Performance Analysis of Sorting Algorithms

Report No. TR-2024-03
February 10, 2024

Executive Summary

This report presents a comprehensive analysis of common sorting algorithm performance characteristics. We evaluate time complexity, space complexity, and real-world performance across different input sizes and data distributions.

Key Findings:
- QuickSort performs best for random data (average case)
- MergeSort provides consistent $O(n \log n)$ performance
- RadixSort excels for integer sorting with limited range

1. Introduction

Sorting is a fundamental operation in computer science with applications spanning databases, search engines, and data analysis. Understanding the performance characteristics of different sorting algorithms is essential for selecting the appropriate algorithm for specific use cases.

1.1 Scope

This study examines:
- Comparison-based sorts: QuickSort, MergeSort, HeapSort
- Non-comparison sorts: RadixSort, CountingSort
- Simple sorts: InsertionSort, SelectionSort

1.2 Methodology

All algorithms were implemented in C++ and compiled with GCC 11.2 with -O2 optimization. Tests ran on:

- CPU: Intel Core i7-9700K @ 3.60GHz
- RAM: 32GB DDR4-3200
- OS: Ubuntu 22.04 LTS

2. Theoretical Analysis

2.1 Time Complexity

Algorithm	Best Case	Average Case	Worst Case
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
RadixSort	$O(dn)$	$O(dn)$	$O(dn)$

where d is the number of digits in RadixSort.

2.2 Space Complexity

Algorithm	Space Complexity
QuickSort	$O(\log n)$
MergeSort	$O(n)$
HeapSort	$O(1)$
InsertionSort	$O(1)$
RadixSort	$O(n + k)$

3. Experimental Results

3.1 Random Input Performance

Testing with arrays of random integers:

Algorithm	n=1K	n=10K	n=100K	n=1M
QuickSort	0.12ms	1.45ms	18.2ms	215ms
MergeSort	0.15ms	1.82ms	22.1ms	258ms
HeapSort	0.18ms	2.34ms	29.8ms	342ms
InsertionSort	2.4ms	245ms	24.5s	—
RadixSort	0.08ms	0.95ms	11.2ms	128ms

3.2 Nearly Sorted Input

Performance on arrays that are 90% sorted:

- InsertionSort: Excellent performance ($\approx O(n)$)
- QuickSort: Poor performance (approaches $O(n^2)$)
- MergeSort: Consistent performance

3.3 Reverse Sorted Input

Worst case for some algorithms:

- QuickSort (naive pivot): $O(n^2)$ confirmed
- MergeSort: Unaffected
- InsertionSort: Maximum comparisons

4. Algorithm Analysis

4.1 QuickSort

QuickSort's partitioning step:

```
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Advantages:
- In-place sorting
- Cache-friendly
- Fast average case

Disadvantages:
- Unstable
- Poor worst-case performance

4.2 MergeSort

The merge operation:

$$\text{merge}(A[1..m], B[1..n]) = C[1..m+n]$$

where C contains all elements from A and B in sorted order.

Advantages:
- Stable sort
- Guaranteed $O(n \log n)$
- Parallelizable

Disadvantages:
- Requires $O(n)$ extra space
- Slower than QuickSort in practice

4.3 RadixSort

For d-digit numbers with base b:

Total time: $O(d(n + b))$

When d and b are constants: $O(n)$ linear time!

5. Recommendations

Based on our analysis:

> **General Purpose:** Use QuickSort (with random pivot selection)
>
> **Guaranteed Performance:** Use MergeSort or HeapSort
>
> **Small Arrays (n < 50):** Use InsertionSort
>
> **Integer Sorting:** Use RadixSort when range is limited
>
> **Nearly Sorted:** Use InsertionSort or TimSort

6. Conclusion

No single sorting algorithm is optimal for all scenarios. The choice depends on:

1. Input size
2. Data distribution
3. Memory constraints
4. Stability requirements
5. Parallelization needs

Modern hybrid algorithms like TimSort (used in Python and Java) combine multiple approaches to achieve robust performance across diverse inputs.

7. Future Work

Future investigations will explore:

- Parallel sorting algorithms
- External sorting for datasets exceeding RAM
- Adaptive algorithms that detect input patterns
- GPU-accelerated sorting

Appendix A: Mathematical Proofs

QuickSort Average Case

Let $T(n)$ be the expected runtime. For random pivot:

$$T(n) = n + \frac{1}{n}\sum_{i=0}^{n-1}[T(i) + T(n-i-1)]$$
$$= n + \frac{2}{n}\sum_{i=0}^{n-1} T(i)$$

Solving this recurrence yields $T(n) = O(n \log n)$.

Appendix B: Test Data Generation

Pseudorandom number generator (Xorshift):

```
uint32_t xorshift32(uint32_t state) {
    state ^= state << 13;
    state ^= state >> 17;
    state ^= state << 5;
    return state;
}
```