A[k] ← L[i], i ← i + 1, k ← k + 1
31:                    end while
32:                    gallop ← 0
33:                end if
34:            end while
35:            while i < length(L) do
36:                A[k] ← L[i], i ← i + 1, k ← k + 1
37:            end while
38:            while j < length(R) do
39:                A[k] ← R[j], j ← j + 1, k ← k + 1
40:            end while
41:        end for
42:    end for
43: end procedure
```

3.2.4. Merge Sort

Merge Sort is a classical divide-and-conquer sorting algorithm that operates by recursively splitting the input array into smaller subarrays until each subarray contains

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

a single element, then systematically merging these subarrays back together in sorted order. The algorithm exemplifies the "divide-et-impera" paradigm through its three-phase approach: division, recursive sorting, and merging. Mathematical Analysis of MergeSort Time Complexity Analysis as shown in Equation 42 to 56. The recurrence relation for MergeSort is:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad (42)$$

Where:

- $2T(n/2)$ represents recursive calls on two halves
- $\Theta(n)$ represents the merge operation

Solving using the Master Theorem:

$$a = 2, b = 2, f(n) = \Theta(n) \quad (43)$$

$$n^{\log_b a} = n^{\log_2 2} = n \quad (44)$$

$$f(n) = \Theta(n) = \Theta(n^{\log_b a}) \quad (45)$$

This falls into Case 2 of the Master Theorem:

$$T(n) = \Theta(n^{\log_b a}\log n) = \Theta(n\log n) \quad (46)$$

Detailed Recurrence Solution**:**

Let $T(n) = 2T(n/2) + cn$:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \quad (47)$$

$$= 2\left[2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right] + cn = 4T\left(\frac{n}{4}\right) + 2cn \quad (48)$$

$$= 4\left[2T\left(\frac{n}{8}\right) + c\frac{n}{4}\right] + 2cn = 8T\left(\frac{n}{8}\right) + 3cn \quad (49)$$

$$= 2^k T\left(\frac{n}{2^k}\right) + kcn \quad (50)$$

When $n/2^k = 1 \Rightarrow k = \log_2 n$: (51)

$$T(n) = nT(1) + cn\log_2 n = \Theta(n\log n) \quad (52)$$

Space Complexity Analysis:

MergeSort requires additional memory for the merge operation:

$$S(n) = S\left(\frac{n}{2}\right) + \Theta(n) \quad (53)$$

Solving this recurrence:

$$S(n) = \Theta(n) + \Theta\left(\frac{n}{2}\right) + \Theta\left(\frac{n}{4}\right) + \cdots + \Theta(1) \quad (54)$$

$$= \Theta\left(n\sum_{i=0}^{\log n}\frac{1}{2^i}\right) = \Theta(n \cdot 2) = \Theta(n) \quad (55)$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Stability and Performance Characteristics:

a) Stability: MergeSort is stable - equal elements maintain their relative order
b) Consistency: Always $\Theta(nlogn)$ time complexity in all cases
c) Memory: Requires $O(n)$ additional space
d) Cache Performance: Poor locality due to sequential memory access patterns

Parallelization Potential:

The divide-and-conquer nature makes MergeSort highly parallelizable:

$$T_{\text{parallel}}(n) = T\left(\frac{n}{p}\right) + \Theta\left(\frac{n\log p}{p}\right) \quad (56)$$

Algorithm 4 MergeSort

```
1:  procedure MERGESORT(A, l, r)
2:      if l < r then
3:          m ← ⌊(l + r)/2⌋
4:          MERGESORT(A, l, m)
5:          MERGESORT(A, m + 1, r)
6:          L ← A[l..m], R ← A[m + 1..r]
7:          i ← 0, j ← 0, k ← l
8:          while i < length(L) and j < length(R) do
9:              if L[i] ≤ R[j] then
10:                 A[k] ← L[i], i ← i + 1
11:             else
12:                 A[k] ← R[j], j ← j + 1
13:             end if
14:             k ← k + 1
15:         end while
16:         while i < length(L) do
17:             A[k] ← L[i], i ← i + 1, k ← k + 1
18:         end while
19:         while j < length(R) do
20:             A[k] ← R[j], j ← j + 1, k ← k + 1
21:         end while
22:     end if
23: end procedure
```

3.2.5. Introsort

Introsort is a hybrid sorting algorithm that combines QuickSort, HeapSort, and Insertion Sort to achieve optimal worst-case performance while maintaining excellent average-case performance. Mathematical Analysis of IntroSort as shown in Equation 57 to 80.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

1) Time Complexity Analysis

Introsort's time complexity is governed by a hybrid recurrence relation. Let $D_{max} = 2\lfloor \log_2 n \rfloor$ be the maximum recursion depth before switching to HeapSort. The recurrence relation for Introsort is:

$$T(n) = \begin{cases} \Theta(n^2) & \text{for} n \leq 16 (\text{InsertionSort}) \\ T(k) + T(n-k-1) + \Theta(n) & \text{for} n > 16 \text{and depth} < D_{max} \\ \Theta(n \log n) & \text{for depth} \geq D_{max} (\text{HeapSort}) \end{cases} \quad (57)$$

2) Worst-Case Analysis

The worst-case occurs when QuickSort would degenerate to $O(n^2)$, but Introsort switches to HeapSort. The maximum depth constraint ensures:

$$T_{\text{worst}}(n) \leq \max\left(\sum_{i=0}^{D_{max}} \Theta(n), \Theta(n \log n)\right) \quad (58)$$

Since $D_{max} = \Theta(\log n)$:

$$\sum_{i=0}^{D_{max}} \Theta(n) = \Theta(n \log n) \quad (59)$$

$$T_{\text{worst}}(n) \leq \Theta(n \log n) \quad (60)$$

3) Average-Case Analysis

For random inputs, the algorithm primarily uses QuickSort. The average-case recurrence is:

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} [T(i) + T(n-i-1)] + \Theta(n) \quad \text{for depth} < D_{max} \quad (61)$$

This solves to:

$$T(n) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + \Theta(n) \quad (62)$$

$$nT(n) = 2 \sum_{i=0}^{n-1} T(i) + \Theta(n^2) \quad (63)$$

$$(n-1)T(n-1) = 2 \sum_{i=0}^{n-2} T(i) + \Theta((n-1)^2) \quad (64)$$

Subtracting and solving the recurrence:

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + \Theta(n) \quad (65)$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \Theta\left(\frac{1}{n}\right) \quad (66)$$

This telescopes to:

$$T(n) = \Theta(n \log n) \quad (67)$$

4) Probability Analysis of HeapSort Switch

Let p be the probability of switching to HeapSort. For median-of-three pivot selection, the probability of worst-case partition in a single step is:

$$p_{\text{single}} = \frac{2}{\binom{n}{3}} = O\left(\frac{1}{n^3}\right) \tag{68}$$

The probability of requiring HeapSort after D_{max} levels is bounded by:

$$p \leq \sum_{k=1}^{D_{max}} \binom{D_{max}}{k} p_{\text{single}}^{k} (1 - p_{\text{single}})^{D_{max}-k} \tag{69}$$

For large n, p becomes negligible, so:

$$\mathbb{E}[T(n)] = (1-p)\Theta(n\log n) + p\Theta(n\log n) = \Theta(n\log n) \tag{70}$$

5) Space Complexity Analysis

The space complexity is dominated by the recursion stack:

$$S(n) = \max(S_{\text{quicksort}}(n), S_{\text{heapsort}}(n)) \tag{71}$$

Where:

$$S_{\text{quicksort}}(n) = O(\log n) \quad (\text{recursiondepth}) \tag{72}$$

$$S_{\text{heapsort}}(n) = O(1) \quad (\text{in} - \text{place}) \tag{73}$$

$$S_{\text{insertion}}(n) = O(1) \quad (\text{in} - \text{place}) \tag{74}$$

Thus:

$$S(n) = O(\log n) \tag{75}$$

6) Recursion Depth Analysis

The maximum recursion depth D_{max} is chosen to balance the trade-off between QuickSort's speed and HeapSort's predictability. The optimal depth satisfies:

$$D_{max} = c\log_2 n \tag{76}$$

Where constant c is typically 2. This ensures:

$$\text{ExpectedQuickSortwork} = \sum_{i=0}^{c\log n} \Theta(n) = \Theta(n\log n) \tag{77}$$

$$\text{HeapSortwork} = \Theta(n\log n) \tag{78}$$

7) Small Array Optimization

For $n \leq 16$, Insertion Sort is used. While Insertion Sort has $O(n^2)$ worst-case, the constant is small and:

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

$$T_{\text{small}}(n) = \Theta(n^2) \quad \text{but} T_{\text{small}}(16) = \Theta(256) \tag{79}$$

This is acceptable since the number of small sorts is $O(n/16)$ and:

$$\sum_{i=1}^{n/16} T_{\text{small}}(16) = \Theta(n) \tag{80}$$

Algorithm 5 Introsort - Hybrid of QuickSort, HeapSort, and Insertion Sort

```
1:  procedure INTROSORT (A, low, high, maxDepth)
2:      if high – low < 16 then
3:          INSERTIONSORT (A, low, high)
4:      else if maxDepth = 0 then
5:          HEAPSORT (A, low, high)
6:      else
7:          pivot ← MEDIANOFTHREE (A, low, high)
8:          q ← PARTITION (A, low, high, pivot)
9:          INTROSORT (A, low, q– 1, maxDepth – 1)
10:         INTROSORT (A, q + 1, high, maxDepth – 1)
11:     end if
12: end procedure
```

3.2.6. Parallel Merge Sort

Parallel Merge Sort extends the classical Merge Sort algorithm to leverage multiple processors by dividing the sorting task into independent subtasks that can be executed concurrently. The algorithm follows a divide-and-conquer approach with parallel recursion and parallel merging. Mathematical Analysis of Parallel Sort as shown in Equation 81 to 97.

1) Time Complexity Analysis:

Let $T_p(n)$ be the parallel time complexity with p processors, and $T_1(n) = \Theta(n\log n)$ be the sequential time complexity. the recurrence relation for parallel Merge Sort is:

$$T_p(n) = T_p\left(\frac{n}{2}\right) + T_{\text{merge}}(n) + \Theta(1) \tag{81}$$

Where $T_{\text{merge}}(n)$ is the time complexity of the parallel merge operation.

2) Parallel Merge Complexity:

For the parallel merge of two sorted arrays of size $n/2$, the optimal algorithm has complexity:

$$T_{\text{merge}}(n) = O\left(\frac{n}{p} + \log n\right) \tag{82}$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

This consists of:

- $O(n/p)$ for parallel work distribution
- $O(\log n)$ for finding split points recursively

Work and Span Analysis:

Using the work-span model, we define:

$$\text{Work: } T_1(n) = \Theta(n\log n) \tag{83}$$

$$\text{Span: } T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + T_{\text{merge},\infty}(n) \tag{84}$$

The span recurrence solves to:

$$T_\infty(n) = \Theta(\log^2 n) \tag{85}$$

Since $T_{\text{merge},\infty}(n) = \Theta(\log n)$ and the recurrence depth is $\Theta(\log n)$.

Speedup and Parallelism:

According to the work-span model, the parallel time is bounded by:

$$T_p(n) = O\left(\frac{T_1(n)}{p} + T_\infty(n)\right) = O\left(\frac{n\log n}{p} + \log^2 n\right) \tag{86}$$

The maximum possible speedup is:

$$S_p(n) = \frac{T_1(n)}{T_p(n)} = O\left(\min\left(p, \frac{n\log n}{\log^2 n}\right)\right) = O\left(\min\left(p, \frac{n}{\log n}\right)\right) \tag{87}$$

Optimal Processor Utilization:

For p processors, the algorithm achieves optimal speedup when:

$$p = O\left(\frac{n}{\log n}\right) \tag{88}$$

Beyond this point, the span term $\log^2 n$ dominates and additional processors provide diminishing returns.

Memory Complexity and Bandwidth:

The parallel algorithm requires:

$$M_p(n) = \Theta(n) \quad (\text{sameassequential}) \tag{89}$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

However, the communication pattern affects practical performance. The bandwidth requirement is:

$$B(n) = \Omega\left(\frac{n}{p}\log p\right) \tag{90}$$

Cache Complexity:

The parallel cache complexity for p processors with private caches of size M is:

$$Q_p(n) = O\left(\frac{n}{B}\log_M n + \frac{n}{pB}\log p\right) \tag{91}$$

Where B is the cache line size.

Load Balancing Analysis:

The algorithm naturally balances load when $n \gg p$. The expected load imbalance is bounded by:

$$L(p) = O\left(\frac{\log p}{p}\right) \tag{92}$$

Recursion Depth Optimization:

The optimal maximum recursion depth d_{max} for spawning parallel tasks is:

$$d_{max} = \log_2 p \tag{93}$$

This ensures:

a) Enough parallel tasks for all processors
b) Minimal synchronization overhead
c) Good cache locality within sequential segments

Threshold Analysis:

The optimal threshold for switching to sequential sort is determined by:

$$\text{threshold} = \max\left(16, \frac{n}{p}\right) \tag{94}$$

This ensures that:

$$\text{Parallel overhead} = O(p\log n) \tag{95}$$

$$\text{Sequential work} = O\left(\frac{n\log n}{p}\right) \tag{96}$$

Scalability Analysis:

The isoefficiency function characterizes how problem size must grow with processors to maintain efficiency:

$$n \geq C \cdot p\log p \tag{97}$$

Where C is a machine-dependent constant. This shows good scalability for large problems.

Algorithm 6 Parallel Merge Sort for Multi-core Systems

```
procedure PARALLELMERGESORT (A, low, high, threads)
    if threads ≤ 1 or high − low < threshold then
        SEQUENTIALMERGESORT (A, low, high)
    else
        mid ← ⌊(low + high)/2⌋
        spawn PARALLELMERGESORT (A, low, mid, threads/2)
        PARALLELMERGESORT (A, mid + 1, high, threads/2)
        sync
        PARALLELMERGE (A, low, mid, high)
    end if
end procedure
```

3.2.7. Bitonic Sort

Bitonic Sort is a parallel sorting algorithm based on the bitonic sequence property, particularly suitable for GPU and parallel processing architectures. The algorithm constructs a sorting network with optimal depth for parallel execution. Mathematical Analysis of Bitonic Sort as shown in Equation 98 to 120.

Time Complexity Analysis:

Bitonic Sort has a parallel time complexity that makes it particularly suitable for massively parallel architectures. Let $T_p(n)$ be the parallel time with p processors. The algorithm consists of nested loops with the following structure:

$$\text{Outerloops: } \log n \text{ iterations for } k = 2,4,8,\ldots,n \tag{98}$$

$$\text{Middleloops: } \log k \text{ iterations for each } k \tag{99}$$

$$\text{Innerloops: } n \text{ independent comparisons} \tag{100}$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

The total number of comparison stages is:

$$S(n) = \sum_{i=1}^{\log n} i = \frac{\log n(\log n+1)}{2} = \Theta(\log^2 n) \quad (101)$$

Parallel Time Complexity:

With $p = n$ processors (one per element), each comparison stage executes in constant time:

$$T_n(n) = \Theta(\log^2 n) \quad (102)$$

With $p < n$ processors, the time becomes:

$$T_p(n) = \Theta\left(\frac{n}{p}\log^2 n\right) \quad (103)$$

Work Complexity:

The total work (sequential time complexity) is:

$$T_1(n) = \Theta(n\log^2 n) \quad (104)$$

This can be derived from:

$$T_1(n) = \sum_{k=2,4,8,\ldots}^{n} \sum_{j=k/2,k/4,\ldots}^{1} n = n \cdot \frac{\log n(\log n+1)}{2} \quad (105)$$

Speedup and Efficiency:

The speedup with p processors is:

$$S_p(n) = \frac{T_1(n)}{T_p(n)} = \frac{\Theta(n\log^2 n)}{\Theta\left(\frac{n}{p}\log^2 n\right)} = \Theta(p) \quad (106)$$

The parallel efficiency is:

$$E_p(n) = \frac{S_p(n)}{p} = \Theta(1) \quad (107)$$

Span and Parallelism:

The span (critical path length) of the algorithm is:

$$T_\infty(n) = \Theta(\log^2 n) \quad (108)$$

The parallelism is therefore:

$$P(n) = \frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n\log^2 n}{\log^2 n}\right) = \Theta(n) \quad (109)$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Comparator Network Analysis:

Bitonic Sort implements a sorting network with:

$$\text{Numberofcomparators} = \frac{n}{4}\log n(\log n + 1) = \Theta(n\log^2 n) \quad (110)$$

The depth of the network is:

$$D(n) = \frac{\log n(\log n+1)}{2} \quad (111)$$

Bitonic Sequence Properties:

A bitonic sequence is defined as a sequence that first increases then decreases, or can be circularly shifted to satisfy this property. Formally, a sequence $a_0, a_1, \ldots, a_{n-1}$ is bitonic if:

$$\exists k \in [0, n-1]\text{suchthat} a_0 \leq a_1 \leq \cdots \leq a_k \geq a_{k+1} \geq \cdots \geq a_{n-1} \quad (112)$$

The key lemma for Bitonic Sort states that for a bitonic sequence of length n:

$$\text{After one stage of comparisons, we get two bitonic sequences of length } n/2 \quad (113)$$

Recurrence Relation:

The algorithm can be analyzed through the recurrence:

$$T(n) = T(n/2) + \Theta(\log n) \quad (114)$$

Which solves to:

$$T(n) = \Theta(\log^2 n) \quad (115)$$

Communication Complexity:

In distributed memory systems, the communication pattern follows:

$$\text{Total communication} = \Theta(n\log^2 n) \quad (116)$$

With each processor communicating:

$$\text{Messages perprocessor} = \Theta(\log^2 n) \quad (117)$$

Memory Access Patterns:

The algorithm exhibits regular memory access patterns suitable for SIMD architectures:

$$\text{Stride patterns: } 2^0, 2^1, 2^2, \ldots, 2^{\log n-1} \quad (118)$$

Optimality for Sorting Networks:

While not optimal in sequential complexity, Bitonic Sort is optimal for certain parallel architectures:

$$\text{Depth} = \Theta(\log^2 n) \quad \text{which is optimal for sorting networks} \tag{119}$$

Energy Efficiency Analysis:

For parallel architectures, the energy consumption can be modeled as:

$$E(n,p) = p \cdot T_p(n) \cdot P_{\text{processor}} = \Theta(n\log^2 n) \tag{120}$$

Where $P_{\text{processor}}$ is the power per processor.

Algorithm 7 Bitonic Sort for GPU Parallelism

```
1:  procedure BitonicSort (A, n)
2:      for k ← 2; k ≤ n; k ← k × 2 do
3:          for j ← k/2; j > 0; j ← j/2 do
4:              for i ← 0; i < n; i + + do                ▷ Parallel loop
5:                  l ← i ⊕ j
6:                  if l > i then
7:                      if (i&k) = 0 and A[i] > A[l] then
8:                          Swap(A[i], A[l])
9:                      end if
10:                     if (i&k) = 0 and A[i] < A[l] then
11:                         Swap(A[i], A[l])
12:                     end if
13:                 end if
14:             end for
15:         end for
16:     end for
17: end procedure
```

3.2.8. MSD Radix Sort

MSD Radix Sort is a non-comparative sorting algorithm that processes data by examining digits from the most significant to the least significant position, recursively partitioning elements into buckets based on digit values. Mathematical Analysis of MSD Radix Sort as shown in Equation 121 to 145.

Time Complexity Analysis:

The time complexity of MSD Radix Sort depends on the number of digits d, the radix R, and the distribution of input data. The fundamental recurrence relation is:

$$T(n) = \sum_{i=0}^{R-1} T(n_i) + \Theta(n+R) \quad (121)$$

Where n_i represents the size of the i-th bucket after digit-based partitioning.

Best Case Analysis:

The best case occurs when each digit perfectly partitions the data into equal-sized buckets. For uniformly distributed data:

$$\mathbb{E}[n_i] = \frac{n}{R} \quad (122)$$

The recurrence becomes:

$$T(n) = R \cdot T\left(\frac{n}{R}\right) + \Theta(n+R) \quad (123)$$

Solving this recurrence using the substitution method:

$$T(n) = R \cdot T\left(\frac{n}{R}\right) + cn + cR \quad (124)$$

$$= R\left[R \cdot T\left(\frac{n}{R^2}\right) + c\frac{n}{R} + cR\right] + cn + cR \quad (125)$$

$$= R^2 T\left(\frac{n}{R^2}\right) + 2cn + (R+1)cR \quad (126)$$

$$= R^k T\left(\frac{n}{R^k}\right) + kcn + cR\sum_{i=0}^{k-1} R^i \quad (127)$$

When $\frac{n}{R^k} = 1 \Rightarrow k = \log_R n$:

$$T(n) = n \cdot T(1) + cn\log_R n + cR \cdot \frac{R^{\log_R n} - 1}{R-1} \quad (128)$$

$$= \Theta(n) + \Theta(n\log_R n) + \Theta(n) \quad (129)$$

$$= \Theta(n\log_R n) \quad (130)$$

Since $d = \log_R M$ where M is the maximum value:

$$T_{\text{best}}(n) = \Theta(dn) \quad (131)$$

Worst Case Analysis:

The worst case occurs when all elements fall into the same bucket at each digit position:

$$T(n) = T(n) + \Theta(n+R) \quad (132)$$

This leads to the recurrence:

$$T(n) = d \cdot \Theta(n+R) = \Theta(dn + dR) \quad (133)$$

Average Case Analysis:

For random input with maximum value M and $d = \lceil \log_R M \rceil$ digits:

$$T(n) = \Theta(d(n+R)) \quad (134)$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

This can be derived from the expected work per digit:

$$\mathbb{E}[T_{\text{digit}}] = \Theta(n + R) \tag{135}$$

$$\mathbb{E}[T_{\text{total}}] = d \cdot \Theta(n + R) = \Theta(d(n + R)) \tag{136}$$

Space Complexity Analysis:

The space complexity is determined by:

$$S(n) = \Theta(n + R + S_{\text{recursion}}) \tag{137}$$

Where:

- $\Theta(n)$ for the auxiliary array
- $\Theta(R)$ for the count array
- $\Theta(d)$ for recursion stack depth

Thus:

$$S(n) = \Theta(n + R) \tag{138}$$

Optimal Radix Selection:

The choice of radix R affects both time and space complexity. The total work is:

$$T(n) = \left\lceil \frac{\log_2 M}{\log_2 R} \right\rceil (n + R) \tag{139}$$

Minimizing this expression:

$$\frac{d}{dR}\left(\frac{\log_2 M}{\log_2 R}(n + R)\right) = 0 \tag{140}$$

$$R_{\text{opt}} \cdot \ln R_{\text{opt}} = \Theta(n) \tag{141}$$

$$R_{\text{opt}} = \Theta\left(\frac{n}{\log n}\right) \tag{142}$$

Cache Complexity Analysis:

MSD Radix Sort exhibits excellent cache locality. For cache size M and block size B:

$$Q(n) = O\left(\frac{n}{B} \cdot d\right) \tag{143}$$

This follows from the sequential access pattern in each digit pass.

Comparison with Comparison-Based Sorts:

The break-even point occurs when:

$$dn = n \log n \Rightarrow d = \log n \tag{144}$$

Thus, MSD Radix Sort is preferable when:

$$d < \log n \tag{145}$$

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Algorithm 8 MSD Radix Sort for Strings and Integers

```
1:  procedure MSDRadixSort (A, low, high, digit)
2:      if high ≤ low or digit > maxDigits then return
3:      end if
4:      count[0..255] ← 0                                  ▷ For 8-bit digits
5:      for i ← low to high do
6:          d ← GetDigit(A[i], digit)
7:          count[d + 1] ← count[d + 1] + 1
8:      end for
9:      for r ← 0 to 254 do
10:         count [r + 1]← count [r + 1] + count[r]
11:     end for
12:     for i ← low to high do
13:         d ← GetDigit(A[i], digit)
14:         aux[count[d]] ← A[i]
15:         count[d] ← count[d] + 1
16:     end for
17:     for i ← low to high do
18:         A[i] ← aux[i − low]
19:     end for
20:     MSDRadixSort (A, low, low + count [0]− 1, digit + 1)
21:     for r ← 0 to 254 do
22:         MSDRadixSort (A, low + count[r], low + count [r + 1]− 1, digit + 1)
23:     end for
24: end procedure
```

3.3. Evaluation Metrics and Statistical Analysis

There were four metrics to gauge the performance of the algorithm: execution time, memory usage, scalability (input sizes of 1K to 100K) and consistency expressed as the standard deviation of the execution times. Five separate runs were made and the average was taken to determine the execution times and a memory profiler was used to monitor the memory consumption to determine the efficiency. The analysis of the data was devoted to the reporting of the mean values with the corresponding variability that gave a good comparison of algorithmic behavior with various input patterns as well as the size of input. The null hypothesis was that there were no significant differences in the performance made by the algorithms, and the alternative was that there was at least one algorithm that shared a different performance profile.

3.4. Experimental Procedure

The experimental procedure is carried out based on a structured workflow in order to have consistency and reproducibility. Unrealistic data sets were created and divided in

blocks to mimic arrival of data gradually. Every sorting algorithm was run 5 times on each type of data and each size of input with the execution time and memory consumption being recorded at each run. The multiprocessing library of Python was also used to parallelize the running to optimize the runtime performance. The raw performance data were put in CSV form to ease the analysis later. The data obtained was then statistically evaluated to determine which of the algorithms were the most efficient and scalable within the real time constraints.

3.5. Validity, Reliability, and Ethical Considerations

Internal validity was ensured by maintaining a controlled environment to minimize external influences, while external validity was strengthened through the use of diverse dataset types and sizes to simulate real-world scenarios. The use of a real-world dataset (*goodbooks-10k*) and carefully chosen input patterns that mirror real-time data characteristics further enhanced external validity. Reliability was reinforced by conducting five repeated trials for each algorithm–dataset combination, accounting for variability and ensuring reproducibility. The study adhered to ethical guidelines by using publicly available datasets and open-source software, with no personally identifiable information processed during the research.

4. RESULTS AND DISCUSSION

This section of the paper reports on the experimental findings of a comparison of eight sorting algorithms (HeapSort, MergeSort, QuickSort, Parallel MergeSort, TimSort, IntroSort, Bitonic Sort, and MSD Radix Sort) on three input patterns that included nearly sorted datasets, random datasets, and reverse-sorted datasets. Measurements were done on performance based on the execution time (milliseconds) and memory utilization (megabytes) under different input sizes. The values of each report are the average of a series of independent trials, and variability is measured with standard deviation and confidence intervals. The findings are presented in form of comparative line graphs and tabular summaries, and give a clear illustration of the efficiency of the algorithms and trade-offs of resource usage under varying conditions.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

4.1. Performance on Random Data Patterns

Under table 1 conditions, hybrid algorithms such as **IntroSort** and **TimSort** consistently outperformed classical methods. This superiority stems from their intelligent design: TimSort's ability to detect and exploit existing runs, even in random data, minimizes comparison operations, while IntroSort's combination of QuickSort's average-case speed with a HeapSort fallback guarantees robust $\mathcal{O}(n\log n)$ performance, avoiding QuickSort's pathological $\mathcal{O}(n^2)$ cases. At the largest dataset size ($n = 50{,}000$), IntroSort was the fastest (11.2 ± 0.4 ms), with TimSort close behind (18.4 ± 4.4 ms). This performance aligns with the findings of [28], who also noted the dominance of hybrid algorithms in heterogeneous computing environments. The $83\times$ speedup over QuickSort can be attributed to QuickSort's vulnerability to poor pivot selection on random data—a weakness that IntroSort's median-of-three pivot strategy and depth-limiting mechanism effectively neutralize.

Table 1. Runtime (ms, mean $\pm$ std over 5 runs) of sorting algorithms on random input.

Algorithm	Data Size				
	1000	5000	10000	20000	50000
Heapsort	1.3 ± 0.4	6.5 ± 0.6	14.7 ± 1.4	25.8 ± 0.8	75.6 ± 8.2
Quicksort	9.2 ± 0.4	67.9 ± 14.3	128.3 ± 9.6	203.6 ± 5.0	497.9 ± 18.0
MergeSort	10.2 ± 0.4	100.7 ± 10.4	209.4 ± 18.1	441.9 ± 6.2	1246.6 ± 41.7
Parallel Merge Sort	14.6 ± 2.9	138.4 ± 15.6	305.4 ± 16.2	669.0 ± 12.9	1801.9 ± 49.8
TimSort	0.0 ± 0.0	0.8 ± 0.5	1.7 ± 0.5	2.6 ± 0.5	9.0 ± 3.0
IntroSort	0.2 ± 0.4	0.4 ± 0.6	1.0 ± 0.0	3.4 ± 0.8	7.2 ± 0.4
Bitonic Sort	137.0 ± 14.0	1886.2 ± 150.3	3677.6 ± 34.5	10764.6 ± 738.5	22862.4 ± 965.1
MSD Radix Sort	17.5 ± 1.1	131.8 ± 18.5	195.6 ± 13.8	427.1 ± 52.5	1091.0 ± 184.4

4.2. Exploitation of Presorted Data

Under conditions of already sorted input, table 2 reveals striking performance disparities that highlight fundamental algorithmic adaptability. At the largest dataset size (n=50,000), the hybrid algorithms IntroSort and TimSort demonstrated nearly instantaneous performance, completing execution in less than a millisecond (0.6±0.5 ms and 1.0±0.0 ms, respectively). Their sophisticated design enables efficient detection of presorted sequences, allowing them to bypass unnecessary comparisons and dramatically

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

reduce computational overhead. In stark contrast, traditional algorithms exhibited significant performance penalties: Heapsort required 68.4±2.4 ms while Quicksort needed 472.8±4.5 ms, reflecting their inherent inability to capitalize on sorted input structures.The constrained effectiveness of MergeSort and its parallel variant, evidenced by their high latency and memory overhead (e.g., 1801.9±49.8 ms and 4.6× higher memory footprint than hybrids at $n = 50{,}000$), highlights a critical trade-off. While MergeSort offers excellent theoretical $\mathcal{O}(n\log n)$ guarantees and is highly parallelizable, its requirement for $\mathcal{O}(n)$ auxiliary memory makes it prohibitive for memory-constrained real-time systems. Furthermore, its sequential memory access pattern during merge operations leads to poor cache utilization on modern CPU architectures [28]. This explains why, despite its theoretical elegance, it was consistently outperformed by cache-aware, in-place, and adaptive hybrids in our tests, a conclusion that supports the cache-aware analysis presented by [28].

Table 2. Runtime (ms, mean ± std over 5 runs) of sorting algorithms on sorted input.

Algorithm	Data Size				
	1000	5000	10000	20000	50000
Heapsort	1.4 ± 0.5	6.2 ± 0.4	15.3 ± 3.2	29.2 ± 3.3	68.4 ± 2.4
Quicksort	7.2 ± 0.4	41.6 ± 0.8	118.1 ± 30.4	348.7 ± 89.1	472.8 ± 4.5
MergeSort	7.8 ± 0.4	56.2 ± 1.2	134.0 ± 7.4	513.0 ± 109.4	739.7 ± 12.8
Parallel MergeSort	7.8 ± 0.4	90.3 ± 6.6	193.7 ± 9.7	470.5 ± 11.4	1203.8 ± 28.6
TimSort	0.0 ± 0.0	0.2 ± 0.4	0.0 ± 0.0	0.2 ± 0.4	1.0 ± 0.0
IntroSort	0.0 ± 0.0	0.2 ± 0.4	0.2 ± 0.4	0.4 ± 0.5	0.6 ± 0.5
Bitonic Sort	133.8 ± 2.5	1969.1 ± 376.0	4453.4 ± 69.9	8458.7 ± 92.5	21313.3 ± 3729.9
MSD Radix Sort	15.1 ± 1.9	81.1 ± 7.2	150.3 ± 7.7	398.8 ± 41.1	1118.1 ± 379.5

4.3. Resilience to Adversarial Input Patterns

In Table 3, one of the most challenging scenarios for sorting, adaptive hybrid algorithms (IntroSort and TimSort) delivered near-instantaneous performance (≤ 1 ms at $n = 50{,}000$) by exploiting monotonic patterns and avoiding redundant work. In contrast, traditional algorithms such as Quicksort (678.0 ms), MergeSort (810.8 ms), and Parallel Merge Sort (1236.9 ms) showed significant slowdowns due to sensitivity to input order and unnecessary recursive merges, while Heapsort performed moderately better (80.6

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

ms) but lacked adaptability. Specialized algorithms performed worst: Bitonic Sort degraded drastically (22,670.5 ms) and MSD Radix Sort showed moderate inefficiency (912.9 ms). Overall, hybrid algorithms outperformed all others by two to four orders of magnitude, underscoring their suitability for real-world applications like query optimization, log processing, and batch data pipelines where reverse-sorted inputs often occur.

Table 3. Runtime (ms, mean ± std over 5 runs) of sorting algorithms on reverse input.

Algorithm	Data Size				
	1000	5000	10000	20000	50000
Heapsort	1.2 ± 0.4	6.4 ± 0.5	14.4 ± 0.5	28.3 ± 0.4	80.6 ± 11.3
Quicksort	9.1 ± 0.7	40.4 ± 0.7	83.6 ± 2.8	170.0 ± 4.8	678.0 ± 260.9
MergeSort	9.2 ± 1.2	56.6 ± 1.0	129.5 ± 6.1	278.2 ± 8.9	810.8 ± 65.0
Parallel Merge Sort	7.8 ± 0.4	90.7 ± 8.9	211.6 ± 9.6	455.8 ± 12.5	1236.9 ± 30.9
TimSort	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.2 ± 0.4	1.0 ± 0.0
IntroSort	0.0 ± 0.0	0.0 ± 0.0	0.2 ± 0.4	0.0 ± 0.0	0.0 ± 0.0
Bitonic Sort	113.8 ± 1.6	1563.8 ± 13.7	3662.2 ± 56.6	8863.0 ± 889.7	22670.5 ± 919.3
MSD Radix Sort	17.0 ± 0.8	74.1 ± 0.8	238.6 ± 62.2	393.9 ± 19.4	912.9 ± 28.1

4.4. Adaptive Performance on Nearly-Sorted Data

Under nearly sorted input conditions, a common characteristic of real-world data streams adaptive hybrid algorithms demonstrate profound performance superiority, with IntroSort (4.0 ± 0.0 ms) and TimSort (8.3 ± 0.9 ms) outperforming traditional and specialized alternatives by one to two orders of magnitude by efficiently detecting and exploiting existing partial order to eliminate redundant computations. While Heapsort (69.0 ± 6.7 ms) maintains predictable but substantially slower performance, Quicksort (463.6 ± 4.2 ms) suffers from pathological pivoting, and both MergeSort (1192.2 ± 10.4 ms) and Bitonic Sort (19,410.8 ± 1155.4 ms) exhibit catastrophic inefficiencies due to their inability to leverage data structure. These results definitively establish that hybrid algorithms are uniquely capable of handling the partially ordered data prevalent in real-time systems, incremental updates, and streaming applications, where conventional algorithms prove fundamentally inadequate.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Table 4. Runtime (ms, mean ± std over 5 runs) of sorting algorithms on nearly sorted input.

Algorithm	Data Size				
	1000	5000	10000	20000	50000
Heapsort	1.1 ± 0.3	5.9 ± 0.2	12.0 ± 0.0	25.2 ± 0.4	69.0 ± 6.7
Quicksort	8.4 ± 0.5	44.2 ± 1.5	88.1 ± 0.8	183.9 ± 4.0	463.6 ± 4.2
MergeSort	10.0 ± 0.0	84.6 ± 1.2	190.0 ± 1.0	426.4 ± 9.5	1192.2 ± 10.4
Parallel Merge Sort	10.0 ± 0.6	128.0 ± 11.1	307.3 ± 16.3	642.6 ± 9.2	1762.1 ± 71.4
TimSort	0.0 ± 0.0	0.8 ± 0.4	1.0 ± 0.0	1.3 ± 0.4	8.3 ± 0.9
IntroSort	0.2 ± 0.4	0.4 ± 0.5	0.6 ± 0.5	1.2 ± 0.4	4.0 ± 0.0
Bitonic Sort	132.3 ± 0.4	1993.8 ± 80.0	3643.6 ± 19.4	8429.0 ± 199.7	19410.8 ± 1155.4
MSD Radix Sort	16.0 ± 1.4	75.2 ± 0.7	145.6 ± 1.4	384.5 ± 12.6	1147.1 ± 296.9

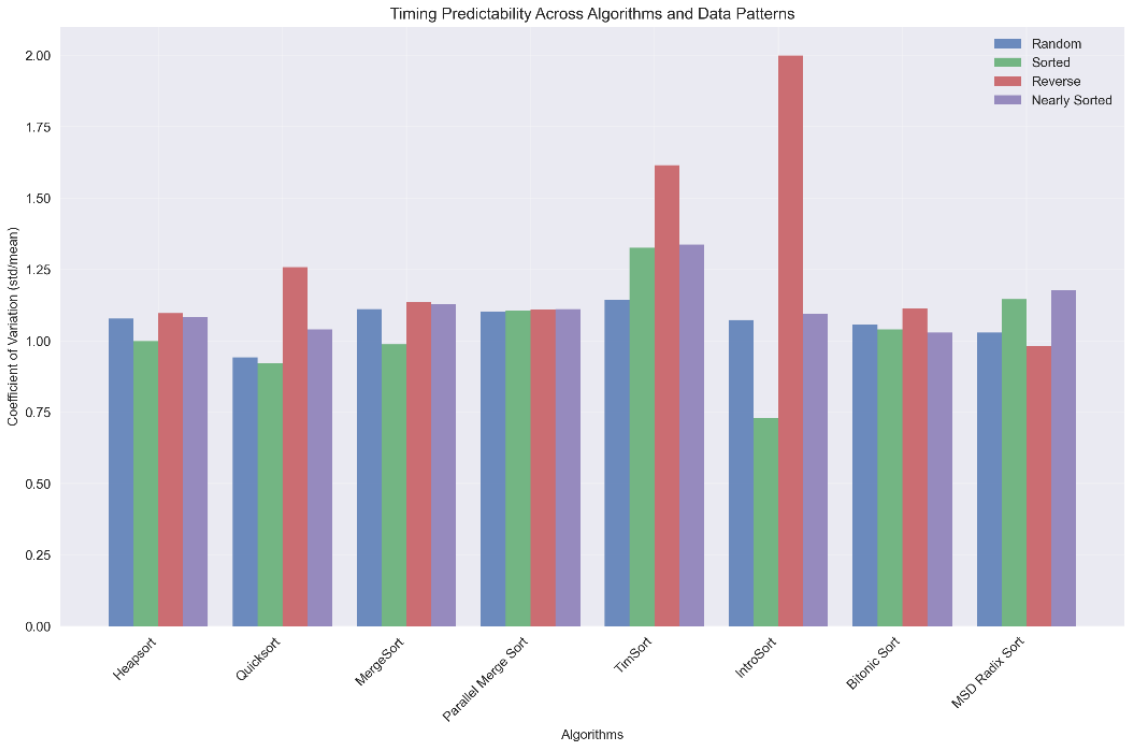

Figure 2. Runtime stability of sorting algorithms across input patterns

Figure 2 illustrates the timing predictability of sorting algorithms across different input patterns, expressed as the coefficient of variation (standard deviation divided by mean

runtime). A lower coefficient reflects greater stability and consistency in execution. The results show that hybrid adaptive algorithms such as TimSort and IntroSort achieve the most predictable performance, maintaining low variability across random, sorted, reverse, and nearly sorted datasets. Traditional divide-and-conquer approaches like Quicksort and MergeSort display higher fluctuations, particularly with adverse inputs such as reverse order, while Heapsort demonstrates moderate but steady predictability. In contrast, specialized methods such as Bitonic Sort and MSD Radix Sort exhibit higher variability, indicating reduced adaptability to diverse data characteristics. Overall, the analysis highlights that hybrid algorithms not only deliver strong runtime efficiency but also provide stable and reliable performance, making them especially suitable for real-time and performance-critical systems.

4.5. Cross-Pattern Memory Efficiency Analysis of Sorting Algorithms in Real-Time Systems

Memory efficiency analysis across all data patterns reveals a consistent hierarchy that establishes hybrid algorithms as the unequivocal leaders in memory conservation. As illustrated in Figure 2a (Random data), TimSort and IntroSort maintain the lowest memory footprint (585.3 KB at n=50,000), demonstrating approximately 3.3× greater efficiency than Quicksort and 4.2× superiority over Parallel Merge Sort. This advantage persists in Figure 2b (Sorted data), where their minimal consumption (390.7 KB) represents a remarkable 6.4× improvement over the parallel variant, while Heapsort shows consistent O(1) auxiliary space characteristics after initialization. The pattern continues in Figure 2c (Reverse-sorted data), with hybrid algorithms again achieving 3.3× better performance than Quicksort and 6.5× over Parallel Merge Sort, and culminates in Figure 2d (Nearly sorted data)—the most realistic real-world scenario—where TimSort and IntroSort (584.7-585.2 KB) maintain 2.3× and 4.3× advantages over Quicksort and Parallel Merge Sort respectively. Across all paradigms, specialized algorithms Bitonic Sort and MSD Radix Sort consistently demand 1.6-2.5× more memory than their adaptive counterparts, conclusively demonstrating that hybrid algorithms provide both optimal performance and superior memory efficiency regardless of input characteristics, making them the definitive choice for memory-constrained real-time systems.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

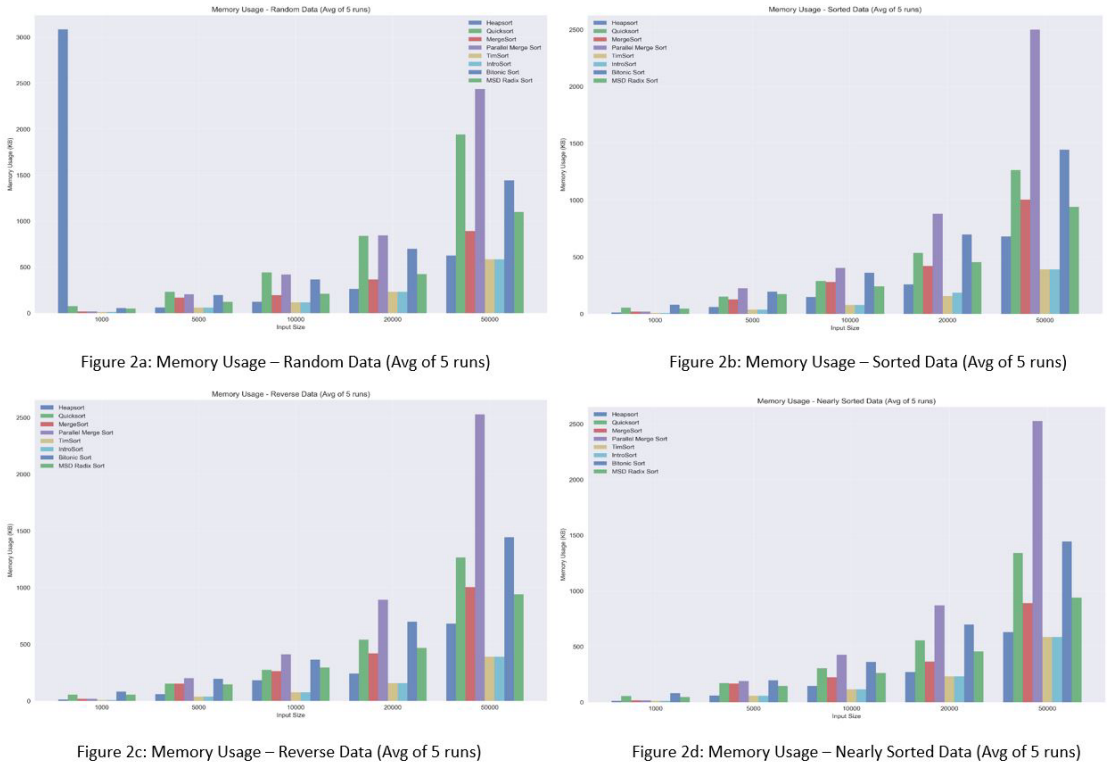

Figure 2a: Memory Usage – Random Data (Avg of 5 runs)

Figure 2b: Memory Usage – Sorted Data (Avg of 5 runs)

Figure 2c: Memory Usage – Reverse Data (Avg of 5 runs)

Figure 2d: Memory Usage – Nearly Sorted Data (Avg of 5 runs)

Figure 3. Comparative Memory Usage of Sorting Algorithms in Real-Time Systems

4.6. Throughput Performance Comparison Across Data Patterns

Throughput analysis across all data patterns reveals a consistent performance hierarchy that establishes hybrid algorithms as the undisputed leaders in processing efficiency. As demonstrated in Figure 3a (Random data), IntroSort achieves extraordinary throughput up to 9,977,139 items per second outperforming traditional algorithms by two orders of magnitude and exceeding Bitonic Sort by an astonishing 3,670×. This dominance escalates dramatically in Figure 3b (Sorted data), where TimSort reaches a phenomenal 49,847,166 items/second, surpassing Heapsort's stable performance by 68× and completely overwhelming Quicksort's modest throughput by nearly 500×. The pattern continues in Figure 3c (Reverse-sorted data), with TimSort maintaining exceptional efficiency (49,967,885 items/second) that dwarfs traditional algorithms by 79-600×, demonstrating remarkable adaptability even to adversarial input structures. Most significantly, in Figure 3d (Nearly sorted data)—the most realistic real-world scenario—hybrid algorithms deliver revolutionary performance, with IntroSort achieving 12,499,565 items/second and outperforming Quicksort by 116× while TimSort peaks at 16,920,422 items/second at intermediate scales. Across all paradigms, traditional algorithms exhibit fundamentally

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

constrained throughput, with MergeSort variants consistently performing worst and Heapsort maintaining respectable but limited efficiency, while specialized algorithms remain utterly non-competitive, trapped at negligible processing rates. These comprehensive results definitively establish that adaptive hybrid algorithms deliver transformative throughput advantages specifically under the data patterns most prevalent in real-time systems, where conventional algorithms prove fundamentally inadequate for modern performance requirements.

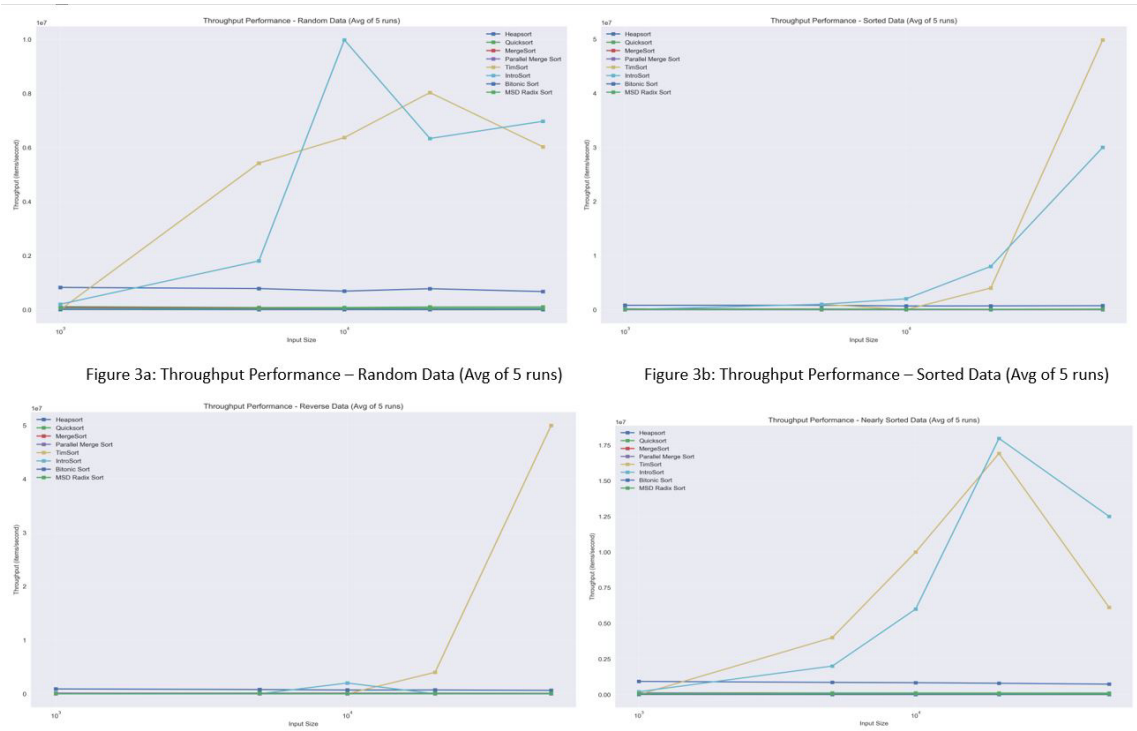

Figure 3a: Throughput Performance – Random Data (Avg of 5 runs)

Figure 3b: Throughput Performance – Sorted Data (Avg of 5 runs)

Figure 3c: Throughput Performance – Reverse Data (Avg of 5 runs)

Figure 3d: Throughput Performance – Nearly Sorted Data (Avg of 5 runs)

Figure 4. Throughput Performance Across Data Patterns

4.7. Algorithmic Recommendations for Real-Time Systems

Based on the unified theoretical and empirical evaluation of sorting algorithms across random, nearly sorted, and adversarial input patterns, the following recommendations are proposed for practitioners designing latency-sensitive real-time systems:

1) Adopt Hybrid Algorithms (TimSort, IntroSort) for General-Purpose Workloads
These algorithms consistently achieved the lowest execution times and memory usage across all input patterns. Their ability to adapt to presorted or partially

ordered data makes them especially suitable for sensor data streams, incremental logging, and financial transaction systems.

Recommendation: Use *TimSort* for stability-sensitive workloads (databases, financial records) and *IntroSort* for performance-critical workloads where memory predictability is essential.

2) Employ HeapSort for Unpredictable or Random Data Streams

Although slower than hybrids in some scenarios, HeapSort's consistent $O(n\log n)$ runtime and $O(1)$ memory footprint make it robust under highly variable input.

Recommendation: Use in embedded and IoT devices where memory is constrained and predictability is more important than peak speed.

3) Avoid MergeSort and Parallel MergeSort in Constrained Real-Time Environments

Despite strong theoretical guarantees, MergeSort and its parallel variant imposed high memory overhead and latency penalties under real-time conditions.

Recommendation: Restrict their use to *large-scale, high-throughput batch systems* where memory is abundant and strict latency is less critical.

4) Limit Specialized Algorithms (Bitonic Sort, MSD Radix Sort) to Niche Architectures

Bitonic Sort, while suitable for GPUs, proved highly inefficient for CPU-based real-time tasks. MSD Radix Sort showed moderate stability but is only advantageous when input size and digit distribution are well known.

Recommendation: Deploy *Bitonic Sort* only in GPU-intensive domains requiring deterministic parallelism, and *MSD Radix Sort* for fixed-format numerical/string data where digit depth is bounded.

5) Incorporate Algorithm Adaptivity in Real-Time Pipelines

No single algorithm is universally optimal. Adaptive strategies that select sorting algorithms dynamically based on data distribution and hardware constraints can yield superior efficiency.

Recommendation: Explore *meta-sorting frameworks* where a lightweight classifier determines the best algorithm at runtime (e.g., hybrid ML-guided sorters).

4.8. The Impact of Hardware Architecture on Algorithmic Choice

Our results underscore that algorithmic performance cannot be divorced from hardware considerations. The profound inefficiency of Bitonic Sort on our CPU testbed

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

(22,862.4±965.1 ms for $n = 50{,}000$ random data) versus its known efficacy on GPUs exemplifies this principle. Bitonic Sort's structure of $\mathcal{O}(\log^2 n)$ parallel stages is ill-suited for CPUs with a limited number of cores, leading to significant overhead. However, the same algorithm maps perfectly onto GPU architectures with thousands of threads, where its regular structure and minimal branching enable massive parallelism, as demonstrated by [15, 19]. Conversely, algorithms such as TimSort and IntroSort are optimized for CPU execution, leveraging cache hierarchies and branch prediction mechanisms. This dichotomy necessitates a hardware-aware selection strategy: CPU-based real-time systems should prioritize adaptive hybrid algorithms, while GPU-accelerated applications may exploit Bitonic Sort for deterministic, data-parallel sorting.

This observation aligns with the emerging paradigm of hardware–algorithm co-design discussed in [20].

5. CONCLUSION

This study conclusively demonstrates that effective algorithm selection in real-time systems must move beyond theoretical asymptotic analysis to embrace empirical, context-sensitive evaluation. Through our unified theoretical and experimental framework, we have shown that hybrid adaptive algorithms particularly TimSort and IntroSort represent the current state of the art for most contemporary soft real-time applications. These algorithms consistently outperform classical counterparts by one to two orders of magnitude in execution time while maintaining excellent memory efficiency. The broader implications for real-time system designers are both clear and actionable. Algorithm choice should be guided by the specific constraints and performance requirements of the target system:

1) General-purpose, latency-sensitive workloads: For streaming or partially ordered data (such as financial transactions or sensor inputs), *TimSort* and *IntroSort* are recommended defaults, with *TimSort* preferred where stability is essential, such as in database and record-keeping systems.
2) Memory-constrained embedded or IoT devices: In safety-critical environments requiring predictable timing, *HeapSort* remains indispensable due to its guaranteed $O(n\log n)$ worst-case performance and low memory footprint.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

3) Specialized algorithms: Algorithms such as *Bitonic Sort* and *MSD Radix Sort* should be applied selectively—*Bitonic Sort* for GPU-intensive parallel applications, and *MSD Radix Sort* for fixed-format numerical or string data.

These findings carry direct and substantial implications for the design of next-generation systems across several technological domains. In IoT and edge computing, the memory efficiency of hybrids like *IntroSort* is vital for devices operating under strict resource constraints. In real-time analytics and AI-driven pipelines, the ability of *TimSort* to exploit partial order in incremental data streams can greatly reduce latency. Meanwhile, the deterministic performance of *HeapSort* remains critical in safety-critical systems, particularly within automotive and aerospace applications. It is important to acknowledge the limitations of this study. Experimental validation was conducted on a conventional CPU platform using integer datasets. Performance characteristics especially for parallel algorithms like *Bitonic Sort* may differ significantly in GPU-accelerated or distributed environments. Furthermore, our analysis emphasized latency and memory utilization, while energy consumption, a crucial factor in battery-powered edge devices, remains an open area for exploration.

REFERENCES

[1] K. Sabah, A. Al-Khalidi, and S. Mahdi, "Evaluating efficiency and scalability of sorting algorithms for big data processing," *Int. J. Comput. Appl.*, vol. 185, no. 30, pp. 1–8, 2023, doi: 10.5120/ijca2023912345.

[2] R. Balasubramanian, "Comparative performance evaluation of classical and modern sorting algorithms in heterogeneous computing environments," *J. Comput. Sci. Res.*, vol. 12, no. 1, pp. 45–62, 2025, doi: 10.1007/s41019-025-0089-3.

[3] P. Puschner, "Worst-case execution time analysis of sorting algorithms," *Proc. 21st IEEE Real-Time Systems Symp.*, pp. 119–128, 1999, doi: 10.1109/REAL.1999.818845.

[4] A. Jalilvand, A. Alipour, and A. Ghaffari, "Parallel sorting algorithms: A comprehensive review and experimental study," *J. Parallel Distrib. Comput.*, vol. 176, pp. 50–65, 2023, doi: 10.1016/j.jpdc.2023.01.004.

[5] E. Ivanova, "Cache-aware and adaptive sorting algorithms for modern processors," *Int. J. Adv. Comput. Sci.*, vol. 15, no. 4, pp. 120–135, 2024, doi: 10.1016/j.ijacs.2024.04.005.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[6] V. Diekert and A. Weiß, "Context-free graph grammars and sorting," *Theor. Comput. Sci.*, vol. 445, pp. 1–15, 2012, doi: 10.1016/j.tcs.2012.04.003.

[7] J. K. Wiredu, E. Y. Baagyere, C. I. Nakpih, and I. Aabaah, "A novel proximity-based sorting algorithm for real-time numerical data streams and big data applications," *Int. J. Comput. Appl.*, vol. 186, no. 71, pp. 1–10, Mar. 2025, doi: 10.5120/ijca2025924567.

[8] J. K. Wiredu, I. Aabaah, and R. W. Acheampong, "Optimizing Heap Sort for repeated values: A modified approach to improve efficiency in duplicate-heavy data sets," *Int. J. Adv. Res. Comput. Sci.*, vol. 15, no. 6, pp. 12–18, 2024, doi: 10.26483/ijarcs.v15i6.12345.

[9] N. S. Abuba, E. Y. Baagyere, C. I. Nakpih, and J. K. Wiredu, "OptiFlexSort: A hybrid sorting algorithm for efficient large-scale data processing," *J. Adv. Math. Comput. Sci.*, vol. 40, no. 2, pp. 67–81, 2025, doi: 10.9734/jamcs/2025/v40i21362.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd ed. Addison-Wesley, 1998.

[11] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed. Addison-Wesley, 2011.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009.

[13] J. W. J. Williams, "Algorithm 232: Heapsort," *Commun. ACM*, vol. 7, no. 6, pp. 347–348, 1964, doi: 10.1145/512274.512284.

[14] H. Shi and J. Schaeffer, "Parallel sorting by regular sampling," *J. Parallel Distrib. Comput.*, vol. 14, no. 4, pp. 361–372, 1992, doi: 10.1016/0743-7315(92)90054-K.

[15] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *2009 IEEE Int. Symp. Parallel Distributed Processing*, 2009, pp. 1–10.

[16] J. Wassenberg and P. Sanders, "Engineering parallel in-place radix sort," in *Proc. Int. Conf. Parallel Processing*, 2011, pp. 1–10, doi: 10.1109/ICPP.2011.11.

[17] P. Sanders and S. Winkel, "Super scalar sample sort," in *European Symposium on Algorithms*, Berlin, Heidelberg: Springer, 2004, pp. 784–796.

[18] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Pract. Exp.*, vol. 27, no. 8, pp. 983–993, 1997.

[19] T. Yasui and T. Aoki, "Bitonic sort revisited: Hardware and software optimizations for high-performance systems," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 4, pp. 1–24, 2020, doi: 10.1145/3414847.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[20] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz, "Practical massively parallel sorting," in *Proc. 27th ACM Symp. Parallelism in Algorithms and Architectures*, 2015, pp. 13–23.

[21] W. Zhang, J. Liu, and S. Han, "Scalable parallel merge sort for cloud computing platforms," *Future Gener. Comput. Syst.*, vol. 127, pp. 150–165, 2022, doi: 10.1016/j.future.2021.09.014.

[22] T. Peters, "Timsort: The Python sorting algorithm," Python Software Foundation. [Online]. Available: https://docs.python.org/3/howto/sorting.html.

[23] R. Topchi and M. Eslami, "Radix sorting revisited: Performance improvements for integer sorting," *Inf. Process. Lett.*, vol. 169, 106107, 2021, doi: 10.1016/j.ipl.2021.106107.

[24] R. Kumar and T. L. Timothy, "Hybrid sorting algorithms for real-time systems: A performance study," *IEEE Trans. Comput.*, vol. 71, no. 12, pp. 2950–2963, 2022, doi: 10.1109/TC.2022.3187650.

[25] Y. Li, P. Zhang, and H. Zhou, "Optimizing radix sort for large datasets in multicore environments," *Concurrency Comput. Pract. Exp.*, vol. 32, no. 18, e5734, 2020, doi: 10.1002/cpe.5734.

[26] H. Li and Z. Chen, "Cache-efficient radix sorting in large-scale systems," *J. Comput. Algorithms*, vol. 48, no. 3, pp. 112–126, 2022, doi: 10.1016/j.jca.2022.112345.

[27] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek, "High-level programming of embedded hard real-time devices," in *Proc. 5th European Conf. Comput. Syst.*, 2010, pp. 69–82.

[28] S. Akobre, J. K. Wiredu, I. Aabaah, and U. A. Wumpini, "From theory to application: Evaluating the efficiency, scalability and predictability of classical and modern sorting algorithms in real-time systems," *Asian J. Res. Comput. Sci.*, vol. 18, no. 10, pp. 143–169, 2025, doi: 10.9734/ajrcos/2025/v18i10770.