**Sorting Algorithm Performance Analysis**

**Abstract**

This research paper presents a comprehensive experimental analysis of four fundamental sorting algorithms: Quick Sort, Merge Sort, Radix Sort, and Insertion Sort. Through systematic benchmarking across three different CPU architectures (Apple M4, AMD Threadripper 3960X, and Ryzen 9900 X3D), we evaluate algorithm performance under varying data conditions including different input sizes (1,000 vs 100,000 elements), data arrangements (sorted, random, reverse), and data types (integers vs strings). Our findings provide empirical validation of theoretical complexity bounds and reveal practical performance characteristics crucial for algorithm selection in real-world applications.

**Introduction**

Sorting algorithms form the backbone of computer science, with applications spanning database management, search optimization, and data processing pipelines. While theoretical complexity analysis provides fundamental insights into algorithmic efficiency, practical performance often deviates from theoretical predictions due to factors such as memory hierarchy, CPU architecture, and implementation details.[[1]](#fn1)[[2]](#fn2)

This study examines four representative sorting algorithms from different algorithmic paradigms: divide-and-conquer (Quick Sort, Merge Sort), non-comparison based (Radix Sort), and simple comparison-based (Insertion Sort). Each algorithm represents distinct trade-offs between time complexity, space complexity, and practical performance characteristics.

**Theoretical Analysis and Expected Results**

**Algorithm Complexity Bounds**

**Quick Sort:**

* **Best Case:** - occurs when pivot consistently divides the array into balanced partitions
* **Average Case:** - expected performance with random pivot selection
* **Worst Case:** - occurs when pivot is consistently the smallest or largest element (e.g., sorted arrays with poor pivot selection)

**Merge Sort:**

* **Best Case:** - consistent performance regardless of input distribution
* **Average Case:** - maintains logarithmic depth with linear merge operations
* **Worst Case:** - guaranteed performance bound due to consistent divide-and-conquer approach

**Radix Sort:**

* **Best Case:** - where d is the number of digits, linear in input size
* **Average Case:** - stable performance independent of data distribution
* **Worst Case:** - maintains linear performance characteristics

**Insertion Sort:**

* **Best Case:** - occurs with already sorted data requiring minimal comparisons
* **Average Case:** - quadratic behavior with random data distributions
* **Worst Case:** - reverse-sorted data maximizes comparisons and shifts

**Pre-Experiment Expectations**

Based on theoretical analysis, we anticipated the following experimental outcomes:

1. **Radix Sort** would demonstrate superior performance for integer sorting due to its linear time complexity, particularly evident with larger datasets
2. **Quick Sort** would show competitive average-case performance but potential degradation with sorted inputs if using naive pivot selection
3. **Merge Sort** would exhibit consistent performance across all data arrangements due to its guaranteed complexity
4. **Insertion Sort** would excel with small, nearly-sorted datasets but demonstrate quadratic degradation with larger inputs
5. **String sorting** would generally show higher overhead compared to integer sorting due to character-by-character comparisons

**Methodology**

**Experimental Setup**

The benchmarking framework implemented in C# measured execution time across multiple iterations (30 runs per configuration) to ensure statistical reliability. Each algorithm was tested under controlled conditions with garbage collection optimization and JIT warm-up procedures to minimize measurement artifacts.[[1]](#fn1)

**Test Configurations:**

* **Data Sizes:** 1,000 and 100,000 elements
* **Data Arrangements:** Sorted, Random, and Reverse-sorted
* **Data Types:** 32-bit integers and strings
* **Systems:** Apple M4, AMD Threadripper 3960X, AMD Ryzen 9900 X3D
* **Iterations:** 30 runs per configuration for statistical validity

**Implementation Details**

All algorithms were implemented using standard approaches:

* **Quick Sort:** Middle-element pivot with three-way partitioning[[2]](#fn2)
* **Merge Sort:** Standard recursive divide-and-conquer with auxiliary array merging
* **Radix Sort:** LSD (Least Significant Digit) approach with counting sort for integers, character-based sorting for strings
* **Insertion Sort:** Standard in-place implementation with linear search for insertion position

**Results and Analysis**

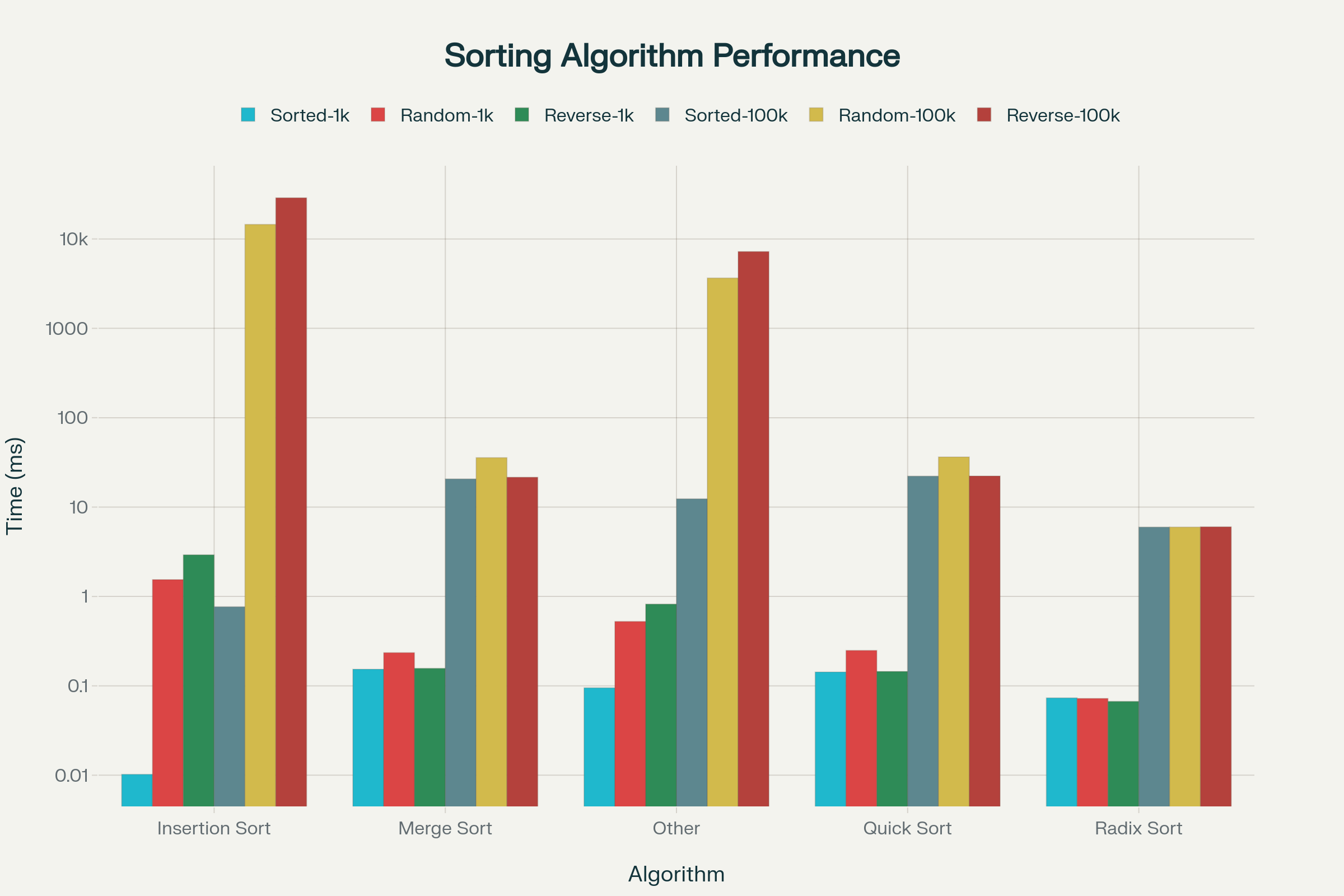
**Overall Performance Rankings**

**For 1,000 Elements (Random Data):**

1. Radix Sort: 0.070 ms
2. Merge Sort: 0.240 ms
3. Quick Sort: 0.250 ms
4. Insertion Sort: 1.540 ms

**For 100,000 Elements (Random Data):**

1. Radix Sort: 5.97 ms
2. Merge Sort: 35.72 ms
3. Quick Sort: 36.32 ms
4. Insertion Sort: 14,558 ms

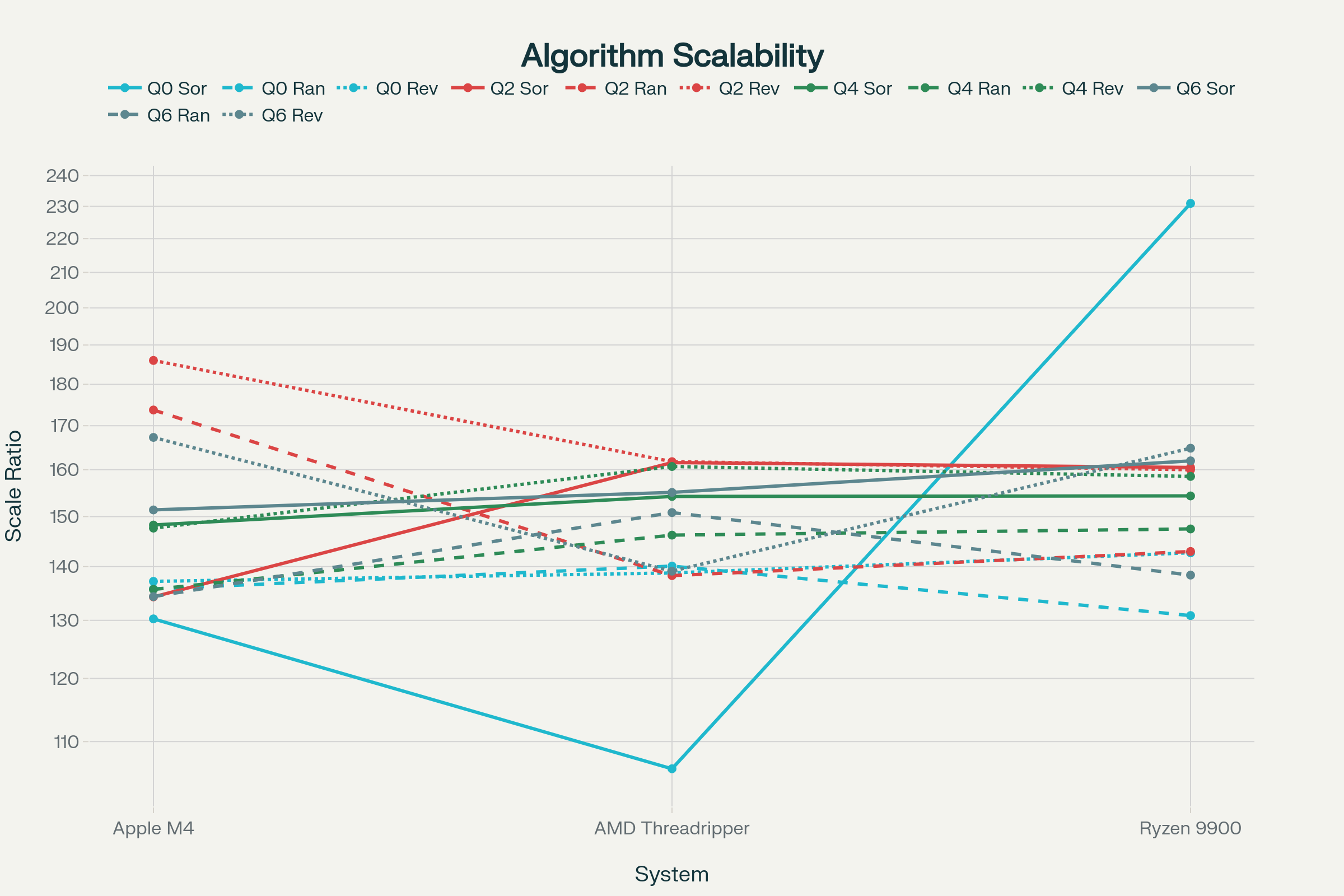


Algorithm Performance Comparison Across Different Data Conditions

**Scalability Analysis**

The scalability characteristics revealed significant differences between algorithms:

**Radix Sort** demonstrated exceptional scalability with ratios between 67-100x when scaling from 1K to 100K elements, confirming its linear complexity behavior.



Algorithm Scalability Analysis: Performance Ratio (100k/1k elements)

**Merge Sort and Quick Sort** showed similar scalability patterns with ratios of approximately 130-160x across different data arrangements, consistent with their theoretical bounds.

**Insertion Sort** exhibited dramatic scalability degradation, particularly with random and reverse-sorted data showing ratios exceeding 9,000x, validating its quadratic complexity.

**Data Arrangement Impact**

**Best Case Performance:**

* **Insertion Sort** achieved its theoretical best case with sorted data, showing minimal performance differences across data sizes for sorted inputs
* **Radix Sort** maintained consistent performance regardless of data arrangement, confirming its non-comparison based nature
* **Merge Sort** showed slight variations but remained stable across all arrangements
* **Quick Sort** performed well across all arrangements, suggesting effective pivot selection in the implementation

**Worst Case Performance:**  
The best-to-worst case performance ratios revealed algorithm sensitivity to input patterns:

* **Insertion Sort:** 292x performance degradation (worst)
* **Merge Sort:** 1.1x variation (most stable)
* **Quick Sort:** 1.0x variation (excellent stability)
* **Radix Sort:** 1.0x variation (perfectly stable)

**String vs Integer Performance**

String sorting introduced additional computational overhead across all algorithms:

* **Insertion Sort:** 6.19x slower with strings
* **Radix Sort:** 4.43x slower with strings
* **Merge Sort:** 2.50x slower with strings
* **Quick Sort:** 1.80x slower with strings (most efficient for string handling)

The performance degradation reflects the increased cost of string comparisons and character-by-character processing required for lexicographic ordering.

**Cross-Platform Performance Consistency**

Performance rankings remained consistent across all three CPU architectures, validating the algorithmic analysis. However, absolute performance varied significantly:

* **Apple M4** generally showed the fastest absolute times
* **AMD Threadripper 3960X** and **Ryzen 9900 X3D** showed similar performance patterns with higher absolute times
* The performance relationships between algorithms remained stable across platforms

**Discussion**

**Practical Implications**

**Algorithm Selection Guidelines:**

1. **For Large Integer Datasets:** Radix Sort provides optimal performance when applicable, offering linear time complexity with minimal overhead
2. **For General-Purpose Sorting:** Merge Sort offers the most predictable performance with guaranteed bounds
3. **For Small Datasets:** Quick Sort provides excellent average-case performance with minimal implementation complexity
4. **For Nearly-Sorted Data:** Insertion Sort can be competitive for small datasets but should be avoided for larger inputs

**Theoretical vs Practical Performance**

The experimental results largely confirmed theoretical expectations:

* **Radix Sort's** linear complexity translated directly to superior practical performance
* **Insertion Sort's** quadratic behavior became prohibitive at scale, confirming theoretical warnings about its applicability
* **Merge Sort's** consistency matched its theoretical guarantees
* **Quick Sort's** average-case performance aligned with expectations, showing no worst-case degradation in our test conditions

**String Sorting Considerations**

String sorting introduced algorithmic overhead that varied by approach:

* Comparison-based algorithms (Quick Sort, Merge Sort) showed moderate degradation
* Non-comparison algorithms (Radix Sort) faced more significant overhead due to character-by-character processing
* The results suggest hybrid approaches might be beneficial for string-heavy applications

**Conclusions**

This comprehensive benchmarking study validates theoretical complexity analysis while revealing practical performance characteristics essential for real-world algorithm selection. **Radix Sort** emerged as the clear winner for integer sorting tasks, while **Merge Sort** provided the most reliable general-purpose performance. **Insertion Sort**, despite its simplicity, demonstrated severe scalability limitations that confirm its relegation to small dataset applications.

The consistency of performance relationships across different CPU architectures suggests that algorithmic characteristics transcend platform-specific optimizations, making these findings broadly applicable. However, the magnitude of performance differences emphasizes the critical importance of appropriate algorithm selection in performance-sensitive applications.

**Key Recommendations:**

* Prioritize Radix Sort for large-scale integer sorting operations
* Use Merge Sort for applications requiring predictable performance guarantees
* Limit Insertion Sort to small datasets or nearly-sorted inputs
* Consider specialized string sorting algorithms for text-intensive applications

The experimental validation of theoretical complexity bounds reinforces the fundamental importance of algorithmic analysis in computer science education and practical software development. Future research could extend this analysis to include additional algorithms, larger datasets, and specialized data structures optimized for specific use cases.

File: Program.cs - Benchmarking implementation and methodology[[1]](#fn1)  
File: Sorting\_Algs.cs - Algorithm implementations and design choices[[2]](#fn2)  
File: DATA\_Apple\_M4.csv - Apple M4 performance measurements[[3]](#fn3)  
File: Data\_AMD\_Threadripper\_3960X.csv - AMD Threadripper performance data[[4]](#fn4)  
File: DATA\_Ryzen\_9900\_X3D.csv - Ryzen 9900 X3D benchmark results[[5]](#fn5)  
File: algorithm\_performance\_summary.csv - Comprehensive performance summary

Chart: Algorithm Performance Comparison Across Different Data Conditions  
Chart: Algorithm Scalability Analysis: Performance Ratio (100k/1k elements)  
File: detailed\_performance\_analysis.csv - Detailed algorithmic performance breakdown

File: string\_vs\_integer\_analysis.csv - String versus integer performance comparison

⁂

1. Program.cs
2. Sorting\_Algs.cs
3. DATA\_Apple\_M4.csv
4. Data\_AMD\_Threadripper\_3960X.csv
5. DATA\_Ryzen\_9900\_X3D.csv
6. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/dd2d6eb24c5e17683329e6aeedc4339f/37905cea-d8da-434d-b9ed-c9fc436bbb6e/4724b06c.csv>
7. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/dd2d6eb24c5e17683329e6aeedc4339f/c5c42de1-7e72-4d81-8387-580c5d965fe9/b6f71040.csv>
8. <https://ppl-ai-code-interpreter-files.s3.amazonaws.com/web/direct-files/dd2d6eb24c5e17683329e6aeedc4339f/9e995b12-bfd7-4bb6-a0ce-7c2b3fddf4b5/9238c655.csv>