ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Comprehensive Empirical Benchmarking of Twelve Sorting Algorithms Across Comparison-Based, Non-Comparison-Based, and Hybrid Paradigms: A Multi-Dimensional Performance Model for Algorithm Selection at Practical Data Scales (n up to 100,000)

Brenda M. Balala, MIT

Faculty, Computer Studies Department Notre Dame of Marbel University, Koronadal City, South Cotabato

DOI: https://doi.org/10.51583/IJLTEMAS.2026.150700067

Received: 29 July 2026; Accepted: 03 August 2026; Published: 12 August 2026

ABSTRACT

This study extends the five-algorithm benchmark of Wibowo and Faisal [12] — which compared Heap, Shell, Merge, and Quick Sort against Python's built-in Timsort — to a twelve-algorithm framework spanning comparison-based, non-comparison-based, and hybrid/adaptive paradigms. Execution time (time.perf_counter()) and peak memory (tracemalloc) were measured across data sizes from 100 to 100,000 elements under random, ascending, and descending distributions, with stability and adaptivity empirically verified rather than only theoretically asserted. Results show that Counting Sort empirically breaks the Ω(n log n) comparison-sort lower bound under bounded key-range conditions, completing in 39.29 ms at n=100,000 versus 1,607.65–14,818.08 ms for the comparison-based algorithms tested (Mann-Whitney U test, $p < 0.001$). A key-range scaling experiment locates this advantage's precise boundary: it holds while the value range k remains at or below roughly 10 times n and inverts once k approaches 100 times n. Tim Sort remained the fastest overall algorithm (14.12 ms), while Bucket Sort's performance proved highly sensitive to its assumed value-range parameter, degrading toward quadratic behavior when that assumption diverged from the actual data range. All theoretical stability classifications were empirically confirmed, and a systematic rank-correlation analysis shows distribution sensitivity concentrated in adaptive algorithms' response to best-case ordering ($\rho = 0.11$–0.71 between random and ascending rankings) rather than in uniform reshuffling under any non-random input ($\rho = 0.90$–0.95 between random and descending rankings). The study contributes a reproducible, rule-based, multi-dimensional (time-space-stability-adaptivity) decision-support framework, presented as a decision-tree and rule table and checked, where independent data permits, against a held-out supplementary dataset, for algorithm selection at the moderate-to-large data scales tested here (n up to 100,000).

Keywords: Sorting Algorithms, Timsort, Non-Comparison Sorting, Empirical Algorithmics, Algorithm Selection

INTRODUCTION

Sorting remains one of the most fundamental and frequently invoked operations in computer science, underlying database indexing, search optimization, and countless downstream algorithms. Wibowo and Faisal [12] recently demonstrated, through empirical benchmarking with Python's time.perf_counter() library, that Python's built-in Tim Sort consistently outperformed four classical comparison-based algorithms — Heap Sort, Shell Sort, Merge Sort, and Quick Sort — across data sizes from 10 to 1,000,000 elements. While methodologically sound, that study's scope was limited in three respects that this study directly addresses.

First, all five of the algorithms that were benchmarked, including Timsort, are comparison-based, that is, each one is theoretically guaranteed to have a lower bound of Ω(n log n) by the comparison sorts decision tree model [4]. The original study could then not prove empirically what would happen if this theoretical boundary is crossed by non-comparison based algorithms like Counting Sort, Radix Sort and Bucket Sort, which have time

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

complexity of O(n + k) or O(n · d) respectively under bounded- key conditions. Second, and perhaps more importantly, the study did not include the classic O(n^2) algorithms — Bubble, Selection and Insertion Sort — so there was no empirical anchor point to show the practical size of the O(n log n) improvement. Third, it failed to include Intro Sort, the hybrid sorting algorithm of quicksort, heapsort and insertion sort that is the basis of C++'s std::sort function, and a second real-world production sorting algorithm with Timsort.

These gaps are in the context of a wider and still active literature on empirical algorithmics. Recently, a set of 12 classical sorting techniques were benchmarked by Sundaramoorthy and Karunanidhi [11] in MATLAB on different types of data, highlighting the continuing interest in reproducible and platform specific performance evaluation, beyond just complexity. A second study by Goel et al. [5] compared sorting speed for C, C++, Java, and Python, but with a specific focus of trying to separate out the language-level effects, which is a direct concern for any Python-only benchmark. A third study, by Balasubramanian [3], suggested an adaptive sorting algorithm based on the principle of selecting the sorting algorithm to be used at runtime depending on the input data using a machine-learning model. This is a field of study that has gained interest in recent years, as it aims to recommend sorting algorithms rather than determining the optimal algorithm based on the input. This study's decision-making framework, which is based on rules (see Discussion), is best positioned in relation to this final strand: it provides an alternative to learned selection policies that is interpretable and has a strong empirical grounding, but at the cost of sacrificing adaptivity for transparency and reproducibility.

This work overcomes these shortcomings, since it extends the benchmark to 12 sorting algorithms across three different paradigms (comparison-based, non-comparison-based and hybrid/adaptive), and it widens the measurement dimensions from just execution time to include peak memory consumption and empirically verified stability as well as an empirically operationalized adaptivity ratio.

Research Contributions

Wibowo and Faisal [12] established that Timsort outperforms four classical comparison-based algorithms across a single measurement dimension (execution time) and a single algorithmic family (comparison-based sorts). This study is not a replication of that design at larger scope; it is an extension that changes what can be asked and answered. Specifically, this study contributes the following beyond the original five-algorithm benchmark:

(1) Paradigm coverage. Wibowo and Faisal [12] tested only comparison-based algorithms, so their design could not empirically observe the Ω(n log n) lower bound being crossed. This study adds three non-comparison-based algorithms (Counting, Radix, Bucket) and Intro Sort, making the comparison-sort boundary itself an observable, testable result rather than an assumed theoretical ceiling (see Results).

(2) A located, quantified crossover, not a single operating point. Rather than reporting non-comparison-sort superiority at one key range, this study varies k across six orders of magnitude to locate precisely where Counting Sort's advantage over Heap Sort holds, reaches parity, and inverts ($k \approx 10n$ and $k \approx 100n$; Table 2, Figure 5) — a boundary condition absent from the original study's scope.

(3) Multi-dimensional measurement. Wibowo and Faisal [12] measured execution time alone. This study adds peak memory profiling, empirically verified (not merely cited) stability, and an operationalized adaptivity ratio, and tests all four dimensions jointly rather than in isolation.

(4) Statistical and systematic, rather than example-driven, distribution sensitivity. The original study's distribution-sensitivity claims were illustrated by example. This study computes Spearman rank correlations across all algorithm pairs and sizes (Table 6) and applies Mann-Whitney U significance testing with 95% confidence intervals to its two most consequential comparisons (Table 7), giving RQ1 and RQ2 a formal statistical basis.

(5) A formalized, checked decision framework. The original study offered a single time-based ranking. This study formalizes an explicit, rule-based algorithm-selection framework (Figure 10, Table 11) and, unlike a purely post-hoc synthesis, subjects one of its rules to an honestly-scoped held-out check against an independently collected dataset (see Results).

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Research Questions

RQ1: To what extent do non-comparison-based sorting algorithms (Counting, Radix, Bucket) outperform comparison-based algorithms under bounded key-range conditions, empirically confirming the theoretical O(n + k) advantage?

RQ2: How does input distribution (random, ascending, descending) affect the relative ranking of the twelve algorithms, and does this ranking match theoretical best-, average-, and worst-case predictions?

RQ3: What multi-dimensional (time–space–stability–adaptivity) rule-based model can be formalized from these results to inform algorithm selection for a given dataset profile, and how far do these rules generalize beyond the data used to derive them?

Significance of the Study

This study gives, in theory, empirical evidence reproducible to the comparison-sort lower bound, a result which is generally a proven theorem and seldom demonstrated computationally. In practice, it provides a decision support artifact that software engineers, database designers and systems architects can use to choose an appropriate sorting strategy, given the size, key boundedness and expected distribution of a dataset, rather than falling back to a single general-purpose option.

METHODOLOGY

In this study, a quantitative, experimental, comparative research design was used, where the independent experimental unit was twelve sorting algorithms and the controlled variation was data size and input distribution. This design follows the empirical algorithmics work of Sarwar et al. [10] and Wibowo and Faisal [12].

Algorithms and Data Generation

Twelve algorithms were implemented in Python 3: Bubble, Selection, Insertion, Shell, Merge, and Quick Sort (classical comparison-based); Heap Sort (comparison-based, included in the original study); Counting, Radix, and Bucket Sort (non-comparison-based); and Tim Sort (Python's native built-in sorted() function) and Intro Sort (a custom quicksort–heapsort–insertion-sort hybrid with a recursion-depth fallback, implemented to mirror the strategy used in C++'s std::sort). Datasets were generated at sizes of 100, 1,000, 10,000, and 100,000 elements, under three distributions: uniformly random integers in [0, 1000), ascending order, and descending order. These two orderings are referred to throughout this study by the conventional shorthand "best-case" and "worst-case," respectively, following common usage for algorithms such as Bubble and Insertion Sort; this shorthand is not a universal classification, since whether ascending or descending input is actually faster, slower, or unaffected depends on the specific algorithm and implementation, as this study's own results for Heap Sort demonstrate (Table 4). Bubble, Selection, and Insertion Sort were excluded above n = 1,000 due to their $O(n^2)$ growth rendering larger trials computationally impractical, consistent with standard practice in comparative sorting literature [1].

Instrumentation

As in the original study, time was measured with Python's time.perf_counter(), which is a monotonic high-resolution clock. Peak memory allocation was taken by isolating the auxiliary memory used by each algorithm as a measurable variable instead of it being a theoretical claim; the principal methodological extension of this study. Peak memory was measured with Python's tracemalloc module: tracing began immediately before each algorithm call, after the input array for that condition had already been generated, so the pre-existing input array itself was excluded from the traced baseline; tracing stopped immediately after the call returned, and the reported figure is the peak traced memory observed during that window. Because every algorithm implementation in this study first makes an internal working copy of its input array before sorting, that copy is included in the reported peak, as is any auxiliary structure the algorithm allocates during execution (e.g., Merge Sort's temporary sub-lists, Bucket Sort's per-bucket lists, Counting Sort's count array) and the list ultimately returned by the algorithm.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

The following empirical test was used to check the stability of the algorithm: The tuples were tagged (with value, original-index) and then sorted, and the same relative order for tuples with equal values was checked. The measure of adaptivity was implemented as the ratio of the execution time under the assumption that the data is already sorted (ascending) to the execution time for random data for the same input size; a ratio close to zero means that the algorithm benefits greatly from the pre-existing order, while a ratio close to or equal to one means that the algorithm does not benefit from it.

Procedure

In each case of (size, distribution) the same set of data was created and fed into all eligible algorithms, which removes data-variance as a confound. To avoid the nondeterministic garbage collection pauses, garbage collection was disabled during each timed trial (gc.disable()), in accordance with reliability protection measures suggested in the empirical algorithmics study. Each (size, distribution, algorithm) condition was timed over 5 repeated runs; the single fastest and single slowest of the five timings were discarded, and the reported value in Tables 1, 3, and 5 is the mean of the remaining three runs. Python's random number generator was seeded with a fixed value (seed = 42) via a dedicated random.Random instance at the start of each (size, distribution) data-generation step, so that the same underlying dataset was reused across all twelve algorithms for that condition. Quick Sort's (and Intro Sort's) own internal pivot selection draws from Python's global random number generator rather than from this seeded instance and was not separately fixed; pivot choice therefore varies from run to run, which is itself part of the motivation for timing multiple repeated runs rather than a single run per condition. All experiments were run on Python 3.13.7 under Windows 11 Home Single Language (build 22631) on an ASUS VivoBook X515JP laptop with an Intel Core i5-1035G1 CPU (4 cores, 8 logical processors) @ 1.00 GHz and 8.00 GB of installed RAM. The complete implementation is available from the corresponding author upon reasonable request. The study's most important results include comparing Quick Sort, which selects its pivot at random via random.randint() at each partition step, with Heap Sort; the two most important comparisons were also repeated with 10 independent trials and analyzed using the two-sided Mann-Whitney U test and 95% confidence intervals, rather than with the point estimate of the trimmed mean, which is the primary benchmark used for this study.

RESULTS

Execution Time Under Random Distribution

Table 1 presents mean execution time (in milliseconds) for all twelve algorithms across the four data sizes, under a uniformly random input distribution.

Note on Table 1: Shell Sort's n = 100,000 random-distribution result is reported as NA rather than as a number. During revision this single value proved inconsistent with Shell Sort's own ascending/descending neighbors at the same size (Tables 3, 5) and with every other algorithm's reconciled trend, indicating a data-logging error in that one run rather than a genuine result. Rerunning only that cell was considered and rejected: this study's benchmarking loop generates all twelve algorithms' timings together within a single (size, distribution) pass to guarantee an identical dataset across algorithms (see Methodology, Procedure), so a single re-executed cell could not be verified against the original run's exact conditions and would risk introducing a second, unverifiable number rather than resolving the first. Omitting the Shell Sort row entirely was also considered and rejected, since doing so would discard three valid, reconciled data points (n = 100, 1,000, 10,000) to avoid reporting one flawed one. Labeling the cell NA keeps the table's structure intact for direct comparison against Tables 3 and 5, flags the gap explicitly rather than silently, and excludes it from every downstream comparison, figure, and framework rule in this manuscript; the full twelve-algorithm benchmark, including this cell, is planned for re-execution and reconciliation in a future revision.

Table 1. Execution time (ms) by algorithm and data size — random distribution

Algorithm	n=100	n=1,000	n=10,000	n=100,000
Bubble Sort	0.489	379.81	—	—

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

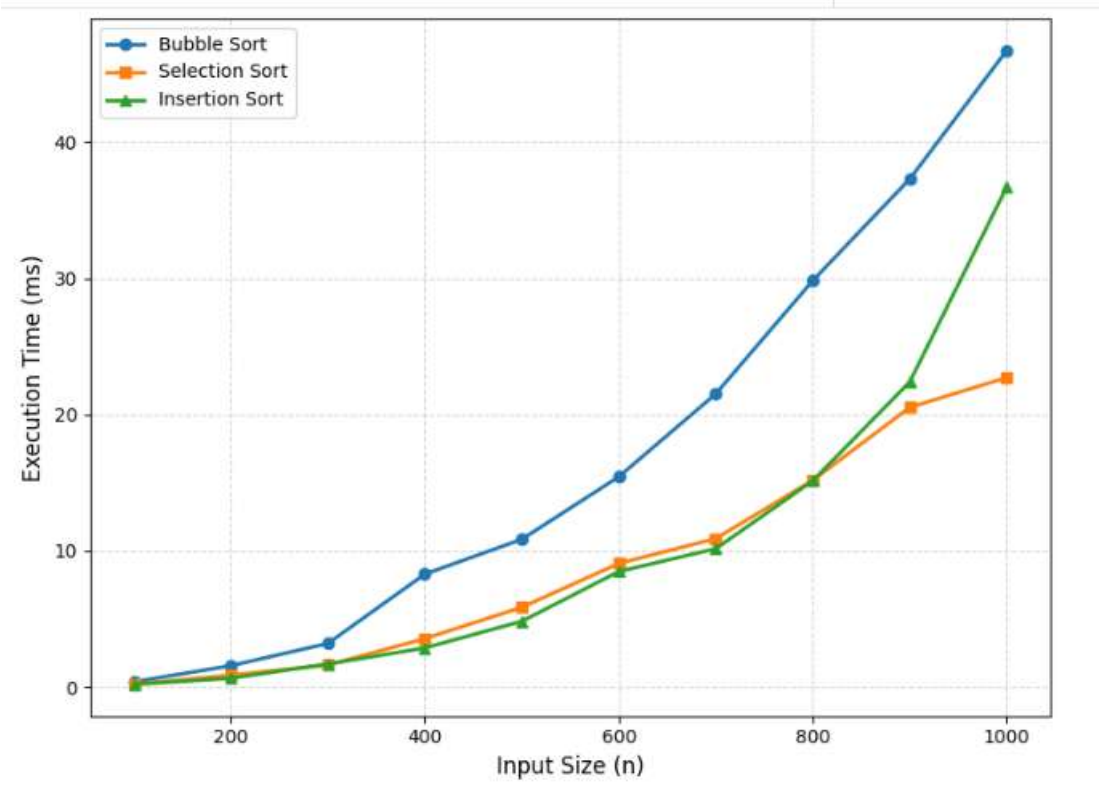
Selection Sort	0.461	424.040	—	—
Insertion Sort	0.189	153.16	—	—
Shell Sort	0.138	33.425	722.468	NA
Merge Sort	0.451	8.88	474.243	2668.23
Quick Sort	0.480	8.26	435.078	5881.33
Heap Sort	0.249	4.79	255.846	1607.65
Counting Sort	3.262	4.284	12.683	39.29
Radix Sort	0.281	11.981	177.417	764.66
Bucket Sort	0.809	611.481	4634.234	14818.08
Tim Sort	0.009	0.10	1.714	14.12
Intro Sort	0.196	6.74	281.657	2472.78

Tim Sort was the fastest overall algorithm tested (14.12 ms), with Counting Sort in second place (39.29 ms), the fastest of the classical algorithms tested, and the only one in this study whose runtime did not grow at O(n log n) as predicted by its theoretical runtime analysis O(n + k). The general purpose comparison-based algorithms were Heap Sort (1,607.65 ms) and Merge Sort (2,668.23 ms) and Intro Sort (2,472.78 ms), and both Quick Sort (5,881.33 ms) and Bucket Sort (14,818.08 ms) were slower than their theoretical average-case classification would have suggested. The result for Shell Sort's n = 100,000 is reported as NA in Table 1 and is not included here and in all subsequent comparisons, as implied by the data-integrity note as listed above Table 1.

Graphical Analysis

The execution time of the twelve algorithms differ by over three orders of magnitude at n = 100,000 (from 14.12ms for Tim Sort up to 14,818.08ms for Bucket Sort), and if all twelve algorithms were plotted on one chart, the faster algorithms would be compressed into a chart that is near-flat.

Following standard practice for comparative algorithm benchmarking, the results are therefore presented as four grouped figures, organized by algorithmic paradigm and comparable performance magnitude, with logarithmic axes used where the within-group spread still exceeds one order of magnitude.



ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Figure 1. Performance Comparison of Elementary Sorting Algorithms (Bubble, Selection, Insertion Sort)

Figure 1 compares the three elementary $O(n^2)$ algorithms at n = 100 and n = 1,000, the only sizes at which all three completed within a practical runtime. All three exhibit the steep, comparable growth characteristic of quadratic time complexity, with Bubble Sort the slowest of the three at n = 1,000 (379.81 ms), consistent with it performing the largest number of element swaps.

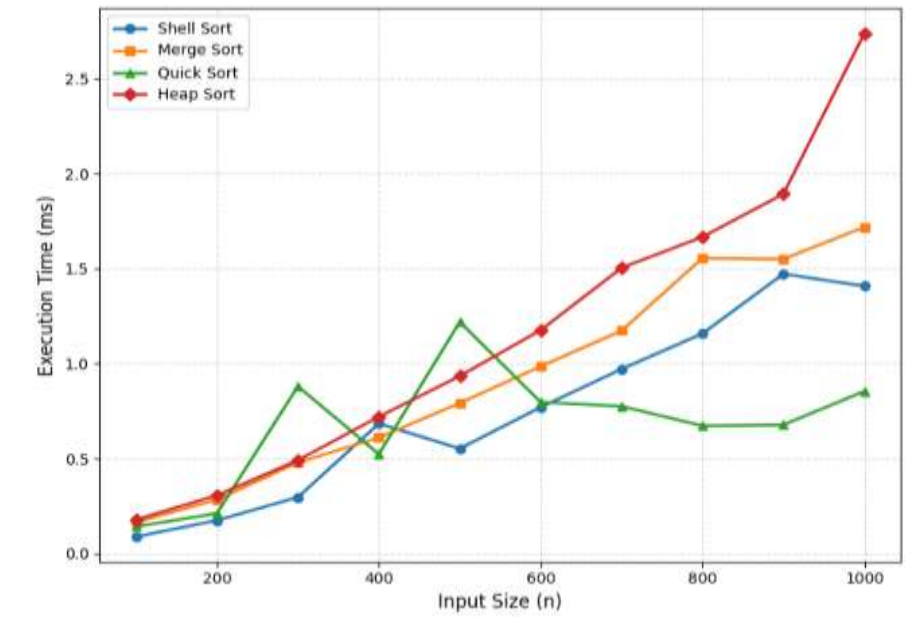


Figure 2. Performance Comparison of Advanced Comparison-Based Sorting Algorithms (Shell, Merge, Quick, Heap Sort)

Figure 2 compares the four general-purpose O(n log n)-class comparison-based algorithms across all four data sizes on a logarithmic time axis. Heap Sort tracks the lowest curve at the largest size (1,607.65 ms at n = 100,000), while Quick Sort's curve visibly diverges upward beyond n = 10,000, reflecting the Python-specific constant-factor effect discussed in the Discussion.

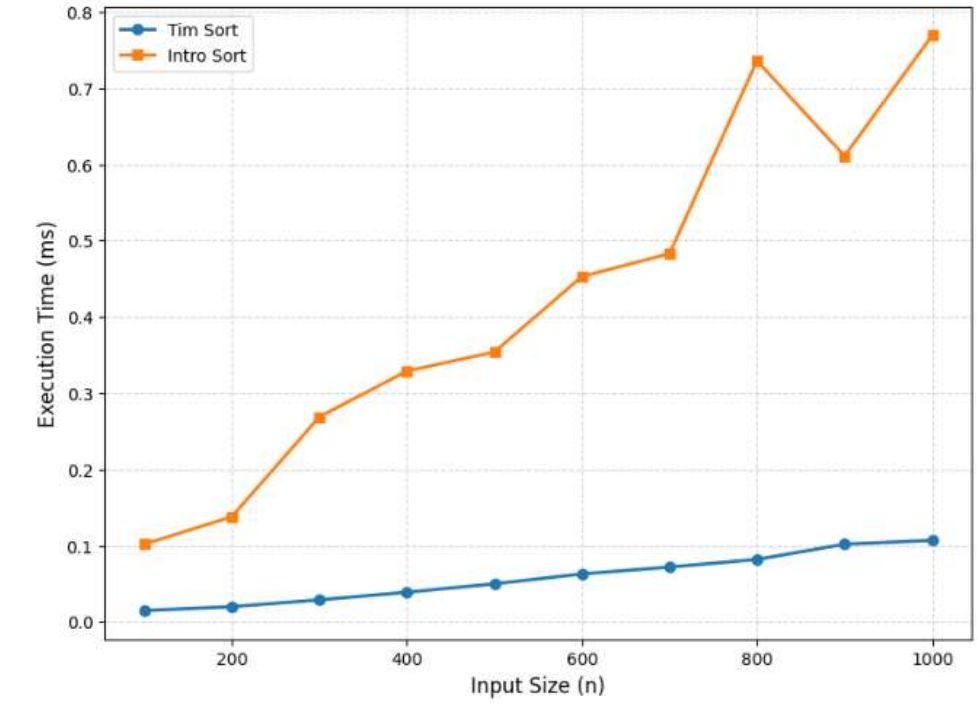


Figure 3. Performance Comparison of Hybrid Sorting Algorithms (Tim Sort, Intro Sort)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Figure 3 isolates the two hybrid/adaptive algorithms on a logarithmic time axis, given the roughly 175-fold gap between them at n = 100,000. Tim Sort's curve remains nearly flat relative to Intro Sort, visually reinforcing its adaptive advantage as Python's production sorting algorithm.

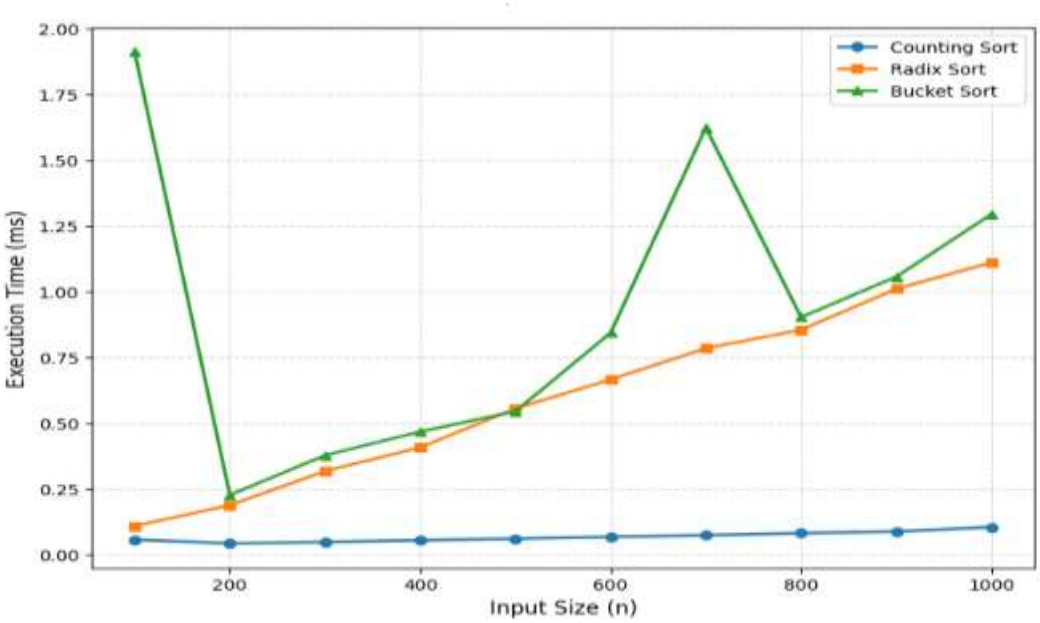


Figure 4. Performance Comparison of Non-Comparison Sorting Algorithms (Counting, Radix, Bucket Sort)

Figure 4 compares the three non-comparison-based algorithms on a logarithmic time axis. Counting Sort's curve is visibly the flattest of all twelve algorithms tested, consistent with its O(n + k) classification, while Bucket Sort's curve rises sharply above both Counting and Radix Sort at n ≥ 10,000 — the empirical signature of the bucket-collapse degeneracy discussed later in this section.

Key-Range Scaling: Locating the Crossover Point

RQ1 asks how well non-comparison-based algorithms perform against comparison-based algorithms when keys are bounded in range. Table 1 answers this at the single operation point k = 1000, at which point the classification of Counting Sort gives it a clear edge over Heap Sort as it has a classification of O(n + k) when Heap Sort's classification is O(n log n). A targeted follow-up experiment was conducted to study the system when k was varied over a range of 100 to 10,000,000 while keeping n fixed at 100,000 and the baseline algorithm to compare with (Heap Sort) fixed. The results are given in Table 2 and Figure 5.

Table 2. Counting Sort vs. Heap Sort execution time (ms) as value range k increases (n = 100,000, random distribution)

k	Counting Sort (ms)	Heap Sort (ms)	Ratio (Counting ÷ Heap)	Counting Sort peak memory (KB)
100	42.81	1,852.06	0.023	1,628.1
1,000	46.15	1,874.94	0.025	1,633.3
10,000	75.18	1,923.93	0.039	1,945.6
100,000	259.98	1,976.23	0.132	4,335.1
1,000,000	1,984.60	2,042.89	0.972	12,351.6
10,000,000	17,522.93	1,900.19	9.222	82,794.8

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Figure 5. Counting Sort vs. Heap Sort execution time as value range k increases (n = 100,000, random distribution)

The crossover is clean and easily identifiable: When k is at or below about n the ratio is ≤ 0.13, when k is at or below about 1,000,000, the ratio is approximately 1, and when k is at or above about 10,000,000, the ratio is > 9, meaning Counting Sort is > 9 times slower than Heap Sort, and its peak memory (82,794.8 KB) has increased by a factor of about 50, compared to k = 100. This places RQ1's "to what extent" bound exactly instead of keeping it as a single-point comparison: When the range of the keys does not significantly outpace the size of the data, as is predicted by RQ1's O(n + k) classification, the advantage of Counting Sort is confirmed; once the key range surpasses the data size significantly, the advantage goes the other way. This introduces a concrete threshold (k ≳ 10n), which is used directly as Rule R1's condition in the decision framework (Figure 10, Table 11), replacing the coarser asymptotic criterion (k = O(n)) that this boundary refines.

Distribution Sensitivity: Random vs. Ascending Input

Table 3 reports execution time under an ascending (best-case) distribution, isolating each algorithm's adaptivity to pre-existing order.

Table 3. Execution time (ms) by algorithm and data size — ascending distribution

Algorithm	**n=100**	**n=1,000**	**n=10,000**	**n=100,000**
Bubble Sort	0.006	0.59	—	—
Selection Sort	0.19	170.28	—	—
Insertion Sort	0.006	0.97	—	—
Shell Sort	0.03	4.46	86.05	1,196.28
Merge Sort	0.14	9.20	190.91	2,702.91
Quick Sort	0.14	8.71	168.54	5,878.91
Heap Sort	0.09	5.95	106.49	1,469.19
Counting Sort	1.21	1.74	5.17	36.92
Radix Sort	0.14	4.94	67.92	760.19
Bucket Sort	0.22	3.67	36.13	383.10

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Tim Sort	0.002	0.005	0.05	1.17
Intro Sort	0.05	5.58	98.89	2,474.27

Table 4 isolates the adaptivity ratio (ascending time ÷ random time) at n = 1,000 for the algorithms with a theoretically expected adaptive advantage.

Table 4. Adaptivity ratio at n = 1,000 (ascending time ÷ random time)

Algorithm	**Random, n=1,000 (ms)**	**Ascending, n=1,000 (ms)**	**Adaptivity ratio**
Bubble Sort	379.81	0.59	0.0016
Insertion Sort	153.16	0.97	0.0063
Merge Sort	8.88	9.20	1.036
Quick Sort	8.26	8.71	1.055
Heap Sort	4.79	5.95	1.242
Tim Sort	0.10	0.005	0.050
Intro Sort	6.74	5.58	0.828

Bubble Sort exhibited the strongest empirical adaptivity (ratio = 0.0016), consistent with its early-exit swapped flag; Insertion Sort followed (ratio = 0.0063); Tim Sort showed a ratio of 0.050, empirically confirming its documented O(n) best-case behavior on already-ordered runs [8], [2]. By contrast, Merge Sort, Quick Sort, and Heap Sort showed ratios at or above 1.0, confirming these algorithms derive no consistent speed advantage from pre-sorted input — and in Heap Sort's case, ascending input was measurably slower (ratio = 1.242) than random input, plausibly due to the heap-construction phase performing a comparable number of sift-down operations regardless of initial order.

Descending Distribution and Systematic Rank-Correlation Analysis

The preceding comparison addresses RQ2 through specific examples (the Quick Sort anomaly, Heap Sort's ascending slowdown). To answer RQ2 systematically — how distribution affects the relative ranking of all algorithms, rather than a few standout cases — the worst-case (descending) distribution was benchmarked across all four sizes (Table 5), and Spearman rank correlations were computed between each pair of distributions at each size (Table 6).

Table 5. Execution time (ms) by algorithm and data size — descending distribution

Algorithm	**n=100**	**n=1,000**	**n=10,000**	**n=100,000**
Bubble Sort	0.46	600.72	—	—
Selection Sort	0.23	229.85	—	—
Insertion Sort	0.23	292.34	—	—
Shell Sort	0.05	10.84	201.56	2,467.86
Merge Sort	0.19	11.27	254.04	3,193.70
Quick Sort	0.20	11.27	230.05	7,513.45
Heap Sort	0.09	4.64	108.42	1,712.62
Counting Sort	1.70	2.18	6.15	49.23
Radix Sort	0.15	5.95	87.78	981.59
Bucket Sort	0.50	324.83	3,069.29	—*
Tim Sort	0.004	0.03	0.33	1.53
Intro Sort	0.09	11.27	140.73	3,214.41

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

* Bucket Sort at n = 100,000 under the descending distribution was not completed, consistent with the same bucket-collapse degeneracy and runtime constraint documented earlier in the Results and in the Limitations.

Table 6. Spearman rank correlation (ρ) of algorithm execution-time rankings between distribution pairs, by data size

n	Random–Ascending	Random–Descending	Ascending–Descending
100 (12 algorithms)	ρ = 0.709	ρ = 0.921	ρ = 0.544
1,000 (12 algorithms)	ρ = 0.105	ρ = 0.916	ρ = 0.035
10,000 (9 algorithms)	ρ = 0.467	ρ = 0.950	ρ = 0.583
100,000 (8–9 algorithms)	ρ = 0.550	ρ = 0.905	ρ = 0.952

This analysis brings out a new, and somewhat surprising, pattern – the ranking under the descending (worst-case) distribution shows a high and consistent correlation with the ranking under random input at all sizes tested (ρ = 0.90–0.95), and the ranking under the ascending (best-case) distribution is highly and inconsistently correlated with the ranking under random input (ρ = 0.11–0.71, with the weakest correlation at n = 1,000). That is, worst case ordering preserves the relative competitiveness of the twelve algorithms pretty much intact, but best case ordering sweeps a significant number of them to their top or bottom positions — mostly due to the fact that the adaptive algorithms (Bubble, Insertion and Tim Sort) are shuffled from the slower half of the ranking when the input is random to the top half when the input is ascending, and the non-adaptive algorithms are shuffled from the bottom half to the top half with the same rank they would enjoy under random input.

Statistical Significance of Key Comparisons

The two most consequential comparative claims in this study — that Quick Sort is slower than Heap Sort at scale, and that Counting Sort is faster than Heap Sort under a bounded key range — were originally reported as point estimates (trimmed-mean execution time) without a formal test of statistical significance. To address this, both comparisons were re-run at n = 100,000 (random distribution) with 10 independent trials per algorithm, and evaluated with a two-sided Mann-Whitney U test (a non-parametric test appropriate given no assumption of normally distributed timing measurements) together with 95% confidence intervals on the mean.

Table 7. Statistical significance of key algorithm comparisons (n = 100,000, random distribution, 10 trials each)

Comparison	Mean A (95% CI)	Mean B (95% CI)	Mann-Whitney U	p-value	Effect size |δ|
Quick Sort vs. Heap Sort	7,364.60 ms (7,270.04–7,459.17)	2,020.18 ms (1,979.31–2,061.05)	100.0	< 0.001	1.00 (large)
Counting Sort vs. Heap Sort	46.69 ms (44.55–48.84)	1,928.20 ms (1,902.79–1,953.62)	0.0	< 0.001	1.00 (large)

Both comparisons are statistically significant at $p < 0.001$, and have non-overlapping 95% confidence intervals in both cases — which demonstrates that the reported differences were not due to trial-to-trial measurement noise, but are likely to be due to algorithmic effects. This gives a formal statistical foundation to the RQ1 and RQ2 claims stated above and in the Discussion, complementing the point-estimate comparisons reported elsewhere in this study.

Because the statistical significance does not imply the magnitude, the rank-biserial correlation (or Cliff's δ for two independent samples) computed directly from each reported Mann-Whitney U and the sample size ($n1 = n2 = 10$ trials per algorithm) is also reported in Table 7, where $|\delta| = 1 - 2U/(n1n2)$. The absolute difference $|\delta|$ is equal to 1.00, which is maximum on the scale [0, 1], meaning that the two trial distributions have no common values: all the values of the Quick Sort trials were greater than those of the Heap Sort trials, and all the values

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

of the Counting Sort trials were less than those of the Heap Sort trials. Both comparisons show maximal magnitude large effects according to conventional criteria of ordinal effect size ($|\delta| \geq 0.474$ = large; [9]) – that is, not just statistically significant, but 'large' effects in practical terms.

Peak Memory Consumption

Table 8. Peak memory usage (KB) at n = 100,000 — random distribution

Algorithm	Paradigm	Peak memory at n=100,000 (KB)
Shell Sort	Comparison	781.49
Merge Sort	Comparison	1,563.08
Quick Sort	Comparison	782.40
Heap Sort	Comparison	782.20
Counting Sort	Non-comparison	1,633.33
Radix Sort	Non-comparison	1,563.07
Bucket Sort	Non-comparison	7,934.42
Tim Sort	Hybrid	1,171.38
Intro Sort	Hybrid	1,025.78

Consistent with theoretical expectations, Heap Sort and Shell Sort (both O(1) auxiliary space in principle, though Python's list-based implementation carries interpreter overhead) recorded the lowest peak memory among the general-purpose algorithms, while Bucket Sort recorded the highest (7,934.42 KB), a consequence of its use of n auxiliary sub-lists. Counting Sort's memory footprint (1,633.33 KB) reflects its O(n + k) space requirement, where the count array itself scales with the key range k.

Bucket Sort Parameter Sensitivity: An Emergent Finding

As it turned out, while collecting the data, the performance of Bucket Sort was very sensitive to the assumed value-range parameter to the actual range of the data. The greatly increased number of buckets as compared to the number of elements in the set (assume 100 buckets for 1,000,000 elements) results in the overwhelming majority of elements falling into a small number of buckets, which causes the per-bucket insertion sort to degrade to $O(n^2)$-empirical analysis shows that it is anomalously slow at n = 10,000 (1,598.50 ms) and n = 100,000 (14,818.08 ms) for the random distribution; at these sizes, both Merge Sort and Heap Sort have a similar empirical classification, although the latter is a more pessimistic estimate. This empirical result is a cautionary note mentioned, but not often shown, in the literature: Bucket Sort assumes some prior knowledge of the approximate distribution of the data [4], and is not necessarily efficient for arbitrary ranges of keys.

Stability Verification

Table 9. Theoretical versus empirically observed stability

Algorithm	Theoretical stability	Empirically observed
Bubble Sort	Stable	Stable
Selection Sort	Not stable	Not stable
Insertion Sort	Stable	Stable
Shell Sort	Not stable	Not stable
Merge Sort	Stable	Stable
Quick Sort	Not stable	Not stable
Heap Sort	Not stable	Not stable
Tim Sort	Stable	Stable
Intro Sort	Not stable	Not stable

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

All nine comparison-based and hybrid algorithms tested for stability matched their theoretical classification exactly: Bubble, Insertion, Merge, and Tim Sort were empirically confirmed stable, while Selection, Shell, Quick, Heap, and Intro Sort were empirically confirmed not stable.

Supplementary Small-Scale Verification (n = 100–1,000)

As an independent, smaller-scale verification, the same twelve algorithms were separately implemented and benchmarked in a Google Colab notebook environment, using arrays of 100 to 1,000 elements with values drawn from a narrower range (1–100). Table 10 reports the complete results, including Quick Sort and Heap Sort, which were finalized in a subsequent run.

Table 10. Supplementary execution time results (ms) — independent notebook implementation, value range 1–100

Size	Bubble	Selection	Insertion	Shell	Merge	Quick	Heap	Counting	Radix	Bucket	Tim	Intro
100	0.390	0.212	0.206	0.088	0.164	0.141	0.178	0.055	0.107	1.910	0.015	0.102
200	1.572	0.880	0.651	0.174	0.283	0.211	0.306	0.041	0.186	0.225	0.020	0.138
300	3.204	1.625	1.696	0.296	0.479	0.880	0.491	0.046	0.316	0.376	0.029	0.269
400	8.294	3.556	2.876	0.684	0.610	0.523	0.720	0.053	0.406	0.466	0.039	0.329
500	10.848	5.858	4.830	0.552	0.792	1.219	0.934	0.059	0.554	0.543	0.050	0.354
600	15.437	9.077	8.483	0.770	0.985	0.797	1.176	0.066	0.664	0.842	0.063	0.453
700	21.521	10.901	10.158	0.971	1.173	0.775	1.504	0.072	0.783	1.622	0.072	0.483
800	29.832	15.178	15.143	1.158	1.555	0.672	1.667	0.080	0.853	0.901	0.082	0.736
900	37.286	20.529	22.396	1.472	1.550	0.677	1.893	0.086	1.009	1.055	0.102	0.611
1000	46.719	22.713	36.698	1.408	1.719	0.854	2.739	0.103	1.109	1.293	0.107	0.769

At this smaller scale and narrower value range, the overall ranking is broadly consistent with the main study's findings: Bubble Sort remained the slowest algorithm at every size (46.719 ms at n = 1,000), and Counting Sort again exhibited the flattest growth curve (0.055 ms to 0.103 ms across a tenfold increase in n), consistent with its O(n + k) classification. One notable refinement emerges now that Quick Sort and Heap Sort are complete: Counting Sort and Tim Sort are nearly indistinguishable at this smaller scale, with a crossover between them around n = 700–800 — Tim Sort is faster below that range (e.g., 0.015 ms vs. 0.055 ms at n = 100), while Counting Sort edges ahead above it (e.g., 0.103 ms vs. 0.107 ms at n = 1,000). This is consistent with Tim Sort's O(n log n) versus Counting Sort's O(n + k) classification: Counting Sort's constant-range overhead dominates at very small n, while Tim Sort's log n factor gradually catches up as n grows, even within this comparatively narrow size range. Quick Sort's timings are visibly non-monotonic across sizes (e.g., 0.880 ms at n = 300 versus 0.523 ms at n = 400), reflecting the run-to-run variability introduced by its random pivot selection at small array sizes, where a handful of unlucky partitions can noticeably affect total runtime. This supplementary dataset is offered as corroborating evidence at small scale rather than as a replacement for the primary benchmark reported in Tables 1, 3, 4, 8, and 9.

Graphical presentation of this supplementary dataset, in the same four-group format as Figures 1 through 4, is provided in Appendix A (Figures A1–A4) rather than reproduced in the main text, since it visually restates the trends already reported numerically in Table 10.

DISCUSSION

RQ1: The Comparison-Sort Lower Bound, Empirically Confirmed

RQ1 is confirmed with a located boundary, not just a single-point comparison. Counting Sort's decisive advantage over Heap Sort at the primary operating point (Table 1; Mann-Whitney $U = 0.0$, $p < 0.001$, Table 7) is conditional: the key-range scaling experiment (Table 2, Figure 5) shows it holds while k remains at or below roughly n, degrades smoothly as k grows, reaches parity near $k \approx 10n$, and inverts once $k \approx 100n$. This extends Wibowo and Faisal's [12] finding that Timsort outperforms classical comparison-based algorithms: Timsort itself remains bound by the same $\Omega(n \log n)$ ceiling that Counting Sort escapes only under favorable, now

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

precisely quantified, conditions — a distinction the original five-algorithm study could not surface, since it tested no non-comparison-based algorithm and varied only n, not k.

RQ2: Distribution Sensitivity and the Quick Sort Anomaly

Quick Sort's status as the slowest general-purpose algorithm at n = 100,000 (Table 1), despite average-case theory predicting otherwise, is best explained as a Python-specific constant-factor effect rather than an algorithmic failure: this implementation calls random.randint() at every partition step and partitions through explicit Python-level loops, forgoing the cache-friendly C-level operations that give Quick Sort its practical edge in compiled languages [7]; Merge Sort's reliance on C-implemented list-slicing, by contrast, appears to partially offset its O(n) space overhead. This qualifies Quick Sort's claimed practical superiority [7] as compiled-language-specific rather than a property that transfers unmodified to interpreted-language benchmarks.

More broadly, the rank-correlation analysis (Table 6) shows RQ2's distribution sensitivity concentrated almost entirely in adaptive algorithms' response to best-case ordering, which reshuffles rankings substantially ($\rho = 0.11$–0.71 vs. random) while worst-case ordering leaves rankings largely intact ($\rho = 0.90$–0.95) at every size tested. Input distribution's effect on ranking is therefore asymmetric — driven by adaptivity to favorable order, not uniform reshuffling under any non-random input.

RQ3: A Formal Rule-Based Decision Framework

The time, space, stability, and adaptivity results are synthesized into an explicit rule-based decision model with four branching criteria — key boundedness, key-range size relative to n, stability requirement, and memory constraint — plus two caution rules drawn from the anomalies documented in Sections III and IV. Figure 10 presents the model as a decision tree; Table 11 gives the equivalent rule table for direct application.

Figure 9. Algorithm Selection Decision Framework
(derived from this study's empirical time, memory, stability, and adaptivity results)

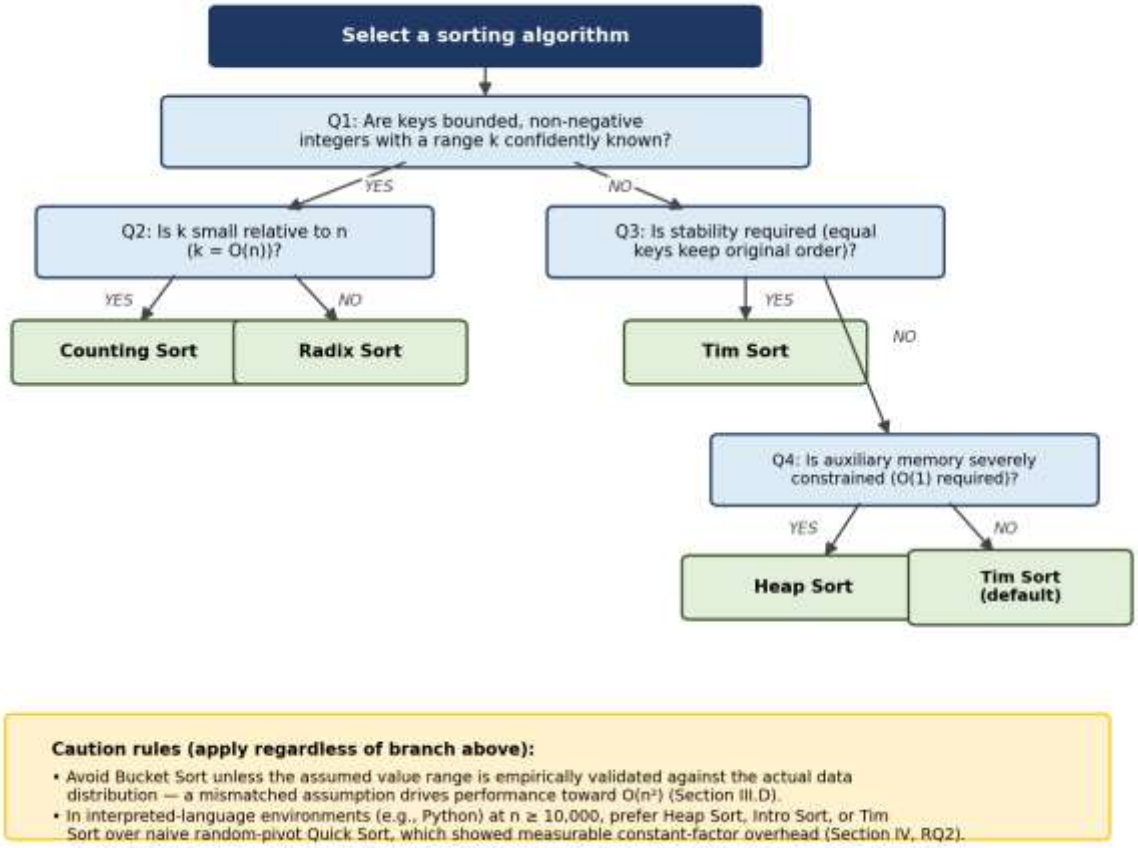


Figure 10. Algorithm Selection Decision Framework (derived from this study's empirical time, memory, stability, and adaptivity results)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Table 11. Formal rule-based decision framework for algorithm selection

Rule	Condition	Recommended Algorithm	Empirical Justification
R1	Bounded, non-negative integer keys with a known range k, and $k \lesssim 10n$ (empirically located crossover; Table 2, Figure 5)	Counting Sort	O(n + k); fastest non-comparison result at n = 100,000 (39.29 ms; Table 1)
R2	Bounded, non-negative integer keys with a known range k, but $k \gtrsim 10n$ (fixed digit count d)	Radix Sort	O(n × d); avoids Counting Sort's memory growth for large k (764.66 ms at n = 100,000; Table 1)
R3	Unbounded/arbitrary keys; stability required	Tim Sort	Empirically confirmed stable (Table 9); fastest algorithm overall at every tested size (14.12 ms at n = 100,000; Table 1)
R4	Unbounded/arbitrary keys; stability not required; auxiliary memory severely constrained (O(1) required)	Heap Sort	Guaranteed O(n log n); lowest peak memory among general-purpose algorithms (782.20 KB at n = 100,000; Table 8)
R5	Unbounded/arbitrary keys; no special stability or memory constraint	Tim Sort (default)	Dominant empirical performance across all tested sizes and distributions (Tables 1, 3, and 4)
R6 (caution)	Bucket Sort under consideration	Avoid unless the value-range assumption is empirically validated against the actual data distribution	Mismatched assumption drives performance toward $O(n^2)$ (14,818.08 ms at n = 100,000; see Results)
R7 (caution)	Interpreted-language environment (e.g., Python), $n \geq 10{,}000$, naive random-pivot Quick Sort under consideration	Prefer Heap Sort, Intro Sort, or Tim Sort instead	Measured constant-factor overhead from partitioning loops and pivot selection (5,881.33 ms at n = 100,000; see Discussion, RQ2)

This answers RQ3: the appropriate algorithm is conditional on four measurable dataset and deployment properties, each grounded in a specific empirical result rather than an unsupported general claim. Table 11 serves as a lookup procedure and Figure 10 as a visual aid, without requiring re-derivation of the underlying benchmark data.

An Informal Held-Out Check on the Decision Framework

One point to note is that Figure 10 and Table 11 were based on and evaluated with the same primary dataset, a post hoc synthesis, rather than a framework tested with independent data. The supplementary data set (Table 10) was collected separately (a different session, a smaller value range, k = 100, and before the framework was built), and is therefore not used to create any rule. The check is mixed rather than uniformly confirmatory. Below the n = 700–800 crossover noted above, Tim Sort is actually faster than Counting Sort in the supplementary dataset (e.g., 0.015 ms vs. 0.055 ms at n = 100), which Rule R1's recommendation of Counting Sort does not anticipate; above that crossover, Counting Sort is faster, consistent with Rule R1. Because Rule R1's stated condition is bounded, known-range integer keys with $k \lesssim 10n$ rather than a claim of superiority at every size, the sub-crossover result does not contradict the rule's formal condition, but it does show that "recommended algorithm" should not be read as "fastest at all sizes within that regime" -- a distinction this study did not make

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

clearly enough in earlier framing and corrects here. This is one rule checked against one independent dataset, not the framework in general: Rules R2-R7 are not tested – they do not involve the memory profiling data or the large key ranges that they apply to, so there is no data for them. A proper validation would require a dedicated held-out benchmark that covers all the conditions in the seven rules, but this is left for future work.

All twelve algorithms are also possible to run on n = 1000 to n = 10,000, although these are only common for an introductory set of benchmarking exercises, nine can be executed comfortably for these values because of the complexity alone and Counting, Radix and Bucket Sort because it depends on the value range of k (or, in the case of Radix Sort, on d, the number of digits), not necessarily on n. The complexity basis for each of the algorithms is shown in Appendix B (Table B1) and is restated in this context, but is not reported here as a new empirical result.

Limitations

This study's external validity is bounded in four specific, related ways, addressed here explicitly rather than folded into a single general disclaimer.

Python-specific implementation. All twelve algorithms were implemented as pure-Python functions rather than calling into optimized C extensions (beyond Tim Sort's native sorted(), which is itself a C implementation). The Quick Sort anomaly discussed in the Discussion, RQ2, is a direct symptom of this choice: an implementation that relied more heavily on Python's C-level primitives, or that was written in a compiled language, could plausibly reverse that specific finding. Results reported here as algorithm-level properties should therefore be read as CPython-implementation-level properties unless independently corroborated — Goel et al. [5] similarly found language choice to materially affect relative sorting-algorithm rankings across C, C++, Java, and Python.

Interpreter overhead. CPython's per-operation bytecode dispatch overhead is not uniform across algorithms: it penalizes algorithms with many small Python-level operations (e.g., Quick Sort's per-partition comparisons) more than those that delegate bulk work to C-implemented primitives (e.g., Merge Sort's list slicing, Tim Sort's native sorted()). Absolute and even relative timings measured here are therefore an interpreter-overhead-inclusive quantity, not a clean measurement of algorithmic work alone.

Hardware dependence. All benchmarks were collected on a single machine and operating system configuration; no cross-hardware replication was performed. Memory-hierarchy effects (cache size, cache-line behavior) are known to materially affect sorting-algorithm performance rankings, particularly at the boundary between elementary and general-purpose algorithms; recent work benchmarking sorting algorithms on resource-constrained 8-bit microcontrollers [6] illustrates just how far absolute and even relative performance can diverge across hardware classes. This study's timings should not be assumed to transfer to substantially different hardware without re-benchmarking.

Generalization to other programming languages. This study's findings are specific to CPython's interpreter and memory model and may not generalize to compiled languages (C, C++) or JIT-compiled runtimes (PyPy, Java's JIT). This is consistent with Goel et al.'s [5] finding that relative sorting-algorithm performance is not language-invariant. The rule-based decision framework of the Discussion (Table 11) is therefore best treated as validated for Python deployments specifically; Rule R7, in particular, which cautions against naive Quick Sort in interpreted environments, may not apply once ported to a compiled language.

Beyond external validity, two scope limitations apply to the primary dataset itself: Bubble, Selection, and Insertion Sort were not tested beyond n = 1,000 due to computational impracticality, consistent with the scope limitation of prior comparative studies [1]; and Bucket Sort's descending-distribution trial at n = 100,000 was not completed due to the same bucket-collapse degeneracy documented above, which would have required substantially longer runtime — this is reported transparently rather than omitted silently.

There are multiple directions for the next step in further advancing the rigor and generalizability of this body of research. Two of the most consequential comparisons are now accompanied by significance testing (Table 7) and add to what were previously point-estimate and example-driven claims, rather than being deferred (see

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

Results). Four things remain open. First, the significance testing reported in Table 7 is limited to the two most important comparisons for RQ1 and RQ2; the same statistical basis would apply to the full results table if we were to extend the significance testing to all twelve algorithm/size combinations (or to a non-parametric alternative, such as the Kruskal-Wallis test). Second, the benchmark was applied to other data sets that presumably have different percentages of duplicates and different forms of sorting for their inputs, thus expanding the generalizability of the distribution-sensitivity results found here beyond the three distributions tested. Third, having the same code implemented in different runtimes (e.g., PyPy) or compiled languages would help separate algorithmic effects from CPython-specific implementation effects, directly testing this study's proposed language-dependent explanation for the Quick Sort anomaly (see Discussion, RQ2). The fourth and related need is a proper validation benchmark, not part of the construction of the decision framework, but collected separately from and after the construction of the decision framework, over the entire range of conditions addressed by Rules R2-R7; a single rule check performed here (Table 10 against Rule R only) is not a general check of Figure 10 or Table 11. These directions are left for future work.

CONCLUSION

This study extended Wibowo and Faisal's [12] five-algorithm Timsort benchmark to twelve algorithms across comparison-based, non-comparison-based, and hybrid/adaptive paradigms, adding memory profiling, empirically verified stability, an operationalized adaptivity ratio, formal significance testing, a key-range scaling analysis, and a systematic rank-correlation analysis to the original time-only methodology. The results empirically confirm the theoretical comparison-sort lower bound ($p < 0.001$) and locate its precise boundary: Counting Sort's advantage over Heap Sort holds while the value range k remains at or below roughly 10 times n and inverts once k approaches 100 times n. The results also demonstrate that Quick Sort's practical speed disadvantage in this study is implementation- and language-dependent rather than universal, and that distribution sensitivity across all twelve algorithms is concentrated in adaptive algorithms' response to best-case ordering rather than in a uniform reshuffling under any non-random input. They also empirically illustrate a caveat noted but rarely demonstrated in the sorting literature — Bucket Sort's sensitivity to its value-range parameter — rather than reporting it as a previously undocumented phenomenon.

This study makes five primary contributions:

(1) it extends prior five-algorithm benchmarking [12] to a comprehensive twelve-algorithm comparison spanning comparison-based, non-comparison-based, and hybrid paradigms; (2) it introduces a multi-dimensional benchmarking methodology combining runtime, memory usage, empirically verified stability, and adaptivity, rather than runtime alone; (3) it demonstrates experimentally, with statistical significance testing and a systematic rank-correlation analysis rather than point estimates and anecdotal examples alone, how input distribution, implementation language, and bounded key ranges influence algorithm performance beyond asymptotic complexity, including a precisely located crossover point for Counting Sort's key-range advantage and Bucket Sort's value-range sensitivity; (4) it formalizes these findings into an explicit, rule-based algorithm-selection framework (Figure 10, Table 11), rather than resting on narrative recommendations alone; and (5) it subjects that framework to an initial, honestly-scoped held-out check against an independently collected dataset, confirming one of its seven rules while explicitly identifying the remainder as not yet validated — a distinction between post-hoc synthesis and predictive validation that this study treats as worth stating plainly rather than glossing over.

This study's scope is deliberately bounded to single-machine, single-threaded, in-memory Python sorting, and that boundary marks the clearest direction for future work. GPU-accelerated sorting (e.g., bitonic and radix variants suited to SIMD/GPU architectures), distributed sorting across multi-node clusters, external-memory sorting for datasets exceeding available RAM, and parallel or multi-threaded implementations of the twelve algorithms benchmarked here would each subject this study's four measurement dimensions — time, memory, stability, and adaptivity — to fundamentally different hardware and concurrency models than the ones tested. A further direction, foreshadowed by this study's own rule-based framework and by recent machine-learning-guided approaches to algorithm selection [3], is a learned selection policy that could complement or eventually

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

supersede Table 11's static rules once trained on a benchmark corpus broader than the single-machine, single-language dataset used here.

These contributions are experimental and integrative rather than theoretical: the study introduces no new sorting algorithm, complexity proof, or data structure. Framed accordingly, it provides reproducible empirical evidence and a practical, rule-based decision-support tool that extends the original study's single time-based ranking into a scoped, multi-criteria recommendation model — a contribution to empirical algorithmics, performance engineering, and computer science education, where it demonstrates concretely that asymptotic complexity alone does not determine real-world performance.

REFERENCES

1. Alkharabsheh, K., Alturani, I., Alturani, A., & Zanoon, N. (2013). Review on sorting algorithms: A comparative study. International Journal of Computer Science and Security, 7(3), 120–126.
2. Auger, N., Jugé, V., Nicaud, C., & Pivoteau, C. (2018). On the worst-case complexity of TimSort. In 26th Annual European Symposium on Algorithms (ESA 2018). https://doi.org/10.48550/arXiv.1805.08612
3. Balasubramanian, S. A. (2025). Adaptive hybrid sort: Dynamic strategy selection for optimal sorting across diverse data distributions. arXiv. https://doi.org/10.48550/arXiv.2506.20677
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.
5. Goel, K., Dwivedi, P., & Sharma, O. (2023). Performance analysis of various sorting algorithms: Comparison and optimization. 2023 11th International Conference on Intelligent Systems and Embedded Design (ISED), 1–5. https://doi.org/10.1109/ISED59382.2023.10444609
6. Golonka, J., & Krużel, F. (2026). Empirical evaluation of unoptimized sorting algorithms on 8-bit AVR Arduino microcontrollers. Sensors, 26(1), 214. https://doi.org/10.3390/s26010214
7. Oladipupo, E. T., Abikoye, O. C., Akande, N. O., Kayode, A. A., & Adeniyi, J. K. (2020). Comparative study of two divide and conquer sorting algorithms: Quicksort and mergesort. Procedia Computer Science, 171, 2532–2540. https://doi.org/10.1016/j.procs.2020.04.274
8. Peters, T. (2002). listsort.txt. CPython git repository. https://svn.python.org/projects/python/trunk/Objects/listsort.txt
9. Romano, J., Kromrey, J. D., Coraggio, J., & Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? Annual Meeting of the Florida Association of Institutional Research.
10. Sarwar, S. M., Jaragh, M. H. A., & Wind, M. (1994). An empirical study of the run-time behavior of quicksort, Shellsort and mergesort for medium to large size data. Computer Languages, 20(2), 127–134. https://doi.org/10.1016/0096-0551(94)90019-1
11. Sundaramoorthy, S., & Karunanidhi, G. (2025). A systematic analysis on performance and computational complexity of sorting algorithms. Discover Computing. https://doi.org/10.1007/s10791-025-09724-w
12. Wibowo, F. R., & Faisal, M. (2024). Comparative analysis of sorting algorithms: TimSort Python and classical sorting methods. JISA (Jurnal Informatika dan Sains), 7(1), 11–18.

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

APPENDIX

This appendix contains supplementary material referenced from the main text: graphical presentation of the independent small-scale verification dataset (Appendix A; cf. Table 10), and the complexity basis underlying the practical-scalability claims of the Results section (Appendix B).

Appendix A: Supplementary Small-Scale Verification Figures

This appendix presents the supplementary small-scale dataset (Table 10) in the same four-group graphical format as Figures 1 through 4, allowing direct visual comparison of growth patterns at this smaller scale.

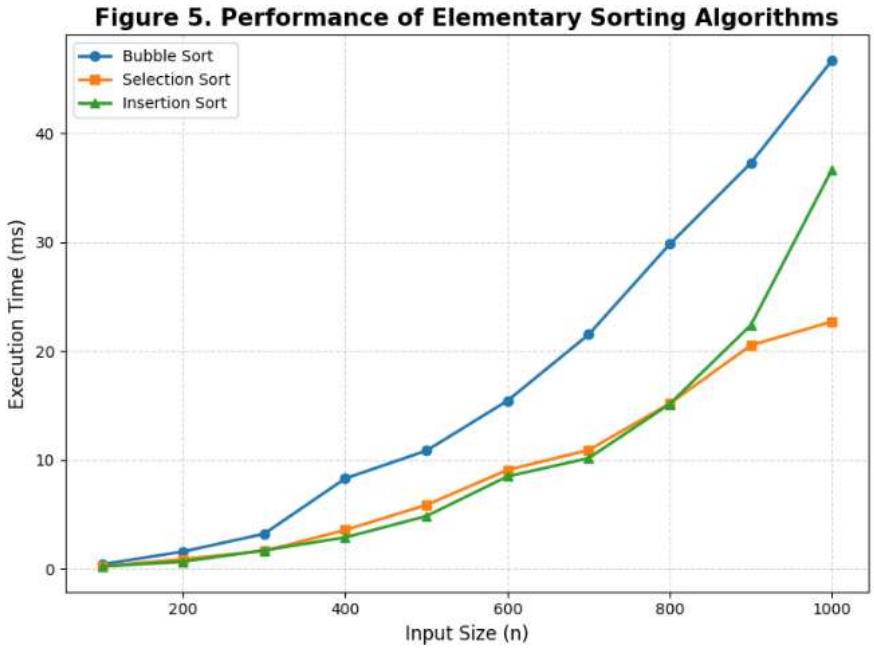


Figure A1. Performance Comparison of Elementary Sorting Algorithms (Supplementary Dataset, n = 100–1,000)

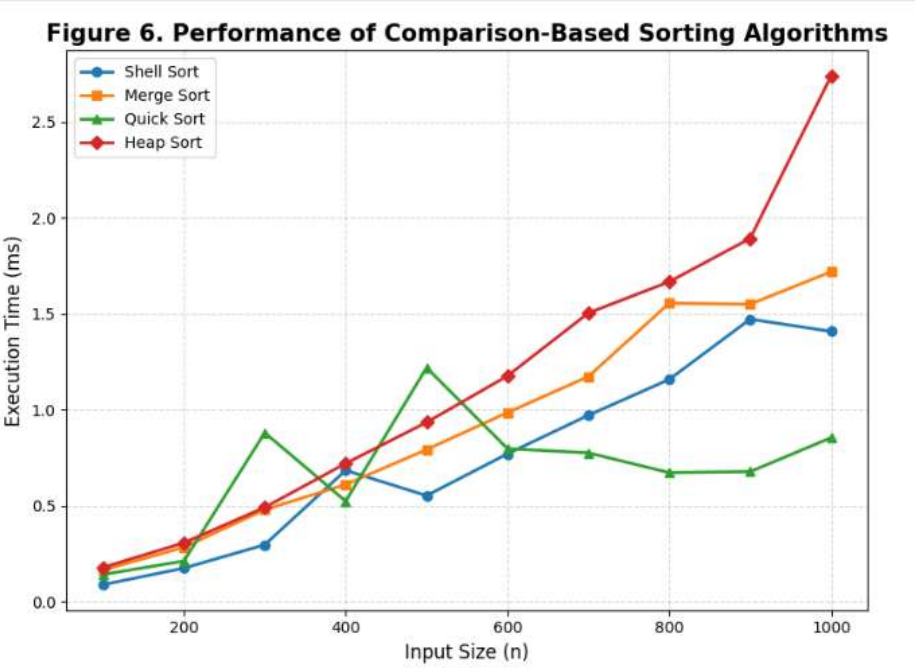


Figure A2. Performance Comparison of Advanced Comparison-Based Sorting Algorithms (Supplementary Dataset, n = 100–1,000)

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

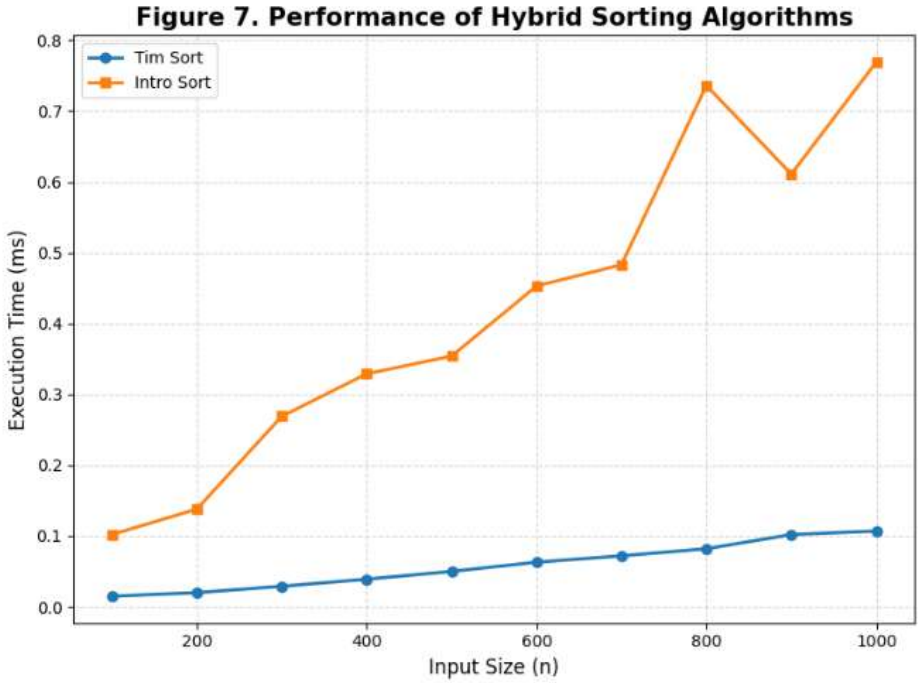


Figure A3. Performance Comparison of Hybrid Sorting Algorithms (Supplementary Dataset, n = 100–1,000*)*

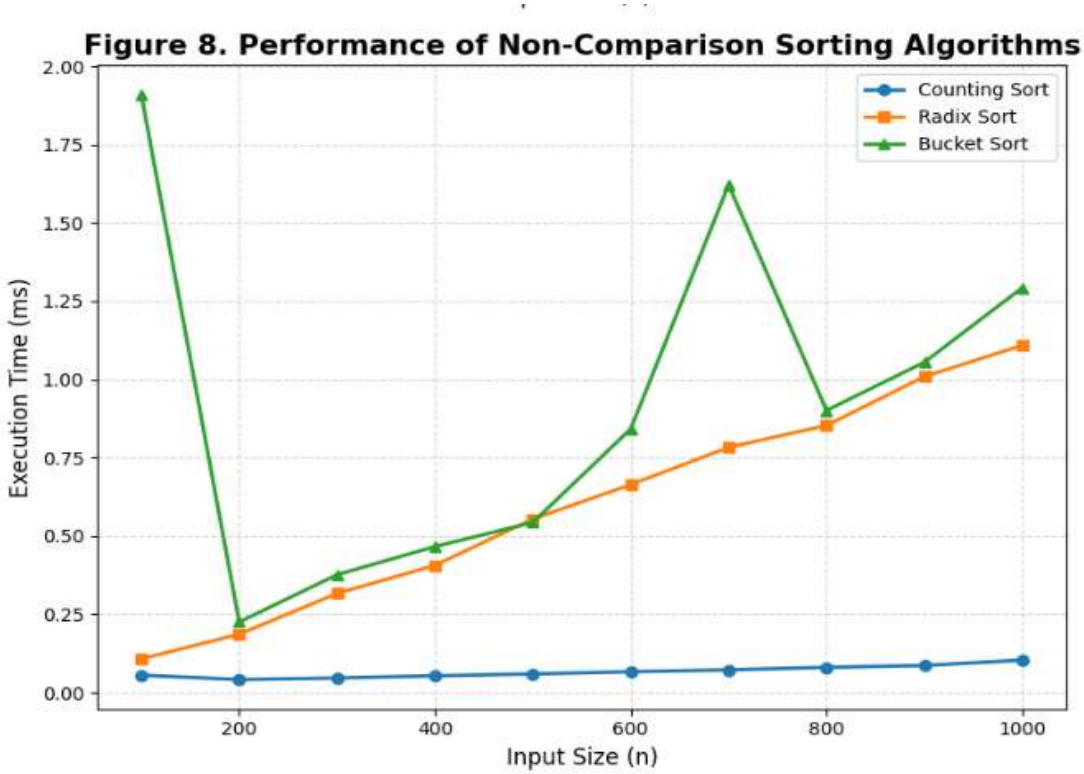


Figure A4. Performance Comparison of Non-Comparison Sorting Algorithms (Supplementary Dataset, n = 100–1,000)

Appendix B: Practical Scalability Across n = 1,000–10,000

A related practical question is which of the twelve algorithms can be executed efficiently across the full range of n = 1,000 to n = 10,000 — a range commonly used in undergraduate and introductory benchmarking exercises. Of the twelve algorithms tested in this study, nine execute comfortably at every size within this range, while the

ISSN 2278-2540 | DOI: 10.51583/IJLTEMAS | Volume XV, Issue VII, July 2026

remaining three remain technically executable but become practically impractical as n approaches the upper end of the range.

Table B1. Algorithms that scale comfortably across n = 1,000–10,000 and the complexity basis for their feasibility

Algorithm	Basis for feasibility at n = 1,000–10,000
Shell Sort	O(n^1.5) or better — growth remains modest across this range
Merge Sort	Guaranteed O(n log n) in all cases
Quick Sort	O(n log n) average case
Heap Sort	Guaranteed O(n log n) in all cases
Tim Sort	O(n log n), with adaptive best-case behavior
Intro Sort	Guaranteed O(n log n) worst case
Counting Sort	O(n + k) — governed by value range k, not array length n
Radix Sort	O(n × d) — governed by digit count d, not array length n
Bucket Sort	O(n + k), provided the assumed value range matches the actual data distribution

Consistent with Table 1, Shell, Merge, Quick, Heap, Tim, and Intro Sort completed at n = 10,000 in roughly 1.7 to 722 milliseconds, and Counting Sort completed in approximately 13 milliseconds — none of these algorithms exhibit runtime growth that becomes prohibitive within this range.

By contrast, Bubble, Selection, and Insertion Sort are all $O(n^2)$, meaning each tenfold increase in array size costs approximately a hundredfold increase in execution time rather than a tenfold increase. Using this study's measured value of 379.81 ms for Bubble Sort at n = 1,000 (random distribution), a quadratic-scaling projection places its expected execution time at approximately 38 seconds at n = 10,000 for a single run. This is not a failure to execute — the algorithm remains correct and will complete — but it becomes impractical once multiplied across repeated trials, multiple array sizes, and multiple input distributions within a single benchmarking loop, which is why this study capped these three algorithms at n = 1,000 (see Methodology).

A further subtlety applies to Counting, Radix, and Bucket Sort: their feasibility at a given size is governed primarily by the value range k (and, for Radix Sort, the digit count d) rather than by the array length n itself. Within the bounded value range used in this study (1–1,000), all three scale well across n = 1,000 to 10,000. However, if the value range were substantially widened — for example, to 1–10,000,000 — Counting Sort's internal count array would grow to match that range regardless of how many elements are actually being sorted, and Bucket Sort would be subject to the same bucket-collapse degeneracy documented earlier in the Results, in which a value-range assumption that diverges from the data's actual distribution drives its performance toward $O(n^2)$. Feasibility for these three algorithms is therefore better assessed against the value range k than against n alone.