QuickSort Variants: Analysis and Benchmark Report

Course Project

November 30, 2025

Abstract

We study three instrumented QuickSort implementations (vanilla, smaller-subtree, and cutoff-16), compare them against Merge Sort and the C++ standard library's `std::sort`, and relate empirical results to classical time and space bounds. Benchmarks cover synthetic and real-world (NIFTY 1-minute) integer datasets, recording wall-clock time and swap counts. All code lives in a single C++17 benchmark driver; plots are produced by accompanying Python scripts.

1 Introduction

QuickSort is a comparison sort that partitions an array around a pivot, recursively sorting the resulting subarrays. It is prized for its in-place memory footprint, cache-friendly sequential scans, and strong expected-time guarantees under randomized pivot selection. Practical libraries often combine QuickSort with other strategies (e.g., insertion sort on small segments, heap sort fallbacks), motivating a careful look at variants and their constant factors.

2 Algorithms and Complexity

2.1 QuickSort procedure (conceptual)

On an input array $A[lo..hi]$:

1. Choose a pivot index uniformly at random from $[lo, hi]$; let p be its value.
2. Partition $A[lo..hi]$ in one linear pass, maintaining a store index s. Scan i from lo to $hi - 1$; whenever $A[i] < p$, swap $A[i]$ with $A[s]$ and increment s. Finally swap $A[s]$ with the pivot. The pivot ends at index s; elements left of s are $< p$; elements right of s are $\geq p$.
3. Recursively sort the left subarray $[lo, s-1]$ and the right subarray $[s+1, hi]$ in any fixed order (or via tail recursion elimination where noted).

Only comparisons and swaps inside the partition pass plus the recursive calls contribute to running time; no auxiliary arrays are used.

2.2 Vanilla QuickSort

The pivot is sampled uniformly from the current subarray; after partitioning, both sides are recursed on.

Expected time (full derivation). Let $T(n)$ be the expected number of primitive operations (or comparisons) on an array of size n. Let $c > 0$ be the partition-pass constant. Conditioning on the pivot rank k (with k elements on the left, $n-1-k$ on the right) and applying the law of total expectation:

$$\begin{aligned}
T(n) &= \frac{1}{n}\sum_{k=0}^{n-1}\Big(T(k) + T(n-1-k) + cn\Big) \\
&= \frac{1}{n}\sum_{k=0}^{n-1} T(k) + \frac{1}{n}\sum_{k=0}^{n-1} T(n-1-k) + \frac{1}{n}\sum_{k=0}^{n-1} cn \\
&= \frac{1}{n}\sum_{k=0}^{n-1} T(k) + \frac{1}{n}\sum_{j=0}^{n-1} T(j) + cn \quad \text{(rename } j = n-1-k\text{)} \\
&= \frac{2}{n}\sum_{j=0}^{n-1} T(j) + cn.
\end{aligned}$$

Define the prefix sum $S(n) = \sum_{j=0}^{n} T(j)$. Then for $n \geq 1$:

$$\begin{aligned}
n\,T(n) &= 2\,S(n-1) + cn^2, \\
S(n) &= S(n-1) + T(n).
\end{aligned}$$

Multiply the second equation by n and substitute $T(n) = S(n) - S(n-1)$ into the first:

$$\begin{aligned}
n\big(S(n) - S(n-1)\big) &= 2\,S(n-1) + cn^2, \\
nS(n) - nS(n-1) &= 2S(n-1) + cn^2, \\
nS(n) &= (n+2)S(n-1) + cn^2, \\
\frac{S(n)}{n+1} &= \frac{S(n-1)}{n} + \frac{cn}{n+1}.
\end{aligned}$$

Unrolling this telescoping form from n down to 1 and noting $S(0) = T(0) = \Theta(1)$:

$$\begin{aligned}
\frac{S(n)}{n+1} &= \frac{S(0)}{1} + c\sum_{i=1}^{n}\frac{i}{i+1} \\
&= \Theta(1) + c\sum_{i=1}^{n}\left(1 - \frac{1}{i+1}\right) \\
&= \Theta(1) + c\left(n - \sum_{i=1}^{n}\frac{1}{i+1}\right) \\
&= \Theta(1) + c\left(n - (H_{n+1} - 1)\right),
\end{aligned}$$

where $H_m = \sum_{j=1}^{m} 1/j$ is the m-th harmonic number. Multiplying by $n+1$ and using $T(n) = S(n) - S(n-1)$, we obtain

$$T(n) \leq 2(n+1)H_n - 4n + O(n),$$

so $T(n) = \Theta(n \log n)$ in expectation because $H_n = \Theta(\log n)$.

Worst and best cases. Worst-case pivot ranks yield $T(n) = T(n-1) + cn = \Theta(n^2)$. Perfectly balanced splits give $T(n) = 2T(n/2) + cn = \Theta(n \log n)$. Stack depth is $O(n)$ in the worst case.

2.3 QuickSort with smaller-subtree recursion

After partitioning, the algorithm recurses on the smaller side and iterates on the larger. The running-time recurrence matches vanilla QuickSort, so the same bounds hold: $\Theta(n \log n)$ expected, $\Theta(n^2)$ worst case. Stack depth satisfies

$$D(n) \le 1 + D\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right),$$

giving $D(n) = O(\log n)$ for all inputs.

Stack depth. We prove $D(n) \le \lfloor \log_2 n \rfloor + 1$ by induction on $n \ge 1$. Base: $n = 1$ gives $D(1) = 1$. Inductive step: assume the claim holds for all sizes $< n$. The smaller side has size at most $\lfloor (n-1)/2 \rfloor$, so

$$D(n) \le 1 + D\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) \le 1 + \left\lfloor \log_2 \left\lfloor \frac{n-1}{2} \right\rfloor \right\rfloor + 1 \le \lfloor \log_2 n \rfloor + 1.$$

Thus depth grows logarithmically regardless of pivot outcomes.

Expected time. The pivot rank is still uniform, so the conditional recurrence is identical to the vanilla case:

$$T_{\text{small}}(n) = \frac{1}{n} \sum_{k=0}^{n-1} \big(T_{\text{small}}(k) + T_{\text{small}}(n-1-k)\big) + cn.$$

Define $S_{\text{small}}(n) = \sum_{j=0}^{n} T_{\text{small}}(j)$. Then for $n \ge 1$:

$$\begin{aligned} n\, T_{\text{small}}(n) &= 2\, S_{\text{small}}(n-1) + cn^2, \\ S_{\text{small}}(n) &= S_{\text{small}}(n-1) + T_{\text{small}}(n). \end{aligned}$$

Substitute $T_{\text{small}}(n) = S_{\text{small}}(n) - S_{\text{small}}(n-1)$:

$$\begin{aligned} n\big(S_{\text{small}}(n) - S_{\text{small}}(n-1)\big) &= 2S_{\text{small}}(n-1) + cn^2, \\ nS_{\text{small}}(n) &= (n+2)S_{\text{small}}(n-1) + cn^2, \\ \frac{S_{\text{small}}(n)}{n+1} &= \frac{S_{\text{small}}(n-1)}{n} + \frac{cn}{n+1}. \end{aligned}$$

Unrolling from n to 1 gives

$$\frac{S_{\text{small}}(n)}{n+1} = \Theta(1) + c \sum_{i=1}^{n} \frac{i}{i+1} = \Theta(1) + c\big(n - (H_{n+1} - 1)\big),$$

so $T_{\text{small}}(n) = S_{\text{small}}(n) - S_{\text{small}}(n-1) = \Theta(n \log n)$, matching the vanilla expectation.

2.4 QuickSort with cutoff 16

When a subarray has size ≤ 16, the implementation immediately applies insertion sort and stops recursing on that branch. Partitioning proceeds as in A1. Costs consist of:

- Partitioning down to size 16: still $\Theta(n \log n)$.
- Insertion sort on many tiny segments: each element participates in at most one small-segment insertion pass, contributing $O(n)$.

Thus the overall bound remains $\Theta(n \log n)$ expected, $\Theta(n^2)$ worst case, but constant factors drop because insertion sort is efficient on very small arrays. Stack depth follows A1 when the tail-recursive strategy is used.

Expected time. Let $T_c(n)$ be the expected time with cutoff $c = 16$. For $n > c$,

$$T_c(n) = \frac{1}{n} \sum_{k=0}^{n-1} \big(T_c(k) + T_c(n-1-k)\big) + cn,$$

identical to the vanilla QuickSort recurrence for the partitioning portion. For $n \le c$, $T_c(n) = O(c^2)$ due to insertion sort. By induction on n,

$$T_c(n) \le T_{\text{vanilla}}(n) + O(n),$$

where $T_{\text{vanilla}}(n)$ is the solution to the vanilla recurrence. Inductive step: if $n > c$,

$$T_c(n) \le \frac{2}{n} \sum_{j=0}^{n-1} \big(T_{\text{vanilla}}(j) + O(j)\big) + cn = T_{\text{vanilla}}(n) + O(n),$$

using the same prefix-sum manipulation as before and the inductive hypothesis on subproblem sizes $< n$. If $n \le c$, $T_c(n) = O(c^2) = O(n)$. Therefore $T_c(n) = \Theta(n \log n)$ in expectation.

Stack depth. Because the algorithm recurses only on the smaller side and iterates on the larger, the depth recurrence matches the smaller-subtree case:

$$D_c(n) \le 1 + D_c\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right),$$

with base $D_c(n) = 1$ for $n \le c$. The same induction as before yields $D_c(n) \le \lfloor \log_2 n \rfloor + 1$. Worst-case pivot choices still yield $T_c(n) = \Theta(n^2)$ time, but stack depth remains $O(\log n)$ because the larger side is never recursed.

2.5 Merge Sort

The top-down implementation recurses on halves and merges with a temporary buffer of size n. The recurrence $T(n) = 2T(n/2) + cn$ yields $\Theta(n \log n)$ in all cases; extra space is $\Theta(n)$ for the buffer.

2.6 `std::sort`

Library `std::sort` is an introspective hybrid (QuickSort + heap sort fallback + insertion sort on small partitions), guaranteeing $O(n \log n)$ worst-case time while remaining in-place. It serves as a realistic production baseline.

2.7 Summary of Bounds

Algorithm	Best	Expected	Worst (time / stack)
QuickSort (vanilla)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$ time, $O(n)$ stack
QuickSort (smaller-subtree)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$ time, $O(\log n)$ stack
QuickSort (cutoff 16)	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$ time, $O(\log n)$ stack
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$, $O(n)$ buffer
`std::sort`	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$ (introsort)

Table 1: Asymptotic behavior of evaluated algorithms.

3 Experimental Workflow

The entire benchmark lives in src/main.cpp; a compiled binary quicksort_bench is driven by scripts/run_benchmarks.py.

3.1 Data Generation

- **Synthetic distributions:** deterministic given (n, seed). Sorted uses $a_i = i$; almost-sorted applies $k = \lfloor 0.05n \rfloor$ random swaps; uniform draws from $[-10^9, 10^9]$; normal draws from $\mathcal{N}(0, 1000^2)$ with clamping to $[-10^9, 10^9]$.
- **Stock data:** preprocessed Kaggle NIFTY 1-minute closes (10^6 integers) stored in data/nifty_1m_int_1M.txt. The benchmark reads the first n values for the stock distribution.
- **Randomness:** all generators and pivot choices use std::mt19937_64 seeded with seed_base + rep, where seed_base is deterministic per configuration (see compute_seed_base in the Python script).

3.2 Benchmark Driver

- **Invocation:** run ./quicksort_bench with arguments $\langle$algo_id$\rangle$, $\langle$dist_id$\rangle$, n, seed_base, reps, stock_path.
- **Metrics:** each repetition prints <time_ns> <swaps>; swap count is -1 for std::sort.
- **Sizes and reps:** run_benchmarks.py sweeps $n = 2^{10}, \ldots, 2^{20}$ (1024 to 1,048,576), plus the full stock file size 1,007,339; repetitions $R = 5$.
- **Outputs:** rows are aggregated into results/raw_results.csv.
  Plots produced by plot_results.py reside in results/plots/ and are embedded below.

3.3 Implementation Notes

All implementations operate on std::vector<int> and use a Mersenne Twister (std::mt19937_64) seeded deterministically per run. The global counter g_swap_count tracks swaps for instrumented algorithms (QuickSort variants and Merge Sort); std::sort leaves it at -1.

4 Results

Figures 1–5 show mean wall-clock time (ms) versus n on log-scaled x and swap counts for QuickSort/Merge Sort variants.

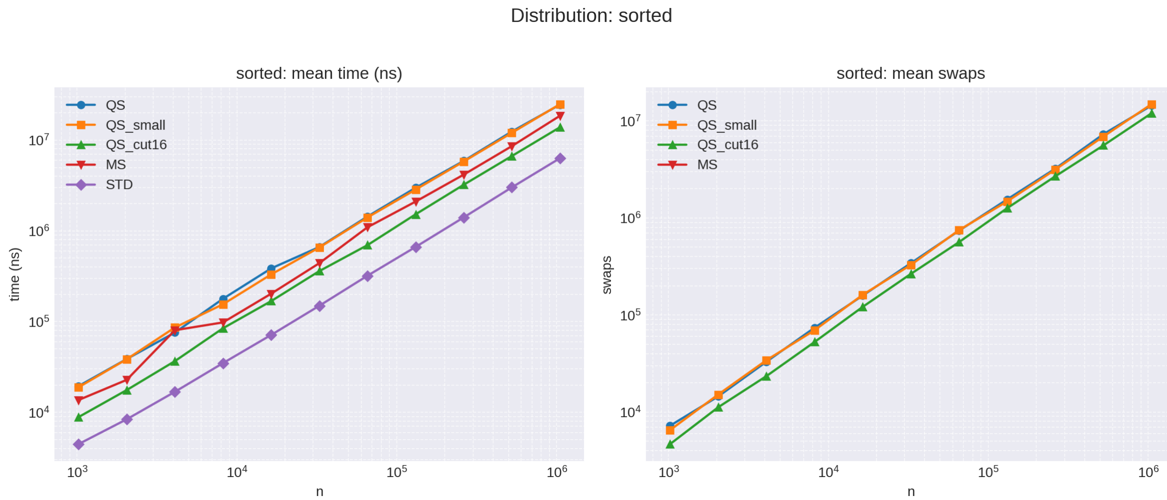

Figure 1: Sorted input.

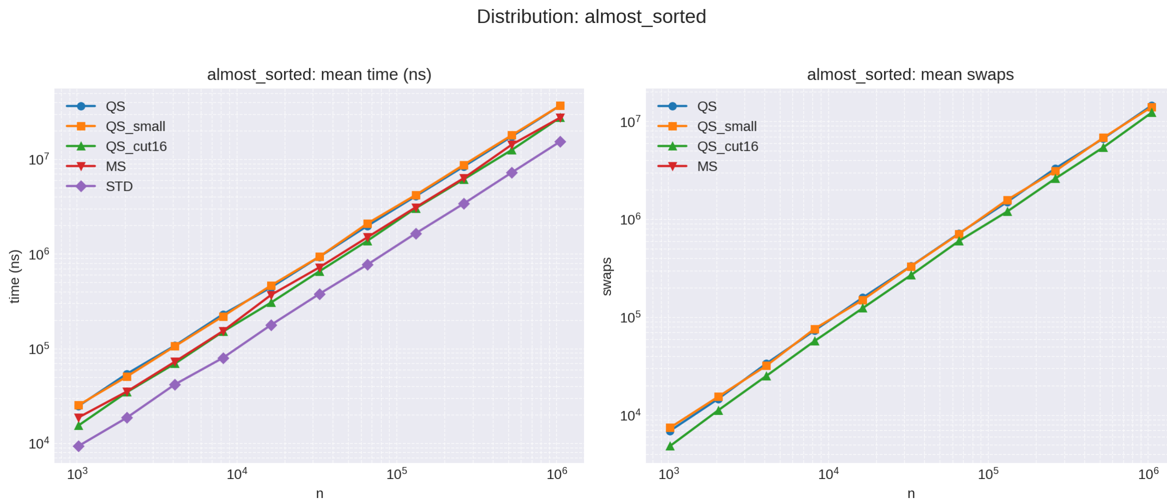

Figure 2: Almost-sorted input (5% swaps).

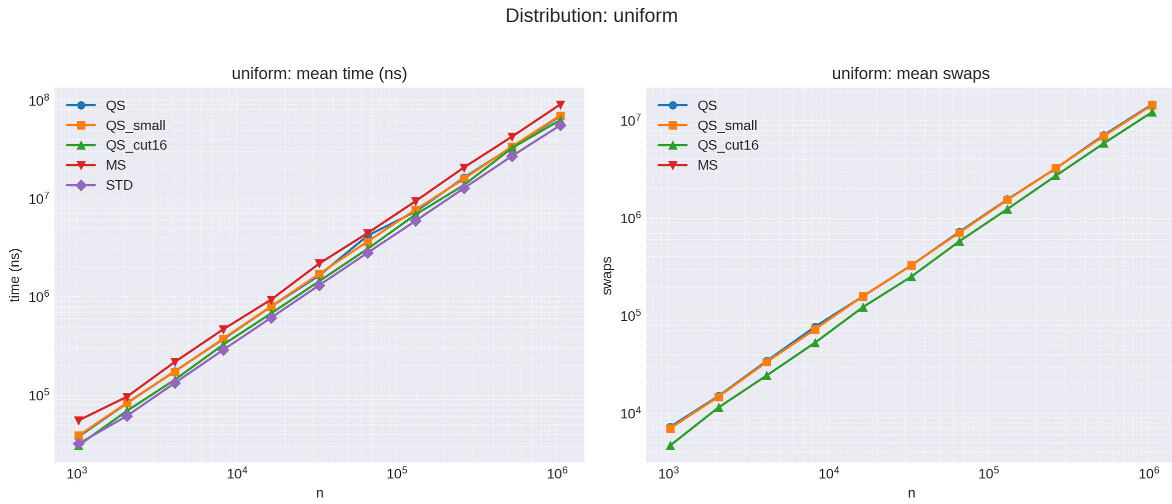

Figure 3: Uniform random input.

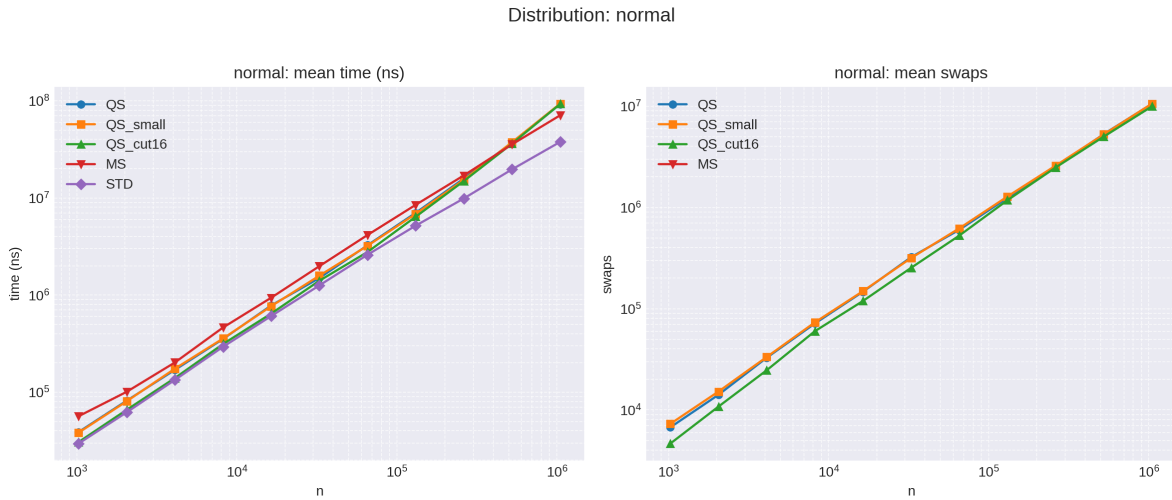

Figure 4: Normal distribution input.

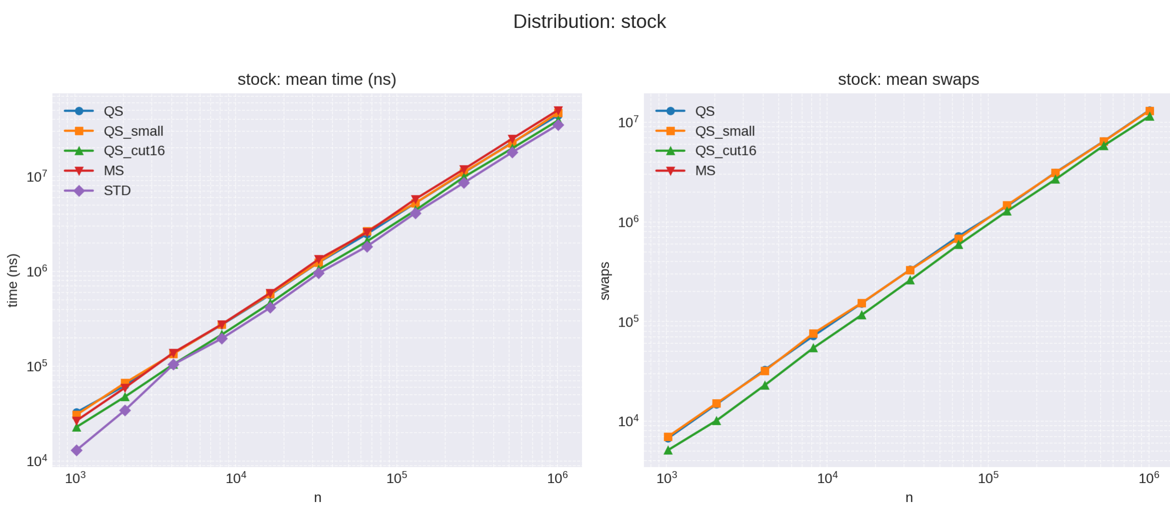

Figure 5: Real stock data input.

4.1 Observations on Figures

- **QS_cut16 leads among custom variants:** Across all distributions, QS_cut16 is the fastest of the instrumented QuickSorts. On uniform input with $n = 2^{20}$, QS_cut16 averages ≈ 62.9 ms versus ≈ 68.8 ms for QS and ≈ 70.1 ms for QS_small; swap counts drop from ≈ 14.6M (QS) to ≈ 12.2M (QS_cut16).
- **Effect of tail recursion control:** QS_small and QS have nearly identical times (e.g., sorted $n = 2^{20}$: ≈ 24.9 ms vs ≈ 24.7 ms) while QS_small caps stack depth at $O(\log n)$.
- **QS vs Merge Sort:** Merge Sort trails the QuickSort variants on noisy data (70–90 ms at $n = 2^{20}$ for uniform/normal) and requires $\Theta(n)$ extra space; on structured data (sorted/almost-sorted) it closes the gap but still lags QS_cut16.
- **Real stock mirrors synthetic:** On the full NIFTY slice ($n = 1{,}007{,}339$), QS_cut16 is the quickest custom option at ≈ 38.9 ms, ahead of QS (≈ 44.1 ms) and QS_small (≈ 46.9 ms); patterns match the synthetic distributions.
- **Why QS_cut16 excels:** Insertion sort on tiny partitions eliminates extra partition passes and reduces swaps; tail recursion keeps stack shallow. These micro-optimizations shrink constant factors without altering the $\Theta(n \log n)$ expectation, yielding measurable speedups on both ordered and random inputs.

5 Why QuickSort Works Well in Practice

- **In-place**: Partitioning needs only $O(1)$ auxiliary memory, which preserves cache locality and avoids large allocations compared to Merge Sort's $O(n)$ buffer.
- **Cache-friendly scans**: Sequential reads/writes during partitioning yield few cache misses, visible in the tight constants of A2 and `std::sort`.
- **Randomization**: Uniform pivot sampling delivers $\Theta(n \log n)$ expected time on non-adversarial inputs—a good match for market and logging data that lack worst-case structure.
- **Hybridization payoff**: Adding insertion sort on tiny ranges (A2) trims swap counts and improves time without changing asymptotic bounds; introsort in `std::sort` pushes this idea further with heap-sort fallbacks.
- **Empirical alignment**: The real-stock results match the synthetic curves, reinforcing that QuickSort-style strategies handle both clustered (sorted/almost-sorted) and noisy (uniform/normal) data effectively.

6 Real-world Use Cases

- **C++ standard library:** `std::sort` is an introspective QuickSort hybrid used pervasively in systems code (cppreference).
- **Java primitive arrays:** `Arrays.sort` for primitives uses a dual-pivot QuickSort (Oracle JDK docs).
- **Database engines:** PostgreSQL uses a QuickSort-based tuplesort for many in-memory sorts before spilling to disk (source code).
- **Systems utilities:** The classic C `qsort` in libc remains a QuickSort-style routine, making the algorithm a default choice for general-purpose array sorting.

7 Conclusion

QuickSort's combination of in-place operation, cache-friendly access, and strong expected bounds makes it a natural first choice for large in-memory datasets. Our experiments confirm that modest engineering—tail recursion elimination and tiny-segment insertion sort—improves constant factors without altering complexity. Production-grade hybrids such as `std::sort` remain the performance ceiling, but tuned QuickSort variants deliver predictable, efficient behavior on both synthetic and real financial data.