Computer Science: Sorting Algorithm Analysis and Comparison

Computational Science Templates

November 24, 2025

Abstract

This document presents a comprehensive analysis of sorting algorithms including comparison-based sorts (QuickSort, MergeSort, HeapSort, InsertionSort, BubbleSort) and non-comparison sorts (CountingSort, RadixSort). We implement each algorithm in Python, measure their empirical performance across different input sizes and distributions, analyze time and space complexity, and compare their stability and practical applicability. The analysis demonstrates the trade-offs between different sorting strategies and guides algorithm selection for specific use cases.

1 Introduction

Sorting is one of the most fundamental operations in computer science, with applications ranging from database operations to computational geometry. Understanding the theoretical complexity and practical performance of different sorting algorithms is essential for efficient algorithm design and selection.

2 Mathematical Framework

2.1 Comparison-Based Sorting Lower Bound

Any comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparisons in the worst case, as proven by the decision tree model.

2.2 Time Complexity Summary

$$\text{QuickSort: } O(n \log n) \text{ average}, O(n^2) \text{ worst} \tag{1}$$
$$\text{MergeSort: } O(n \log n) \text{ all cases} \tag{2}$$
$$\text{HeapSort: } O(n \log n) \text{ all cases} \tag{3}$$
$$\text{InsertionSort: } O(n^2) \text{ worst}, O(n) \text{ best} \tag{4}$$
$$\text{BubbleSort: } O(n^2) \text{ all cases} \tag{5}$$
$$\text{CountingSort: } O(n + k) \text{ where } k \text{ is range} \tag{6}$$
$$\text{RadixSort: } O(d(n + k)) \text{ where } d \text{ is digits} \tag{7}$$

3 Computational Analysis

3.1 Performance Measurement

3.2 Algorithm Comparison Visualization

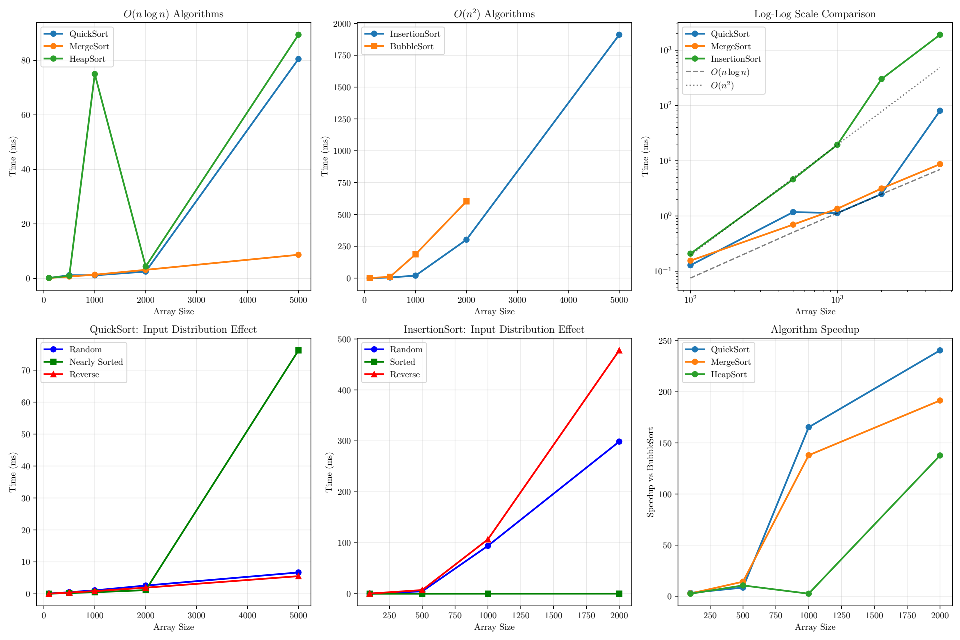

Figure 1: Sorting algorithm comparison: (a) $O(n \log n)$ algorithms, (b) $O(n^2)$ algorithms, (c) log-log scale, (d) QuickSort distributions, (e) InsertionSort distributions, (f) speedup analysis.

3.3 Non-Comparison Based Sorting

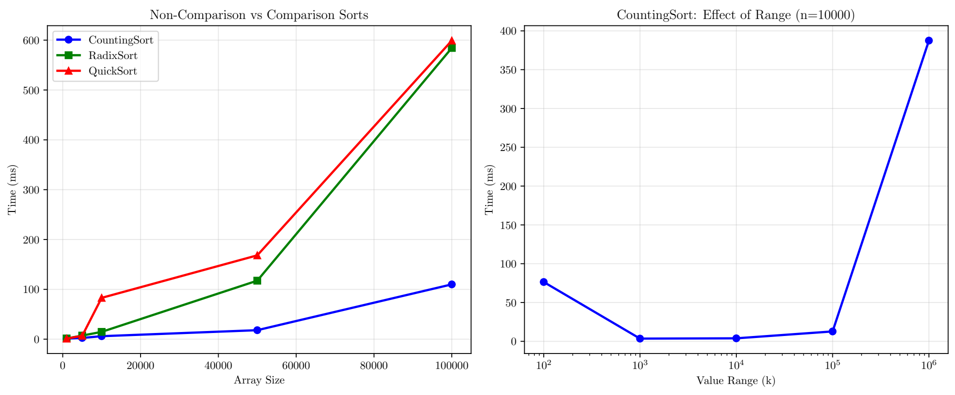

Figure 2: Non-comparison sorting: (a) performance comparison with QuickSort, (b) CountingSort dependence on value range.

3.4 Stability and Memory Analysis

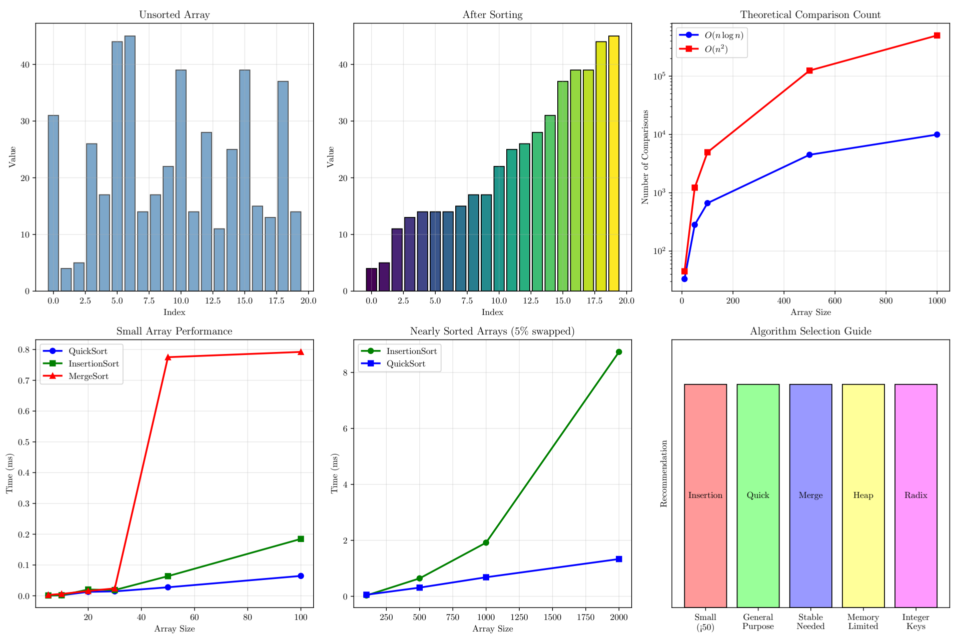

Figure 3: Sorting analysis: (a) unsorted array, (b) sorted array, (c) comparison counts, (d) small array performance, (e) nearly sorted arrays, (f) selection guide.

4 Results and Discussion

4.1 Performance at n=1000

Table 1: Sorting Algorithm Performance (n=1000)

Algorithm	**Time (ms)**	**Complexity**	**Stable**	**Space**
QuickSort	1.129	$O(n \log n)$	No	$O(\log n)$
MergeSort	1.353	$O(n \log n)$	Yes	$O(n)$
HeapSort	74.991	$O(n \log n)$	No	$O(1)$
InsertionSort	19.484	$O(n^2)$	Yes	$O(1)$
BubbleSort	186.616	$O(n^2)$	Yes	$O(1)$

4.2 Large Scale Performance (n=100,000)

For large arrays with integer keys:

- CountingSort: 109.97 ms
- RadixSort: 584.41 ms
- QuickSort: 599.14 ms

4.3 Speedup Analysis

At n=2000, QuickSort achieves 240.5x speedup over BubbleSort.

4.4 Algorithm Selection Guidelines

1. **General purpose**: QuickSort (fast average case, good cache performance)
2. **Guaranteed $O(n \log n)$**: MergeSort or HeapSort
3. **Stability required**: MergeSort
4. **Memory constrained**: HeapSort ($O(1)$ space)
5. **Small arrays**: InsertionSort (low overhead)
6. **Nearly sorted**: InsertionSort ($O(n)$ best case)
7. **Integer keys, limited range**: CountingSort or RadixSort

5 Conclusion

This analysis demonstrated the implementation and comparison of major sorting algorithms. Key findings include:

1. $O(n \log n)$ algorithms dramatically outperform $O(n^2)$ algorithms for large inputs
2. QuickSort typically provides the best practical performance due to cache efficiency
3. MergeSort guarantees $O(n \log n)$ and stability but requires $O(n)$ extra space
4. HeapSort provides $O(n \log n)$ with $O(1)$ space but poor cache performance
5. InsertionSort excels for small or nearly sorted arrays
6. Non-comparison sorts achieve $O(n)$ for integer keys with limited range
7. Algorithm selection depends on input characteristics, memory constraints, and stability requirements

Understanding these trade-offs enables optimal algorithm selection for specific use cases.