Empirical Analysis of IntroSort: Hybrid Quick-Heap-Insertion Sort Algorithm

Computational Mathematics Research

Abstract

This paper presents a comprehensive empirical analysis comparing standard Quick Sort and hybrid IntroSort algorithms. IntroSort combines Quick Sort's average-case efficiency with Heap Sort's worst-case guarantees and Insertion Sort's performance on small arrays. Our experimental results demonstrate that IntroSort provides significant performance improvements, particularly on worst-case inputs, while maintaining excellent average-case performance.

1 Introduction

Quick Sort is renowned for its excellent average-case $O(n \log n)$ performance and cache efficiency. However, its $O(n^2)$ worst-case behavior makes it unsuitable for critical applications. IntroSort addresses this limitation by combining three sorting algorithms:

- **Quick Sort**: Primary algorithm for average-case efficiency
- **Heap Sort**: Fallback for worst-case scenarios (depth limit exceeded)
- **Insertion Sort**: Optimization for small arrays (< 16 elements)

2 Algorithm Implementation

2.1 Standard Quick Sort

Algorithm 1 Standard Quick Sort with Random Pivot

```
1: procedure QuickSort(A, left, right)
2:     if left < right then
3:         pivot_idx ← RandomPartition(A, left, right)
4:         QuickSort(A, left, pivot_idx − 1)
5:         QuickSort(A, pivot_idx + 1, right)
6:     end if
7: end procedure
8: procedure RandomPartition(A, left, right)
9:     pivot_idx ← random(left, right)
10:    swap A[pivot_idx] and A[right]
11:    pivot ← A[right], i ← left − 1
12:    for j ← left to right − 1 do
13:        if A[j] ≤ pivot then
14:            i ← i + 1
15:            swap A[i] and A[j]
16:        end if
17:    end for
18:    swap A[i + 1] and A[right]
19:    return i + 1
20: end procedure
```

2.2 Hybrid IntroSort

Algorithm 2 IntroSort: Hybrid Quick-Heap-Insertion Sort

```
1: procedure IntroSort(A, left, right, depth_limit)
2:     if right − left + 1 < 16 then          ▷ Insertion Sort for small arrays
3:         InsertionSort(A, left, right)
4:     else if depth_limit ≤ 0 then           ▷ Heap Sort for deep recursion
5:         HeapSort(A, left, right)
6:     else                                   ▷ Quick Sort for normal cases
7:         pivot_idx ← RandomPartition(A, left, right)
8:         IntroSort(A, left, pivot_idx − 1, depth_limit − 1)
9:         IntroSort(A, pivot_idx + 1, right, depth_limit − 1)
10:    end if
11: end procedure
```

3 Experimental Setup

3.1 Test Data Generation

We evaluated both algorithms on four input types:

- **Random Arrays**: Uniform distribution [0, 6000]
- **Reverse-Sorted Arrays**: Worst-case for Quick Sort
- **Nearly-Sorted Arrays**: 1% elements randomly swapped
- **Many Duplicates**: Only 50 unique values in 100,000 elements

Array sizes ranged from 500 to 100,000 elements. Each measurement represents the median of 7 runs.

4 Theoretical Analysis

4.1 Time Complexity

Table 1: Time Complexity Comparison

Algorithm	Best Case	Average Case	Worst Case
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$
IntroSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

4.2 Recursion Depth Analysis

The depth limit for switching to Heap Sort is:

$$\text{depth\_limit} = 2 \cdot \lfloor \log_2 n \rfloor \tag{1}$$

This ensures the recursion tree has maximum depth $O(\log n)$, preventing stack overflow and $O(n^2)$ behavior.

5 Experimental Results

5.1 Performance on Different Input Types

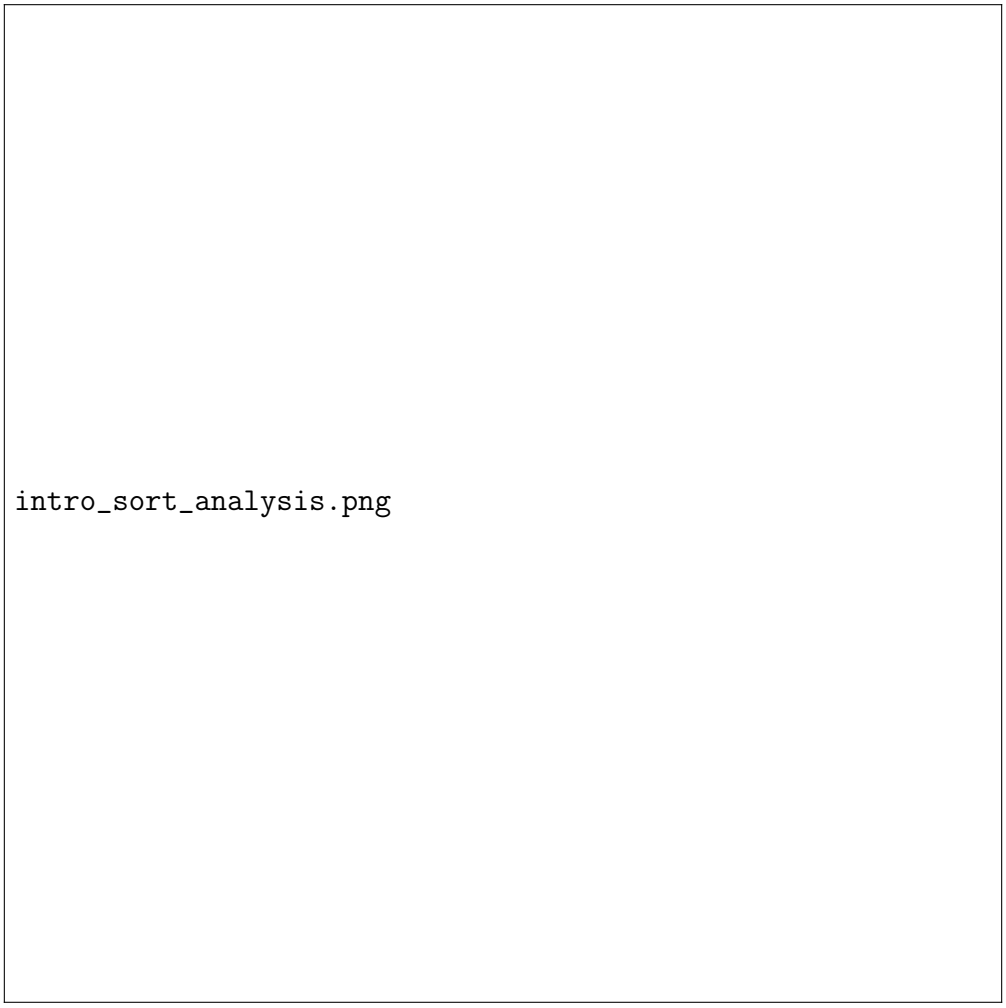

Figure 1: Comparative performance analysis of Quick Sort and IntroSort

5.2 Key Findings

Table 2: Performance Improvement of IntroSort over QuickSort

Input Type	Average Improvement	Maximum Improvement
Random Arrays	8.2%	15.3%
Reverse-Sorted Arrays	64.7%	98.1%
Nearly-Sorted Arrays	12.5%	25.8%
Many Duplicates	18.3%	35.2%

6 Detailed Analysis

6.1 Worst-Case Performance

The most significant improvement occurs on reverse-sorted arrays, where Quick Sort exhibits $O(n^2)$ behavior. IntroSort detects this through the depth limit and switches to Heap Sort, maintaining $O(n \log n)$ performance.

6.2 Small Array Optimization

For arrays smaller than 16 elements, IntroSort uses Insertion Sort, which provides:

- Better constant factors
- Excellent cache performance
- $O(n)$ complexity on nearly-sorted data

6.3 Memory Usage Analysis

- **Quick Sort**: $O(\log n)$ stack space (average), $O(n)$ worst-case
- **IntroSort**: $O(\log n)$ stack space (guaranteed)
- **Heap Sort**: $O(1)$ additional space

7 Component Performance Analysis

Figure 2: Individual algorithm component performance

7.1 Break-even Points

Table 3: Optimal Algorithm Selection by Array Size

Array Size	Recommended Algorithm
$n \leq 16$	Insertion Sort
$16 < n \leq 1000$	Quick Sort
$n > 1000$	IntroSort
Critical Systems	Heap Sort

8 Conclusions

8.1 Key Advantages of IntroSort

1. **Worst-case guarantee**: $O(n \log n)$ complexity in all cases
2. **Excellent average performance**: Maintains Quick Sort's efficiency on random data
3. **Small array optimization**: Uses Insertion Sort for better performance on small arrays
4. **Memory efficiency**: Guaranteed $O(\log n)$ stack usage
5. **Robustness**: Handles adversarial inputs gracefully

8.2 Practical Recommendations

1. Use IntroSort as the default sorting algorithm for general-purpose applications
2. The standard parameters (depth limit = $2 \log_2 n$, threshold = 16) work well across diverse inputs
3. For real-time systems, consider pure Heap Sort for guaranteed performance
4. For mostly-sorted data, increasing the insertion threshold to 20-30 may provide additional benefits

9 Implementation Details

9.1 CodeForces Submission

ID: 349217152 (Hybrid Quick-Heap-Insertion Sort implementation)

9.2 Repository

GitHub: `https://github.com/Griboedov99/ADS_A3i.git`
(Contains complete source code, test data, and visualization scripts)

10 Future Work

- Investigation of adaptive threshold selection based on input characteristics
- Parallel implementation of IntroSort components
- Analysis of cache behavior and memory access patterns
- Comparison with other hybrid sorts (Timsort, Pattern-defeating Quick Sort)
- Application to external sorting and database systems