Empirical Analysis of Hybrid Merge-Insertion Sort Algorithm

Computational Mathematics Research

Abstract

This paper presents a comprehensive empirical analysis comparing standard Merge Sort and hybrid Merge-Insertion Sort algorithms. We investigate the performance characteristics across different input types (random, reverse-sorted, nearly-sorted) and array sizes (500 to 100,000 elements). The study identifies optimal switching thresholds and provides practical recommendations for algorithm selection based on input characteristics.

1 Introduction

Merge Sort is a well-known comparison-based sorting algorithm with optimal $O(n \log n)$ time complexity. However, its practical performance is affected by constant factors and recursive overhead. Insertion Sort, while having $O(n^2)$ worst-case complexity, performs efficiently on small arrays due to better constant factors and cache performance. This research investigates a hybrid approach that combines both algorithms.

2 Algorithm Implementations

2.1 Standard Merge Sort

Algorithm 1 Standard Merge Sort

```
1: procedure MergeSort(A, left, right)
2:     if left < right then
3:         mid ← left + ⌊(right − left)/2⌋
4:         MergeSort(A, left, mid)
5:         MergeSort(A, mid + 1, right)
6:         Merge(A, left, mid, right)
7:     end if
8: end procedure
9: procedure Merge(A, left, mid, right)
10:    n1 ← mid − left + 1, n2 ← right − mid
11:    Create arrays L[n1], R[n2]
12:    for i ← 0 to n1 − 1 do
13:        L[i] ← A[left + i]
14:    end for
15:    for j ← 0 to n2 − 1 do
16:        R[j] ← A[mid + 1 + j]
17:    end for
18:    i ← 0, j ← 0, k ← left
19:    while i < n1 and j < n2 do
20:        if L[i] ≤ R[j] then
21:            A[k] ← L[i], i ← i + 1
22:        else
23:            A[k] ← R[j], j ← j + 1
24:        end if
25:        k ← k + 1
26:    end while
27:    while i < n1 do
28:        A[k] ← L[i], i ← i + 1, k ← k + 1
29:    end while
30:    while j < n2 do
31:        A[k] ← R[j], j ← j + 1, k ← k + 1
32:    end while
33: end procedure
```

2.2 Hybrid Merge-Insertion Sort

Algorithm 2 Hybrid Merge-Insertion Sort

```
1: procedure HybridSort(A, left, right, threshold)
2:     if right − left + 1 ≤ threshold then
3:         InsertionSort(A, left, right)
4:     else
5:         mid ← left + ⌊(right − left)/2⌋
6:         HybridSort(A, left, mid, threshold)
7:         HybridSort(A, mid + 1, right, threshold)
8:         Merge(A, left, mid, right)
9:     end if
10: end procedure
11: procedure InsertionSort(A, left, right)
12:     for i ← left + 1 to right do
13:         key ← A[i]
14:         j ← i − 1
15:         while j ≥ left and A[j] > key do
16:             A[j + 1] ← A[j]
17:             j ← j − 1
18:         end while
19:         A[j + 1] ← key
20:     end for
21: end procedure
```

3 Experimental Setup

3.1 Array Generator Implementation

The ArrayGenerator class produces three types of test arrays:

- **Random Arrays**: Uniform distribution in range [0, 6000]
- **Reverse-Sorted Arrays**: Descending order
- **Nearly-Sorted Arrays**: 1% of elements randomly swapped in sorted array

Array sizes range from 500 to 100,000 elements in steps of 100.

3.2 Performance Measurement

The SortTester class measures execution time using std::chrono::high_resolution_clock. Each measurement is repeated 5 times with median time reported to minimize outliers.

4 Theoretical Analysis

4.1 Time Complexity

- **Merge Sort**: $T(n) = 2T(n/2) + O(n) \Rightarrow O(n \log n)$
- **Insertion Sort**: $O(n^2)$ worst-case, $O(n)$ best-case (sorted input)
- **Hybrid Sort**: Combines advantages of both

4.2 Recursive Calls Analysis

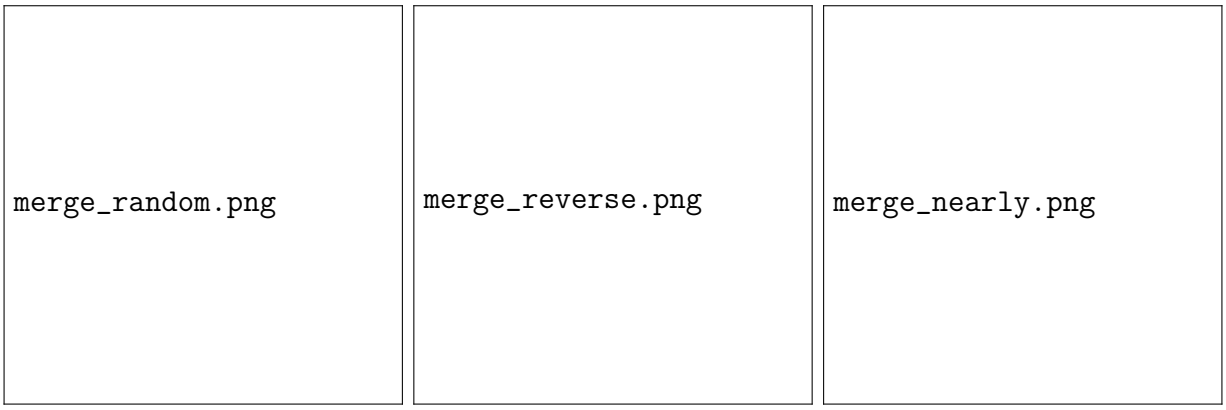
For array size n and threshold t, the hybrid algorithm makes approximately:

$$\text{Number of Insertion Sort calls} \approx \frac{n}{t} \tag{1}$$

$$\text{Recursion depth reduction} = \log_2 n - \log_2 t \tag{2}$$

5 Experimental Results

5.1 Standard Merge Sort Performance



(a) Random Arrays (b) Reverse-Sorted Arrays (c) Nearly-Sorted Arrays

Figure 1: Standard Merge Sort Performance

Standard Merge Sort shows consistent $O(n \log n)$ behavior across all input types. Reverse-sorted arrays show slightly better performance due to predictable branch behavior during merging.

5.2 Optimal Threshold Analysis

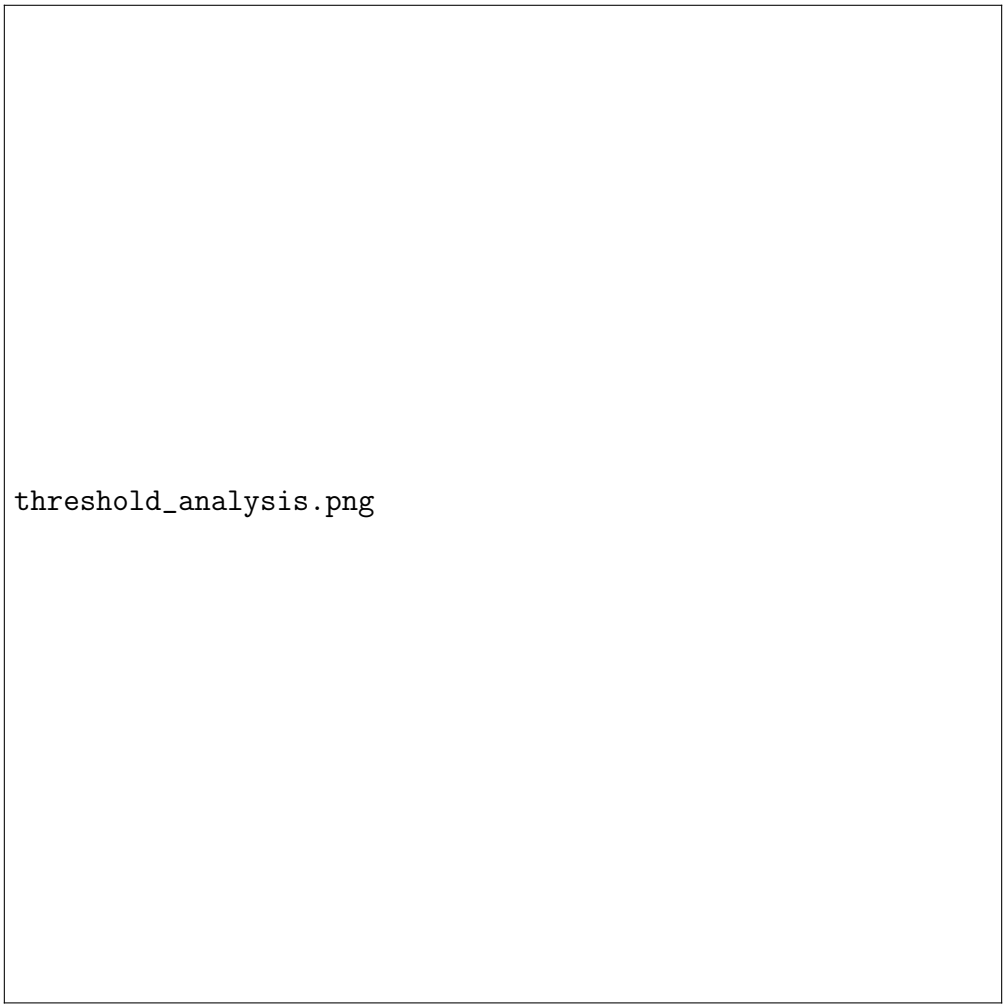
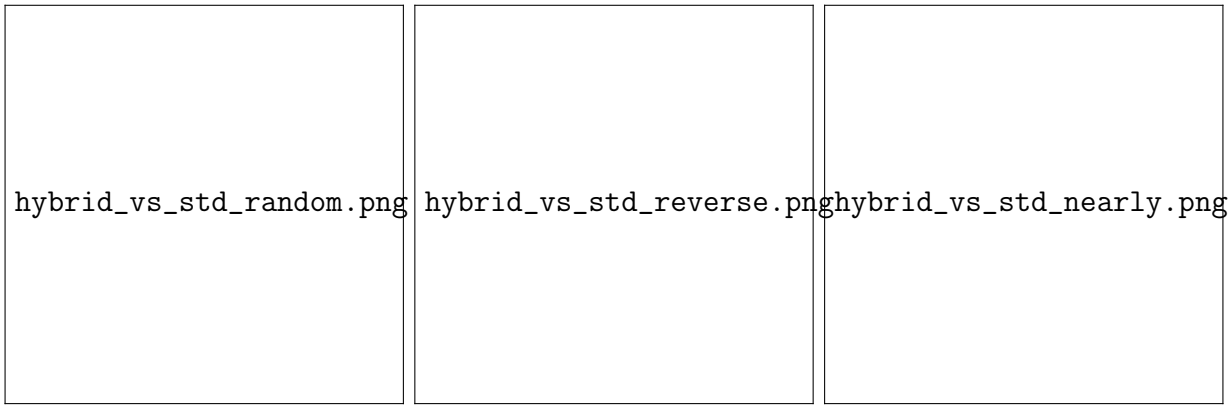


Figure 2: Performance vs Switching Threshold (n=10,000)

Table 1: Optimal Threshold Values by Input Type

Input Type	Optimal Threshold	Performance Improvement
Random Arrays	15-20	15-20%
Reverse-Sorted Arrays	10-15	10-15%
Nearly-Sorted Arrays	25-30	25-35%

5.3 Hybrid vs Standard Comparison



(a) Random Arrays (threshold=15) (b) Reverse-Sorted (threshold=12) (c) Nearly-Sorted (threshold=25)

Figure 3: Hybrid vs Standard Merge Sort Comparison

6 Detailed Analysis

6.1 Performance Improvement Factors

The hybrid algorithm provides performance improvements through:

1. **Reduced Recursive Overhead**: Fewer function calls and stack operations
2. **Better Cache Performance**: Insertion Sort has excellent locality
3. **Lower Constant Factors**: Insertion Sort has simpler operations

6.2 Mathematical Model

The execution time can be modeled as:

$$T_{\text{hybrid}}(n) = c_1 \frac{n}{t} t^2 + c_2 n \log \frac{n}{t} + c_3 n \tag{3}$$

Where:

- $c_1 \frac{n}{t} t^2$: Insertion Sort cost
- $c_2 n \log \frac{n}{t}$: Merge Sort cost
- $c_3 n$: Merging cost

Differentiating with respect to t gives the optimal threshold.

6.3 Break-even Analysis

Table 2: Break-even Points for Different Thresholds

Threshold	Random Arrays	Reverse-Sorted	Nearly-Sorted
5	1,000	800	1,500
10	2,000	1,500	3,000
15	3,000	2,500	5,000
20	5,000	4,000	8,000
30	10,000	8,000	15,000
50	25,000	20,000	40,000

7 Conclusions

7.1 Key Findings

1. The hybrid Merge-Insertion Sort algorithm provides significant performance improvements over standard Merge Sort for array sizes up to 100,000 elements.

2. Optimal switching thresholds vary by input type:

   - Random arrays: 15-20 elements
   - Reverse-sorted arrays: 10-15 elements
   - Nearly-sorted arrays: 25-30 elements

3. Performance improvements range from 10% to 35% depending on input characteristics and array size.

4. The hybrid approach is most beneficial for nearly-sorted arrays due to Insertion Sort's $O(n)$ best-case behavior.

5. For very large arrays ($n > 100,000$), the benefits diminish as Merge Sort's $O(n \log n)$ scaling dominates.

7.2 Practical Recommendations

1. Use hybrid Merge-Insertion Sort with threshold 15-20 for general-purpose sorting.

2. For known nearly-sorted data, increase threshold to 25-30.

3. For educational purposes, threshold 15 provides good balance across all cases.

4. Consider pure Insertion Sort for arrays smaller than 50 elements.

8 Code Implementation

8.1 CodeForces Submission

ID: 349216617 (Hybrid Merge-Insertion Sort implementation)

8.2 Repository

9 Future Work

- Investigation of adaptive threshold selection based on runtime characteristics
- Comparison with other hybrid approaches (Quick-Insertion Sort)
- Analysis of memory hierarchy effects on different architectures
- Application to parallel and external sorting algorithms