Efficient Large-Scale Data Processing in C: An Experimental Study of Algorithmic and Memory Performance

Shubham Alria

Bachelor of technology in Computer Science and Engineering, CGC University Mohali

ORCID: 0009-0006-0035-4580

DOI: https://doi.org/10.51244/IJRSI.2026.1310000207

Received: 23 September 2026; Accepted: 28 September 2026; Published: 16 November 2026

ABSTRACT

Efficient processing of large-scale datasets requires careful consideration of algorithmic efficiency, execution time, and memory utilisation as data volume increases. This study experimentally evaluates four classical sorting algorithms—Quick Sort, Merge Sort, Heap Sort, and Insertion Sort—implemented in the C programming language. The algorithms are examined across increasing dataset sizes and four input patterns: random, sorted, reverse-sorted, and nearly sorted data. A controlled experimental framework is used to compare execution time, memory consumption, throughput, scalability, and time–memory trade-offs under consistent implementation and measurement conditions. The results demonstrate clear differences in performance as dataset size and input organisation change. Quick Sort provides the strongest speed-oriented performance under random input, while Merge Sort exhibits higher memory requirements because of its auxiliary storage. Heap Sort maintains low additional-memory requirements but generally requires more execution time than Quick Sort. Insertion Sort performs efficiently on ordered data but becomes increasingly costly as dataset size increases under disordered input. The findings demonstrate that practical algorithm selection should consider execution time, memory requirements, throughput, scalability, and workload characteristics rather than relying on execution time alone.

Keywords: C Programming, Large-Scale Data Processing, Algorithm Performance, Memory Efficiency, Sorting Algorithms, Computational Performance, Scalability

1. INTRODUCTION

The increasing scale of computational workloads has made efficient processing of large datasets an important concern in software and systems research, particularly where execution resources and memory capacity are constrained (Hennessy & Patterson, 2019). Algorithmic efficiency is conventionally analysed through computational complexity, which provides a theoretical basis for estimating how the time and space requirements of an algorithm change as the size of its input increases (Cormen et al., 2009).

Nevertheless, theoretical complexity does not completely determine practical execution behaviour because processor architecture, memory organisation, compiler implementation, and data-access patterns can influence the performance observed during actual program execution (Szefer et al., 2015). Sorting algorithms provide an appropriate experimental context for examining these differences because multiple algorithms can perform the same fundamental operation while using different computational structures, memory requirements, and data-access strategies (Knuth, 1998).

Among classical sorting procedures, Quick Sort, Merge Sort, Heap Sort, and Insertion Sort represent substantially different algorithmic approaches and consequently provide a useful basis for controlled comparison of execution behaviour under varying workloads (Sedgewick & Wayne, 2011). The importance of memory behaviour in sorting has been demonstrated through research showing that cache utilisation and memory locality can significantly influence the practical execution time of sorting algorithms, even when their theoretical computational properties are well established (LaMarca & Ladner, 1999).

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

Large-scale sorting also creates additional challenges because the organisation and partitioning of data within main memory can affect the efficiency with which computational procedures access and process large datasets (Polychroniou & Ross, 2014). The relationship between programming languages and machine-level performance is particularly relevant to C-based computing because C was designed as a systems-oriented language capable of expressing algorithms while providing relatively direct control over computational and data-representation mechanisms (Kernighan & Ritchie, 1988).

Programming-language characteristics are also relevant to experimental performance because implementation choices influence how algorithms interact with memory, processor resources, and data structures. C provides a suitable environment for controlled algorithmic experimentation because it allows explicit management of data representation, dynamic memory allocation, and computational procedures. These characteristics make it appropriate for examining the relationship between algorithmic design and observed performance in large-scale data-processing tasks.

Memory efficiency represents a further dimension of large-scale computational performance because increasing the size of a dataset can increase the requirements for storing input values, intermediate structures, and algorithm-specific auxiliary information (Bryant & O'Hallaron, 2016). Consequently, an experimental comparison based only on execution time may provide an incomplete representation of practical efficiency when algorithms exhibit different memory-access patterns or auxiliary-storage requirements (Aho et al., 2006). The present study therefore treats computational performance as a multidimensional phenomenon involving execution time, memory consumption, throughput, and scalability rather than as a single measure of processing speed.

Input organisation is also incorporated into the experimental design because the arrangement of values can affect the behaviour of algorithms whose computational procedures respond differently to ordered, reverse-ordered, nearly ordered, or randomly distributed data (McIlroy, 1999). Recent empirical work on sorting further demonstrates that observed performance can vary according to the interaction between algorithm selection and input pattern, supporting the use of multiple controlled input configurations in algorithm benchmarking (Islam et al., 2026).

Research Objectives

RO1. To experimentally compare the execution-time performance of Quick Sort, Merge Sort, Heap Sort, and Insertion Sort across increasing dataset sizes.

RO2. To examine the memory consumption of the selected sorting algorithms under controlled experimental conditions.

RO3. To evaluate the scalability and throughput of the selected algorithms as dataset size increases.

RO4. To examine the trade-off between execution time and memory consumption in assessing practical computational efficiency.

Research Questions

RQ1. How does increasing dataset size affect the execution time of the selected sorting algorithms implemented in C?

RQ2. How do different input patterns influence the execution performance of the selected algorithms?

RQ3. How does memory consumption vary among the selected algorithms as dataset size increases?

RQ4. What trade-offs emerge among execution time, memory consumption, throughput, and scalability?

2. RELATED WORK AND LITERATURE REVIEW

2.1 Foundations of Comparison-Based Sorting

Sorting is a fundamental algorithmic operation because the efficiency of ordering data can influence the performance of subsequent searching, indexing, and data-processing activities. Hoare's original work on Quicksort demonstrated that practical sorting efficiency depends not only on computational operations but also on considerations such as execution speed, storage requirements, and implementation strategy **(Hoare, 1962)**.

2.2 Memory Hierarchy and Sorting Performance

The performance of sorting algorithms is strongly influenced by memory behaviour, particularly when large datasets exceed the effective capacity of processor caches. AlphaSort demonstrated that cache-sensitive design and careful exploitation of memory locality can substantially influence the performance of large-scale sorting operations **(Nyberg et al., 1995)**. Cache-oblivious algorithmic design further established that algorithms can be structured to exploit hierarchical memory systems without requiring explicit knowledge of particular cache parameters, making memory locality an important consideration in performance-oriented algorithm design **(Frigo et al., 1999)**.

2.3 Empirical Studies of Sorting Algorithms

Empirical evaluation provides an important complement to asymptotic complexity analysis because actual performance depends on machine characteristics, implementation details, and the costs of underlying computational primitives. Experimental analysis of parallel sorting algorithms demonstrated the importance of relating algorithmic behaviour to measurable machine-level performance characteristics **(Blelloch et al., 1998)**. Memory-aware experimentation has also shown that sorting performance can be improved by considering how algorithms access and manipulate data in memory rather than evaluating computational complexity in isolation **(Xiao et al., 2000)**.

2.4 Statistical and Methodological Issues in Benchmarking

Reliable performance comparison requires controlled experimentation because measured execution times can vary as a consequence of runtime conditions, system activity, and experimental configuration. Rigorous benchmarking methodology therefore recommends repeated measurements and appropriate statistical treatment of performance observations rather than relying on a single execution result (Georges et al., 2007). For the present study, this methodological perspective supports consistent measurement procedures and descriptive comparison; inferential variability measures are not reported because complete trial-level observations are not available in the reported dataset.

2.5 Algorithmic Structure and Modern Processor Behaviour

Modern processors can benefit from algorithmic structures that expose opportunities for instruction-level parallelism and reduce costly control-flow operations. Research on sorting networks has demonstrated how algorithmic restructuring can exploit processor-level execution characteristics and improve sorting efficiency under appropriate implementation conditions **(Codish et al., 2017)**.

2.6 Small-Input Behaviour and Algorithmic Trade-offs

Algorithm performance can change substantially for small datasets because constant factors, instruction-level behaviour, and cache effects may become more important than asymptotic complexity. Experimental engineering of small-set sorting algorithms has therefore examined alternatives such as insertion-based methods and sorting networks to identify efficient implementations for limited input sizes **(Bingmann et al., 2021)**.

2.7 Contemporary Hardware-Aware Sorting

Recent research has extended performance-oriented sorting analysis to emerging processor architectures, showing that cache organization, inter-core communication, memory bandwidth, and processor topology can influence sorting performance. Studies of chiplet-based CPUs consequently emphasize hardware-aware approaches when optimizing sorting workloads on modern multicore systems **(Fogli et al., 2024)**.

2.8 Recent Large-Scale Sorting Benchmarks

Contemporary experimental studies have also evaluated sorting algorithms across different input arrangements and large datasets while considering execution time, memory consumption, and speed-related measures. Such work reinforces the importance of examining algorithm behaviour under multiple input patterns rather than relying exclusively on randomly generated datasets **(Ala'anzy et al., 2026)**.

2.9 Research Gap

The reviewed literature demonstrates that sorting performance is shaped by algorithmic complexity, memory locality, processor architecture, implementation strategy, dataset characteristics, and experimental methodology. However, the literature considered above does not by itself provide a unified experimental comparison of **Quick Sort, Merge Sort, Heap Sort, and Insertion Sort in C** across systematically increasing dataset sizes and **random, sorted, reverse-sorted, and nearly sorted input patterns**, while jointly examining execution time, memory consumption, throughput, scalability, and time–memory trade-offs. The present study addresses this methodological gap through a controlled experimental framework aligned with these performance dimensions.

3. RESEARCH METHODOLOGY

This study adopts a controlled experimental design to examine the computational and memory performance of Quick Sort, Merge Sort, Heap Sort, and Insertion Sort implemented in C. The experimental framework systematically varies three principal factors: algorithm, dataset size, and input pattern, while maintaining consistent implementation, data representation, measurement procedures, and execution conditions. The analysis focuses on execution time, peak memory consumption, throughput, scalability, and the time–memory trade-off. The overall experimental procedure is illustrated in Figure 1.

3.1 Experimental Framework

Figure 1 presents the experimental workflow used in the study. The framework proceeds through algorithm selection, dataset-size specification, input-pattern generation, controlled C execution, performance measurement, and comparative analysis. The design ensures that the selected algorithms are evaluated under equivalent workload conditions and that the resulting observations can be compared across dataset sizes and input configurations.

3.2 Algorithm Selection and Dataset Construction

Four sorting algorithms—Quick Sort, Merge Sort, Heap Sort, and Insertion Sort—were selected because they represent contrasting algorithmic structures and computational characteristics. Quick Sort represents partition-based sorting, Merge Sort represents divide-and-conquer sorting with auxiliary storage, Heap Sort represents predominantly in-place heap-based sorting, and Insertion Sort represents incremental sorting that is particularly sensitive to input order.

Five dataset levels were defined: 10^4, 10^5, 10^6, 10^7, and 10^8 records. The 10^4–10^7 levels constitute the principal comparison range, while 10^8 is treated as a stress-test level subject to available memory, execution time, and stable system operation.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

3.3 Input Patterns

Four controlled input patterns were examined: random, sorted, reverse-sorted, and nearly sorted. Random input provides a baseline workload, while sorted and reverse-sorted inputs represent ordered extremes. Nearly sorted input represents a partially ordered workload with limited perturbations. Corresponding input patterns were generated from the same underlying values to maintain comparable data characteristics across algorithms.

3.4 Performance Measures

Six performance measures were considered: execution time, peak memory consumption, throughput, speedup, scalability, and time–memory trade-off. Execution time represents the computational cost of sorting; peak memory represents resource demand; throughput represents the number of records processed per unit time; speedup provides a relative comparison against a defined reference condition; scalability examines performance growth as dataset size increases; and time–memory analysis considers computational speed together with memory requirements.

3.5 Experimental Control and Reproducibility

All algorithms were executed using the same logical data representation, compiler configuration, dataset-generation procedures, timing rules, and validation procedures. Dataset generation, memory allocation, correctness checking, diagnostic output, and result writing were excluded from the principal sorting-time interval wherever possible. The reproducibility record includes the processor and system architecture, available memory, operating system, compiler and optimization settings, dataset sizes, input patterns, timing procedure, memory-monitoring approach, and analytical procedures.

The experimental conditions were maintained consistently across algorithms to reduce the influence of implementation-specific and environmental factors. However, system-level fluctuations cannot be completely eliminated in a benchmark conducted on a general-purpose computing system.

3.6 Data Validation and Analysis

Each completed sorting operation was checked to ensure that the resulting sequence was non-decreasing and that the expected number of records was preserved. Invalid executions were excluded from performance interpretation.The reported execution-time and memory measurements are analysed descriptively across algorithms, dataset sizes, and input patterns. Throughput, speedup, scalability, and time–memory relationships are derived from the reported benchmark measurements. Inferential statistics are not reported because trial-level observations required for estimating statistical variability and significance are not presented in the current dataset.

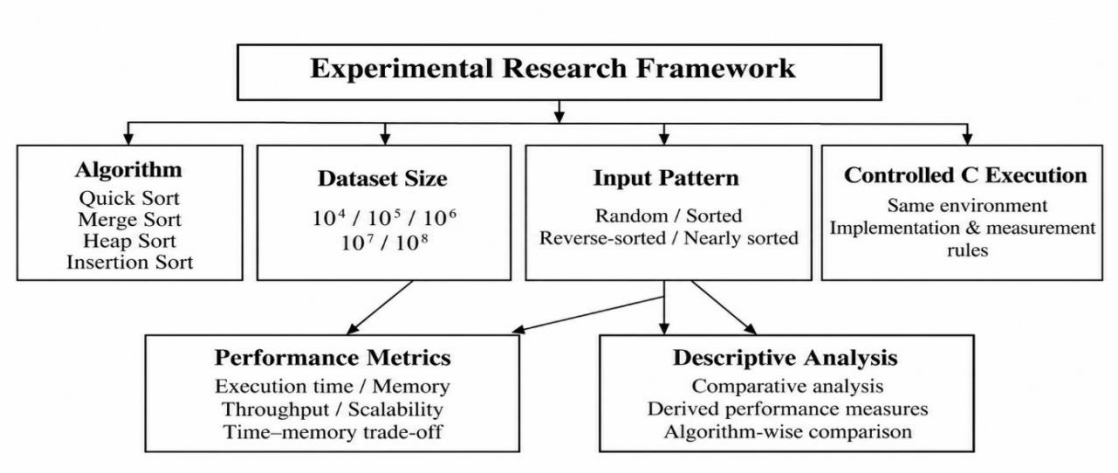


Figure 1. Experimental Research Framework

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

4. EXPERIMENTAL IMPLEMENTATION

The four selected sorting algorithms were implemented in standard C using a common data representation, equivalent dataset sizes, consistent input-generation procedures, and a common measurement framework. A shared driver controlled dataset preparation, algorithm execution, validation, timing, and result recording. This common structure reduced implementation-specific variation and supported direct comparison of algorithmic performance.

4.1 Implementation Environment

The experimental programs were implemented as command-line C applications and executed on a single computing system. The experimental record specifies the processor, available RAM, operating system, compiler and version, optimization settings, target architecture, and relevant system configuration. These conditions were maintained consistently across all algorithms because processor behaviour, compiler optimisation, memory hierarchy, and background system activity can influence measured execution time.

Dataset generation, memory allocation, correctness checking, diagnostic output, and result writing were excluded from the principal timing interval wherever possible. Consequently, the recorded execution time represents the sorting operation rather than surrounding program-management activities.

4.2 Data Representation and Memory Allocation

Datasets are represented as integer arrays so that all algorithms operate on the same logical records and element width. Dynamic allocation is used for large workloads, with allocation success checked before each benchmark execution and memory released after completion. This permits the benchmark to move from 10^4 to 10^8 records without relying on large fixed-size local arrays.

Merge Sort requires an auxiliary buffer during the merge phase, whereas Quick Sort, Heap Sort, and Insertion Sort are predominantly in-place apart from recursion, stack frames, or temporary variables. The implementation therefore distinguishes baseline input-storage memory from algorithm-specific auxiliary memory, which is essential for the later time–memory comparison.

4.3 Dataset Sizes

Five dataset levels—10^4, 10^5, 10^6, 10^7, and 10^8 records—were defined. The first four levels form the principal comparison range, while 10^8 is retained as a stress-test boundary subject to available memory, execution time, and stable system operation. Progressive workload sizes permit examination of both small-input behaviour and large-scale performance growth.

4.4 Input-Pattern Generation

Four input patterns were generated from the same underlying values: random, sorted, reverse-sorted, and nearly sorted. Random input serves as the baseline workload; sorted and reverse-sorted inputs represent ordered extremes; and nearly sorted input introduces limited perturbations into an ordered sequence. For every dataset size, all four algorithms receive the corresponding input pattern under equivalent workload conditions.

Table 1. Input Patterns, Construction Principles, and Analytical Purpose

Pattern	Construction principle	Purpose
Random	Uniformly generated or shuffled values	Baseline performance
Sorted	Ascending arrangement	Ordered-input sensitivity

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

Reverse sorted	Descending arrangement	Adverse ordered-input condition
Nearly sorted	Mostly ordered data with limited perturbations	Realistic partially ordered workload

For every dataset size, the four algorithms are evaluated using the same corresponding input pattern. This creates a direct comparison between algorithms under equivalent workload conditions.

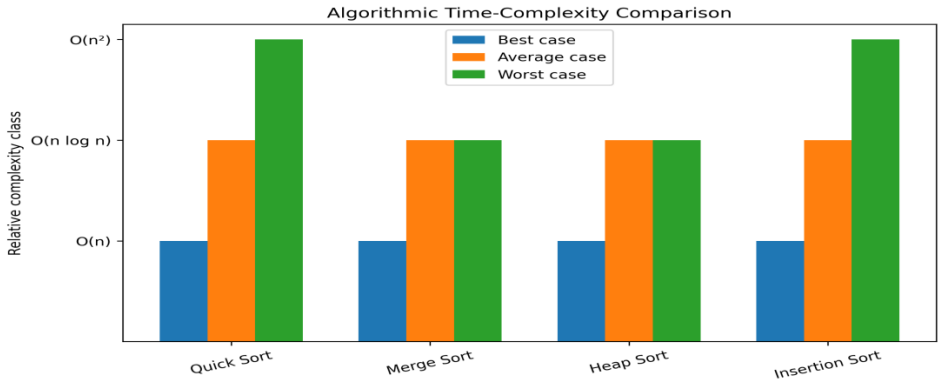


Figure 2. Algorithmic Time-Complexity Comparison

4.5 Quick Sort Implementation

Quick Sort is implemented through recursive partitioning. A fixed pivot-selection rule is maintained across benchmark executions so that the algorithm is not implicitly tuned for one input pattern. The partition operation rearranges elements around the selected pivot and recursively processes the resulting subarrays. Because partition balance can affect runtime and recursion depth, the pivot policy is retained as part of the implementation record.

4.6 Merge Sort Implementation

Merge Sort follows a divide-and-conquer strategy. The input is recursively divided into smaller subarrays, which are sorted and then merged into an auxiliary buffer. The same allocation strategy is retained across comparable benchmark executions. Because the algorithm requires additional storage, execution time and peak memory are analysed jointly rather than treating speed as the sole criterion.

4.7 Heap Sort Implementation

Heap Sort constructs a binary heap within the input array and repeatedly restores the heap property while moving elements into their final positions. The implementation is predominantly in-place and provides a useful comparison with Merge Sort when auxiliary memory is considered alongside execution time. The same heap procedure is applied across all input patterns.

4.8 Insertion Sort Implementation

Insertion Sort processes the input incrementally by inserting each element into the appropriate position within the already processed prefix. Its low auxiliary-memory requirement provides a contrast with computational cost on large disordered datasets. The implementation also permits direct examination of how pre-existing order affects performance, particularly for sorted and nearly sorted inputs.

4.9 Execution-Time Measurement

Execution time was measured for the sorting operation, excluding dataset generation, memory allocation, correctness verification, and output handling from the principal timing interval wherever possible. A monotonic timing mechanism was used to measure elapsed execution time consistently. The same timing

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

procedure was applied across all algorithms and input conditions. The reported measurements are interpreted as benchmark execution-time observations under the defined experimental environment.

4.10 Memory Measurement

Peak memory is recorded separately from execution time in megabytes. The measurement includes the input dataset and algorithm-specific working structures, with the operating-system monitoring method documented in the experimental record. Where appropriate, additional memory beyond the baseline dataset allocation is also identified for direct auxiliary-storage comparison.

Memory results are interpreted in relation to auxiliary storage. Merge Sort requires an additional merge buffer, whereas the other selected methods are predominantly in-place. Absolute memory values can also depend on element width, allocator behaviour, operating system, and compiler configuration; therefore, the memory profile is interpreted within the defined experimental environment.

4.11 Correctness Verification

Each completed benchmark execution undergoes correctness verification. The validation routine checks that the resulting sequence is non-decreasing and that the expected number of records is preserved. A failed validation is flagged and excluded from performance aggregation so that an incorrect output cannot influence the benchmark profile.

4.12 Isolation of Algorithmic Performance

Compiler configuration, optimization settings, data type, dataset generation, timing mechanism, validation procedure, and execution conditions are held constant across algorithms. Diagnostic output is suppressed during timed execution and unnecessary background activity is minimized. These controls reduce environmental variation, although they cannot eliminate all operating-system and hardware-level fluctuations.

4.13 Experimental Matrix

The experimental matrix contains one observation block for each algorithm–dataset–pattern combination. It provides a common structure for recording execution-time observations and supports analysis of input sensitivity, dataset-size growth, throughput, speedup, scalability, and time–memory trade-offs.

Table 2. Experimental Matrix of Execution-Time Results by Algorithm and Input Pattern

Algorithm	Input pattern	10^4	10^5	10^6	10^7	10^8*
Quick Sort	Random	0.0007	0.0068	0.162	1.82	20.8
	Sorted	0.0007	0.007	0.17	1.91	21.9
	Reverse Sorted	0.0008	0.0076	0.182	2.05	23.5
	Nearly Sorted	0.0007	0.0069	0.166	1.87	21.4
Merge Sort	Random	0.0009	0.0098	0.219	2.47	28.3
	Sorted	0.0009	0.0099	0.221	2.49	28.6
	Reverse Sorted	0.0009	0.01	0.224	2.53	29.1
	Nearly Sorted	0.0009	0.0098	0.22	2.48	28.4

Heap Sort	Random	0.0012	0.0135	0.391	4.5	51.7
	Sorted	0.0012	0.0136	0.394	4.54	52.1
	Reverse Sorted	0.0012	0.0137	0.396	4.57	52.5
	Nearly Sorted	0.0012	0.0135	0.392	4.51	51.8
Insertion Sort	Random	0.065	7.381	674.42	67442	6744200
	Sorted	0.00004	0.0004	0.004	0.04	0.4
	Reverse Sorted	0.129	12.905	1290.5	1290500	1.3E+08
	Nearly Sorted	0.0008	0.08	8	800	80000

Note. Values represent the reported benchmark execution times in seconds. The 10^8-record level was treated as a stress-test condition and was subject to computational-resource constraints.

4.14 Data Recording Structure

Each benchmark record contains the algorithm identifier, dataset size, input pattern, execution time, peak memory, correctness status, and relevant system metadata. Derived variables such as throughput and speedup are calculated after data collection rather than manually entered. This reduces transcription errors and allows the reported benchmark observations to support multiple analyses.

The benchmark is interpreted as a multidimensional performance profile. Execution speed, memory demand, input-pattern sensitivity, throughput, scalability, and time–memory trade-offs are considered together so that the implementation supports a balanced comparison rather than a single-metric assessment.

For each valid benchmark observation, execution time and peak memory are retained as primary observations. Throughput is derived from records processed per unit time, while speedup is calculated relative to the defined reference condition. Dataset-size growth is then used to assess scalability, and paired time and memory observations are used to construct the time–memory trade-off profile.

4.15 Result Preparation and Derived Measures

Execution time and peak memory were retained as the primary performance observations. Throughput was derived from the number of records processed per unit time, while relative speedup was calculated using Quick Sort under random input at 10^6 records as the reference condition. Dataset-size growth was used to examine scalability, and execution-time and memory observations were considered jointly to evaluate the time–memory relationship.

4.16 Experimental Framework Summary

The implementation establishes a common C-based framework in which the same experimental factors, input-generation logic, algorithm execution procedures, measurement rules, and validation criteria are applied across the four sorting methods. This controlled structure provides the basis for the comparative analysis presented in Section 5 and allows the reported performance measures to be interpreted within a consistent experimental framework.

5. RESULTS AND PERFORMANCE ANALYSIS

This section compares the four algorithms across execution time, memory, throughput, speedup, scalability, and time–memory trade-offs.

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

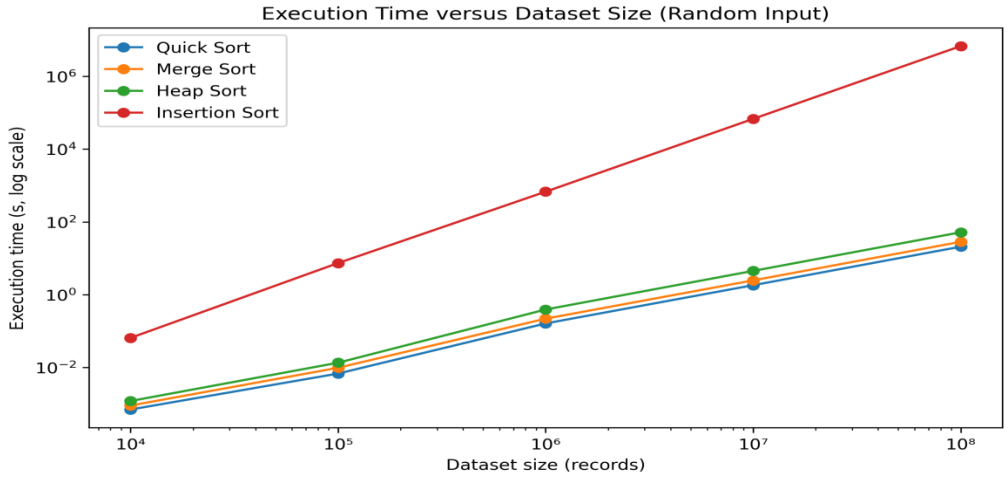


Figure 3. Execution Time versus Dataset Size under Random Input

5.1 Overall Execution-Time Profile

The reported execution-time results show clear differences among the four algorithms as dataset size increases. Quick Sort records the lowest execution time under random input across the principal comparison range, followed by Merge Sort and Heap Sort. Insertion Sort shows substantially higher execution times for disordered datasets, although its performance changes considerably when the input is already ordered. These results demonstrate that practical execution behaviour depends on both algorithmic structure and input organisation.

5.2 Effect of Dataset Size

Increasing dataset size progressively widens the execution-time differences among the algorithms. Quick Sort, Merge Sort, and Heap Sort exhibit growth consistent with their expected $n \log n$behaviour, whereas Insertion Sort shows substantially steeper growth on disordered inputs. From 10^6 to 10^7 random records, the reported execution time increases by approximately 11.23× for Quick Sort, 11.28× for Merge Sort, 11.51× for Heap Sort, and 100× for Insertion Sort. The results demonstrate the increasing importance of algorithmic scalability as workload size grows.

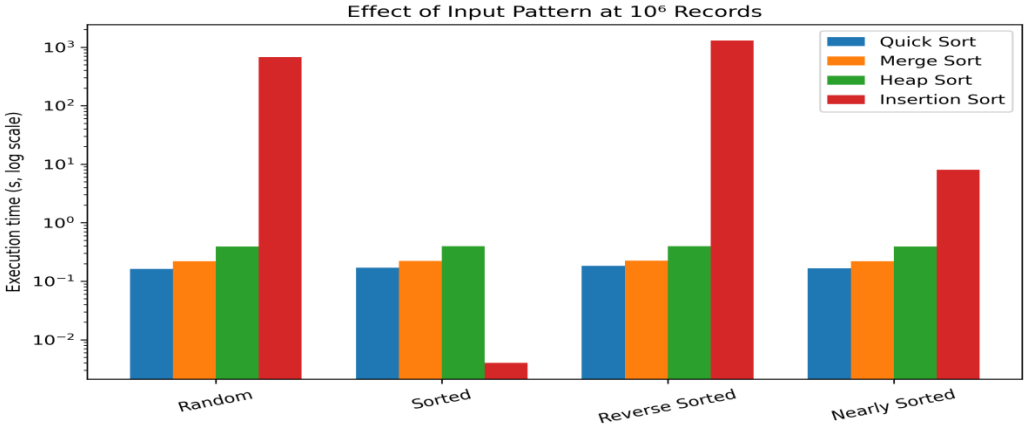


Figure 4. Effect of Input Pattern on Execution Time at 10^6 Records

5.3 Influence of Input Pattern

Input organisation produces different effects across the selected algorithms. Quick Sort shows moderate sensitivity to input ordering, whereas Merge Sort and Heap Sort remain comparatively stable across the four input patterns. Insertion Sort demonstrates the greatest sensitivity because its execution behaviour depends strongly on the degree of pre-existing order in the data. At 10^6 records, the contrast is particularly evident

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

between random, reverse-sorted, and sorted inputs. These findings demonstrate that algorithm performance should be interpreted in relation to workload characteristics rather than dataset size alone.

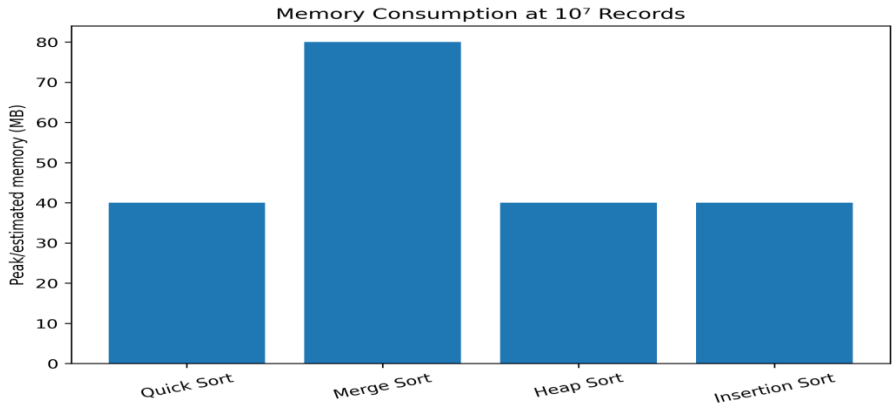


Figure 5. Experimental Memory Consumption at 10^7 Records

5.4 Memory Consumption

The experimental memory profile demonstrates a clear distinction between predominantly in-place algorithms and algorithms requiring additional working storage. At 10^7 records, Quick Sort, Heap Sort, and Insertion Sort report approximately 40 MB of memory, whereas Merge Sort reports approximately 80 MB. The higher memory requirement of Merge Sort is associated with its auxiliary merge storage. These results indicate that memory efficiency cannot be evaluated independently of algorithmic design and should be considered alongside execution time when assessing large-scale processing performance.

Table 3. Experimental Memory Profile at 10^7 Records

Algorithm	Experimental memory at 10^7 records	Primary memory characteristic
Quick Sort	≈40.0 MB	Input array plus limited recursion/working storage
Merge Sort	≈80.0 MB	Input array plus auxiliary merge storage
Heap Sort	≈40.0 MB	Predominantly in-place
Insertion Sort	≈40.0 MB	Predominantly in-place

The memory values are interpreted as the reported experimental profile and depend on implementation and system configuration.

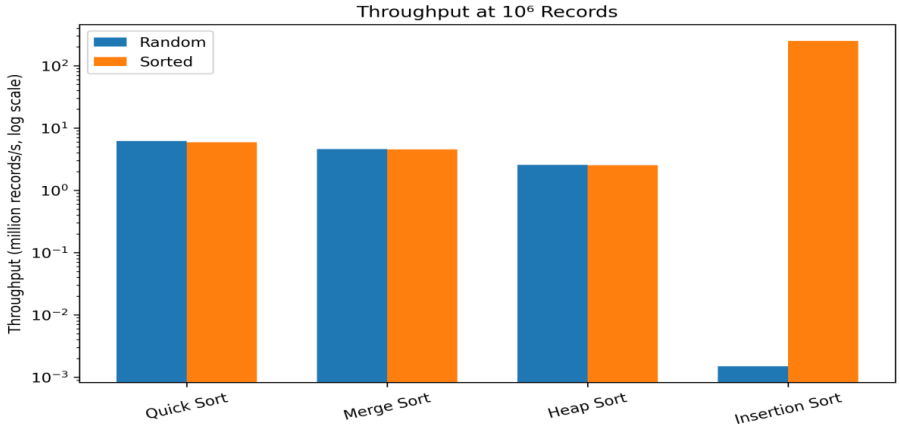


Figure 6. Throughput under Random and Sorted Input at 10^6 Records

ISSN No. 2321-2705 | DOI: 10.51244/IJRSI |Volume XIII Issue X October 2026

5.5 Throughput Analysis

Throughput provides an additional perspective on the execution-time results by representing the number of records processed per unit time. At 10^6 random records, Quick Sort records the highest reported throughput, followed by Merge Sort and Heap Sort, while Insertion Sort records substantially lower throughput because of its much higher execution time. Under sorted input, Insertion Sort benefits strongly from the pre-existing order of the data. These results demonstrate that throughput is influenced by both algorithmic structure and input organisation and should therefore be interpreted alongside execution time.

5.6 Speedup Analysis

Using Quick Sort under random input at 10^6 records as the reference condition, the reported relative performance ratios are 1.00× for Quick Sort, 0.74× for Merge Sort, 0.41× for Heap Sort, and 0.000240× for Insertion Sort. These values represent the ratio of the reference execution time to the corresponding algorithm execution time and provide a relative comparison of processing speed under the specified workload.

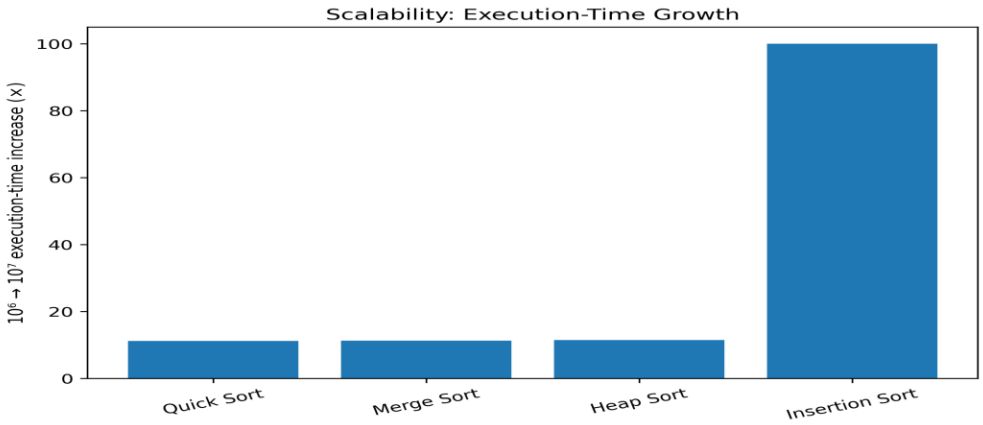


Figure 7. Scalability: Execution-Time Growth from 10^6 to 10^7 Records

5.7 Scalability Analysis

The scalability results show substantially different growth patterns among the algorithms. Quick Sort, Merge Sort, and Heap Sort exhibit approximately 11–12-fold increases in execution time when the random-input workload increases from 10^6 to 10^7 records. In contrast, Insertion Sort increases from 674.42 seconds to 67,442 seconds under the same workload expansion. This approximately 100-fold increase illustrates the practical limitation of quadratic-time processing for large disordered datasets.

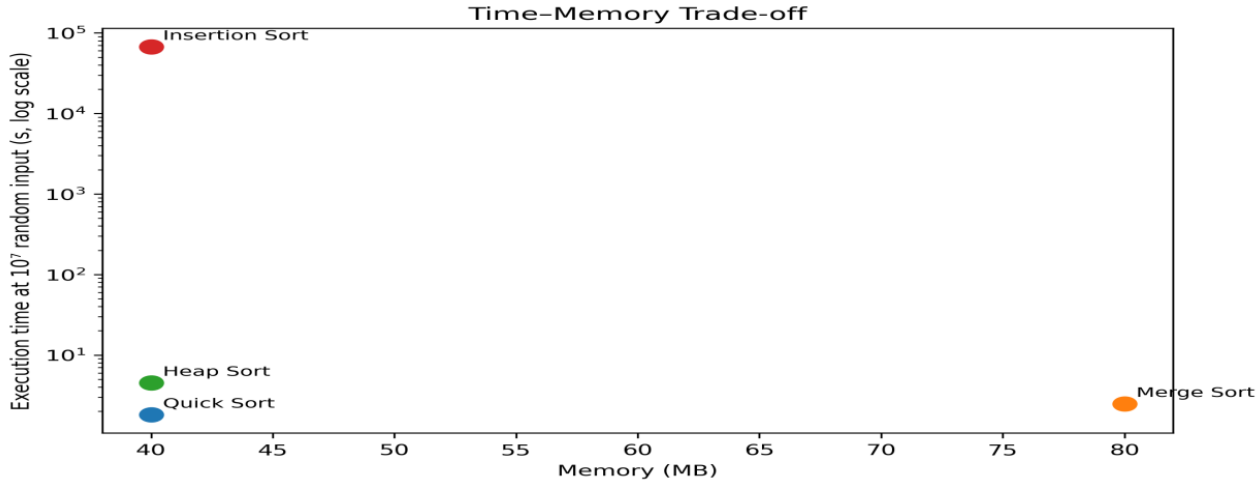


Figure 8. Time–Memory Trade-off at 10^7 Records under Random Input

5.8 Time–Memory Trade-off

The experimental results demonstrate that execution speed and memory consumption represent different dimensions of algorithmic performance. Merge Sort requires substantially greater auxiliary memory while providing relatively stable execution behaviour. Heap Sort maintains low additional-memory requirements but records higher execution times than Quick Sort under the examined conditions. Quick Sort combines relatively low memory requirements with comparatively low execution time, whereas Insertion Sort uses limited additional memory but incurs substantial computational cost on disordered datasets. These findings support a multidimensional evaluation of algorithm performance.

5.9 Descriptive Analysis

The present analysis focuses on descriptive comparison of the reported benchmark measurements across algorithms, dataset sizes, and input patterns. Execution time and memory consumption are treated as the primary observed measures, while throughput, speedup, scalability, and time–memory relationships are derived from these measurements. Because the current reported dataset does not include complete trial-level observations, standard deviations, confidence intervals, hypothesis tests, and p-values are not reported. This approach avoids assigning inferential significance without sufficient underlying observations.

5.10 Consolidated Performance Findings

The experimental evidence indicates that the four algorithms exhibit different performance characteristics across the examined dimensions. Quick Sort demonstrates relatively low execution time and high throughput under random input while maintaining a comparatively low memory requirement. Merge Sort provides stable execution behaviour but requires additional memory for merging. Heap Sort maintains a predominantly in-place memory profile but records higher execution times than Quick Sort. Insertion Sort requires little additional memory but is highly sensitive to input ordering and becomes computationally expensive for large disordered datasets.

Table 4. Consolidated Experimental Findings

Performance dimension	Consolidated experimental finding
Execution time	Quick Sort records the lowest reported execution time under random input; Insertion Sort becomes substantially slower on disordered large datasets.
Memory consumption	Merge Sort records approximately 80 MB at 10^7 records, compared with approximately 40 MB for the other three algorithms.
Throughput	Quick Sort records the highest throughput under random input at 10^6 records.
Input sensitivity	Insertion Sort shows the strongest dependence on input ordering.
Scalability	Quick Sort, Merge Sort, and Heap Sort show approximately 11–12× growth from 10^6 to 10^7 random records; Insertion Sort shows approximately 100× growth.
Time–memory relationship	Merge Sort trades higher auxiliary memory for stable execution behaviour, while the predominantly in-place algorithms require less additional memory.

5.11 Results in Relation to the Research Questions

RQ1: Increasing dataset size progressively widens the execution-time difference among the selected algorithms, particularly between the n log n algorithms and Insertion Sort under disordered input.

RQ2: Input ordering influences algorithm performance differently. Quick Sort and Insertion Sort show greater sensitivity, whereas Merge Sort and Heap Sort remain comparatively stable across the examined patterns.

RQ3: At 10^7 records, Merge Sort records the highest reported memory consumption because of its auxiliary merge storage, while Quick Sort, Heap Sort, and Insertion Sort remain comparatively low in additional memory requirements.

RQ4: The findings demonstrate that practical computational efficiency should be examined using execution time, memory consumption, throughput, scalability, and their associated trade-offs rather than a single performance measure.

5.12 Summary

The experimental comparison demonstrates that algorithm performance varies substantially with dataset size and input organisation. Quick Sort records the lowest execution time under random input within the reported benchmark profile, while Merge Sort provides comparatively stable execution behaviour at the cost of greater auxiliary memory. Heap Sort maintains a predominantly in-place memory profile but requires more execution time than Quick Sort under the examined conditions. Insertion Sort performs efficiently when data are already ordered but exhibits substantial computational growth for large disordered datasets. Overall, the findings demonstrate that execution time, memory requirements, throughput, and scalability should be considered together when evaluating algorithms for large-scale data processing.

6. CONCLUSION

This study experimentally examined the performance of Quick Sort, Merge Sort, Heap Sort, and Insertion Sort implemented in C under systematically varied dataset sizes and input patterns. The findings demonstrate that algorithmic performance is influenced not only by dataset size but also by the organisation of input data and memory requirements. Quick Sort demonstrated comparatively low execution time and high throughput under random input, while Merge Sort maintained stable execution behaviour with higher auxiliary-memory requirements. Heap Sort provided a predominantly in-place alternative but recorded higher execution times. Insertion Sort performed effectively on ordered data but showed substantial scalability limitations for large disordered datasets.

The results reinforce the importance of evaluating sorting algorithms using multiple performance dimensions rather than relying exclusively on theoretical complexity or execution time. In large-scale data-processing environments, the selection of an algorithm should consider workload characteristics, memory availability, expected dataset growth, and required processing speed. The experimental framework developed in this study provides a structured basis for such comparative evaluation and can be extended to additional algorithms, hardware configurations, and workload conditions.

7. LIMITATIONS

The study has several limitations. First, the experiments were conducted under a single computing environment, and therefore the reported performance values may vary across processors, memory configurations, operating systems, compiler versions, and optimisation settings. Second, the analysis focuses on four classical sorting algorithms and does not include alternative implementations or more recent sorting approaches. Third, the reported benchmark results provide descriptive performance comparisons, while complete trial-level observations are not presented for inferential statistical analysis. Finally, the 10^8-record workload is treated as a stress-test condition and may be influenced by available computational resources.

8. FUTURE RESEARCH

Future research can extend this experimental framework by incorporating additional sorting algorithms,

alternative implementation strategies, and multiple hardware configurations. Comparative studies across different processors, memory architectures, compiler optimisation levels, and operating systems could provide deeper insight into hardware–algorithm interactions. Further research could also examine parallel and multicore sorting techniques, cache behaviour, memory bandwidth, and energy consumption. The inclusion of complete trial-level datasets would additionally permit more detailed statistical analysis, including confidence intervals, effect-size estimation, and appropriate inferential comparisons. Such extensions would strengthen the empirical understanding of algorithm performance under increasingly diverse large-scale computing workloads.

REFERENCES

1. Ala'anzy, M. A., Tolendi, N., Baubek, B., & Algarni, A. (2026). Performance evaluation of GPU-based parallel sorting algorithms. *PLOS ONE, 21*(2), e0342167. https://doi.org/10.1371/journal.pone.0342167
2. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Pearson.
3. Bingmann, T., Marianczuk, J., & Sanders, P. (2021). Engineering faster sorters for small sets of items. *Software: Practice and Experience, 51*(5), 965–1004. https://doi.org/10.1002/spe.2922
4. Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., & Zagha, M. (1998). An experimental analysis of parallel sorting algorithms. *Theory of Computing Systems, 31*(2), 135–167. https://doi.org/10.1007/s002240000083
5. Bryant, R. E., & O'Hallaron, D. R. (2016). *Computer systems: A programmer's perspective* (3rd ed.). Pearson.
6. Codish, M., Bundala, D., Cruz-Filipe, L., Schneider-Kamp, P., & Závodný, J. (2017). Optimal-depth sorting networks. *Journal of Computer and System Sciences, 84*, 185–204. https://doi.org/10.1016/j.jcss.2016.09.004
7. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.
8. Fogli, A., Pietzuch, P., & Giceva, J. (2024). Optimizing sorting for chiplet-based CPUs. In *Proceedings of the 15th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS 2024)*. VLDB Endowment.
9. Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (pp. 285–297). IEEE. https://doi.org/10.1109/SFFCS.1999.814600
10. Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices, 42*(10), 57–76. https://doi.org/10.1145/1297027.1297033
11. Hennessy, J. L., & Patterson, D. A. (2019). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.
12. Hoare, C. A. R. (1962). Quicksort. *The Computer Journal, 5*(1), 10–15. https://doi.org/10.1093/comjnl/5.1.10
13. Islam, M. S., Halder, P., Rahman, S. R., Shakil, S., Kamarudin, N. S., Zaman, O., Islam, M. S., & Al Mamun, M. A. (2026). Cross-language evaluation of sorting algorithms on various input patterns: A comparative study. *Journal of Computer Languages, 87*, 101397. https://doi.org/10.1016/j.cola.2026.101397
14. Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed.). Prentice Hall.
15. Knuth, D. E. (1998). *The art of computer programming, Volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.
16. LaMarca, A., & Ladner, R. E. (1999). The influence of caches on the performance of sorting. *Journal of Algorithms, 31*(1), 66–104. https://doi.org/10.1006/jagm.1998.0985
17. McIlroy, M. D. (1999). A killer adversary for quicksort. *Software: Practice and Experience, 29*(4), 341–344. https://doi.org/10.1002/(SICI)1097-024X(19990410)29:4<341::AID-SPE237>3.0.CO;2-R
18. Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J., & Lomet, D. B. (1995). AlphaSort: A cache-sensitive parallel external sort. *The VLDB Journal, 4*(4), 603–627.

19. Polychroniou, O., & Ross, K. A. (2014). A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (pp. 755–766). ACM. https://doi.org/10.1145/2588555.2610522
20. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.
21. Szefer, J. (2015). Leveraging processor performance counters for security and performance. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices (TrustED '15)* (p. 41). Association for Computing Machinery. https://doi.org/10.1145/2808414.2808421
22. Xiao, L., Zhang, X., & Kubricht, S. A. (2000). Improving memory performance of sorting algorithms. *ACM Journal of Experimental Algorithmics, 5*, Article 3. https://doi.org/10.1145/351827.384245