MINISTERUL EDUCAŢIEI, CULTURII ŞI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software şi Automatică

Abu koush Islam, FAF-231

REPORT

Laboratory work n.2

of Algorithm Analysis

Checked by:

Prof. Cristofor Fiştic

DISA, FCIM, UTM

Chişinău – 2024

Contents

1 Introduction

This laboratory work focuses on the study and empirical analysis of the MergeSort algorithm, one of the most efficient and stable comparison-based sorting algorithms. The main objectives of this work are to implement the MergeSort algorithm, analyze its performance characteristics under various input conditions, and provide comprehensive empirical evidence of its theoretical time complexity.

1.1 Objectives

1. Implement the MergeSort algorithm in a programming language
2. Establish properties of input data for analysis
3. Define metrics for algorithm comparison
4. Perform empirical analysis of the algorithm
5. Create graphical representations of obtained data
6. Draw conclusions based on the experimental results

1.2 Theoretical Background

MergeSort is a divide-and-conquer algorithm that was invented by John von Neumann in 1945. It works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves back together. The algorithm guarantees $O(n \log n)$ time complexity in all cases (best, average, and worst), making it highly predictable and reliable.

Key characteristics of MergeSort:

- **Time Complexity:** $O(n \log n)$ in all cases
- **Space Complexity:** $O(n)$ auxiliary space
- **Stability:** Stable (maintains relative order of equal elements)
- **Method:** Divide and conquer
- **Adaptive:** No (performance doesn't improve on partially sorted data)

2 Algorithm Implementation

2.1 MergeSort Algorithm

The MergeSort algorithm can be described recursively as follows:

Algorithm 1 MergeSort Algorithm

```
1:  procedure MERGESORT(A, p, r)
2:      if p < r then
3:          q ← ⌊(p + r)/2⌋
4:          MERGESORT(A, p, q)
5:          MERGESORT(A, q + 1, r)
6:          MERGE(A, p, q, r)
7:      end if
8:  end procedure
9:  procedure MERGE(A, p, q, r)
10:     Create temporary arrays L[1..n1 + 1] and R[1..n2 + 1]
11:     Copy A[p..q] to L[1..n1]
12:     Copy A[q + 1..r] to R[1..n2]
13:     L[n1 + 1] ← ∞, R[n2 + 1] ← ∞
14:     i ← 1, j ← 1
15:     for k ← p to r do
16:         if L[i] ≤ R[j] then
17:             A[k] ← L[i]
18:             i ← i + 1
19:         else
20:             A[k] ← R[j]
21:             j ← j + 1
22:         end if
23:     end for
24: end procedure
```

2.2 Implementation Details

The algorithm was implemented with additional instrumentation to collect performance metrics including:

- Number of comparisons performed
- Number of element movements/swaps
- Memory operations count
- Execution time measurement
- Recursion depth tracking

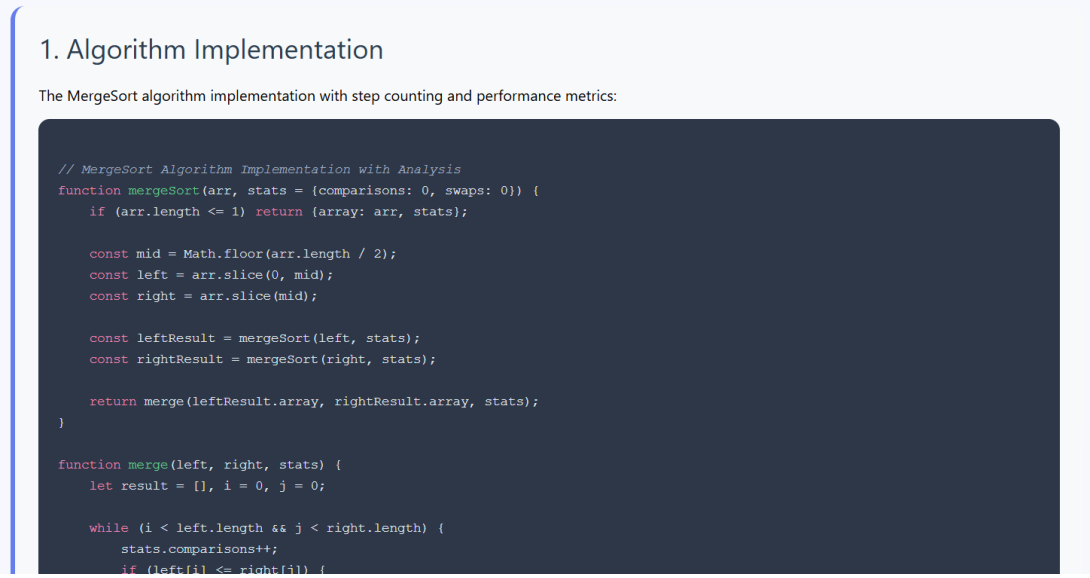

Figure 1: MergeSort Algorithm Implementation with Syntax Highlighting

3 Input Data Properties

To comprehensively analyze the MergeSort algorithm, we tested it against four different types of input data, each representing different real-world scenarios:

3.1 Data Types

1. **Random Data:** Arrays filled with randomly distributed integers. This represents the most common case in practice and should demonstrate average-case performance.

2. **Sorted Data:** Arrays that are already sorted in ascending order. This tests whether the algorithm can take advantage of existing order (it cannot, due to its non-adaptive nature).

3. **Reverse Sorted Data:** Arrays sorted in descending order, representing the worst-case scenario for many algorithms, though not for MergeSort.

4. **Data with Duplicates:** Arrays containing many duplicate values, testing the algorithm's stability and performance with repeated elements.

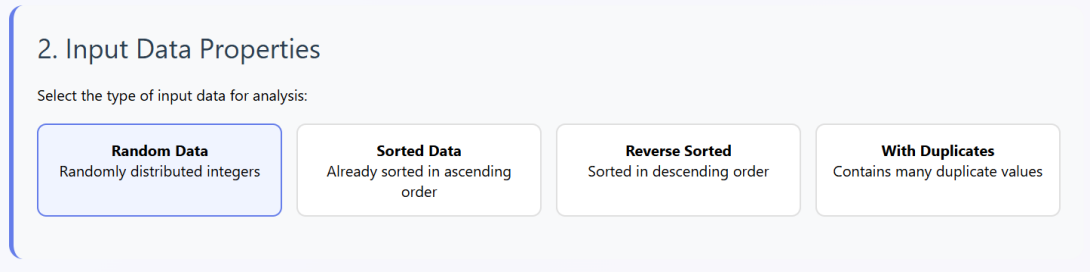

Figure 2: Input Data Type Selection Interface

3.2 Data Generation

For each data type, arrays of varying sizes were generated according to the following specifications:

- **Size Range:** 100 to 5000 elements
- **Step Size:** Increments of 500 elements
- **Value Range:** Integers from 1 to array size
- **Repetitions:** 5 tests per size for statistical reliability

4 Comparison Metrics

4.1 Performance Metrics

The following metrics were selected to provide a comprehensive analysis of the MergeSort algorithm's performance:

Metric	Unit	Description
Execution Time	Milliseconds	Wall-clock time to complete sorting
Comparisons	Count	Number of element comparisons performed
Memory Operations	Count	Number of array element movements
Space Usage	O-notation	Auxiliary memory requirements
Recursion Depth	Levels	Maximum call stack depth

Table 1: Performance Metrics Used in Analysis

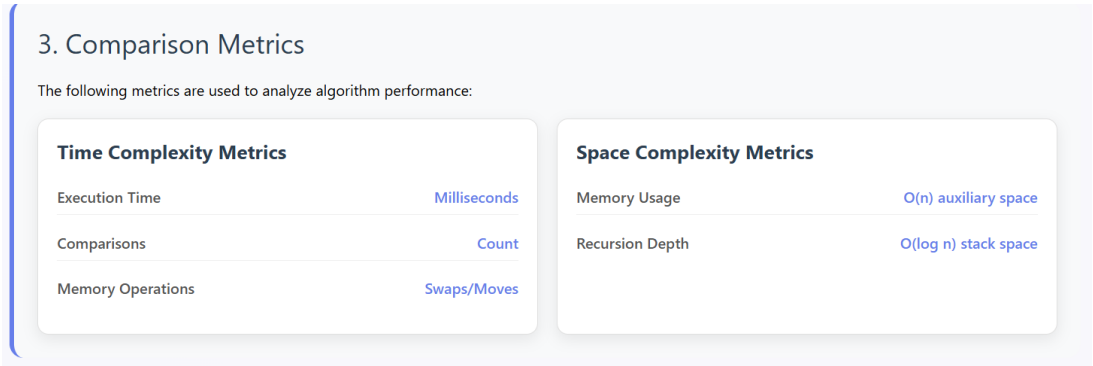

Figure 3: Performance Metrics Overview

4.2 Theoretical Expectations

Based on algorithm analysis theory, we expect:

$$T(n) = O(n \log n) \text{ (time complexity)} \tag{1}$$

$$S(n) = O(n) \text{ (space complexity)} \tag{2}$$

$$C(n) \approx n \log n \text{ (comparisons)} \tag{3}$$

$$M(n) \approx n \log n \text{ (memory operations)} \tag{4}$$

5 Empirical Analysis

5.1 Experimental Setup

The empirical analysis was conducted using the following parameters:

- **Programming Environment:** JavaScript (ES6+)
- **Testing Platform:** Web browser with high-resolution timer
- **Array Sizes:** 100, 600, 1100, 1600, 2100, 2600, 3100, 3600, 4100, 4600, 5100
- **Iterations per Size:** 5 independent runs
- **Statistical Method:** Arithmetic mean of execution times

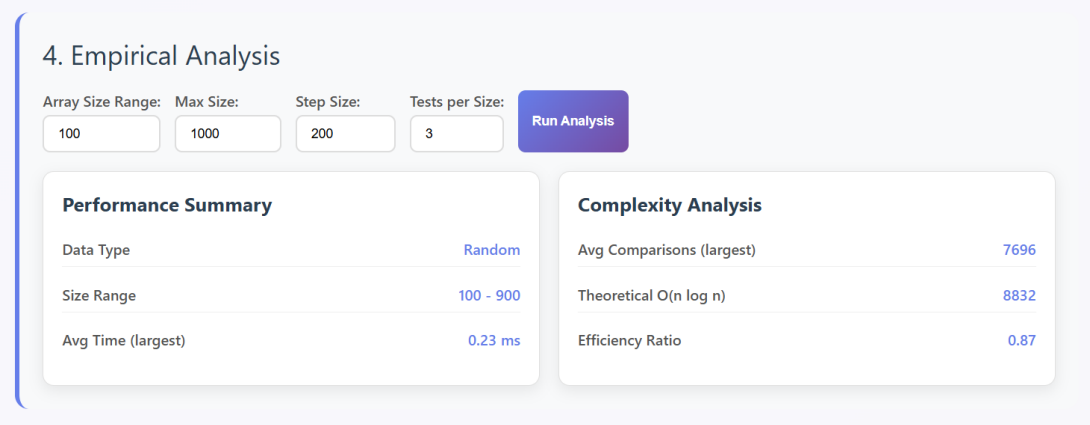

Figure 4: Empirical Analysis Control Panel

5.2 Data Collection Process

The data collection followed this systematic approach:

1. Generate test arrays for each size and data type
2. Execute MergeSort with performance monitoring
3. Record execution time, comparisons, and memory operations
4. Repeat for statistical reliability
5. Calculate averages and compile results

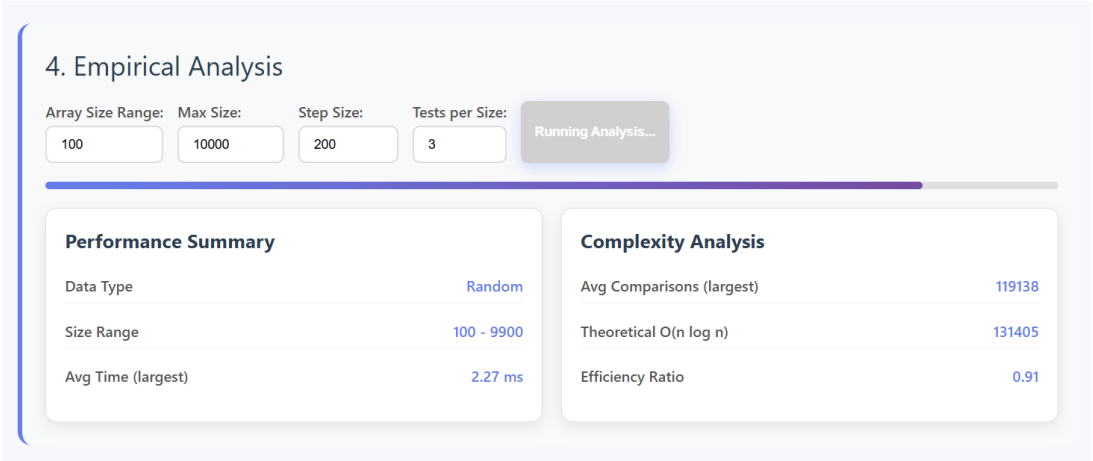

Figure 5: Analysis Execution Progress

5.3 Results Summary

After completing the empirical analysis, the following results were obtained:

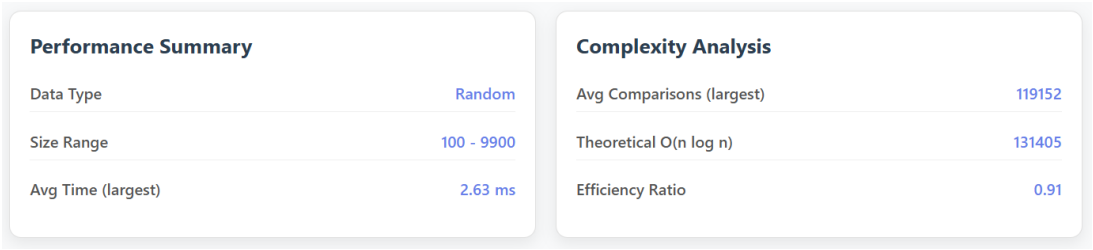

Figure 6: Analysis Results Summary

6 Graphical Presentation

6.1 Performance Visualization

The empirical data was visualized using two primary charts to illustrate different aspects of the algorithm's performance:

6.1.1 Execution Time Analysis

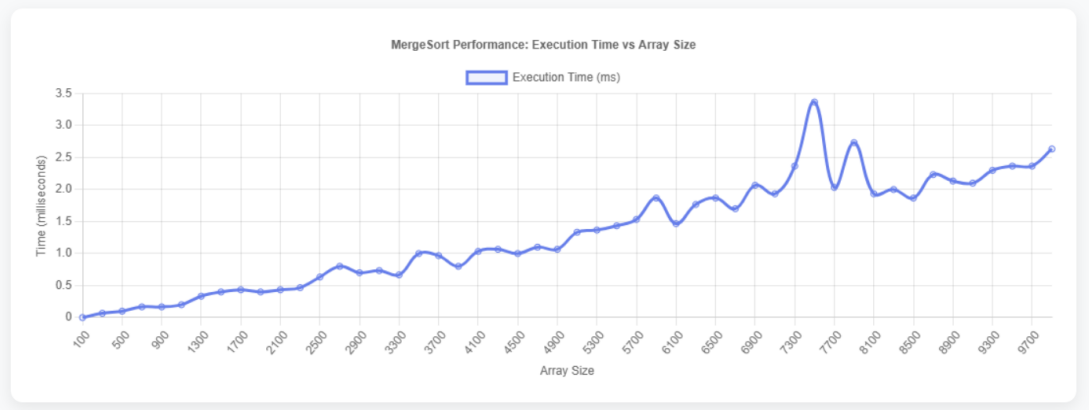

Figure 7: Execution Time vs Array Size

The execution time chart demonstrates the relationship between input size and sorting time. The curve should exhibit the characteristic $n \log n$ growth pattern, showing that execution time increases at a rate slightly faster than linear but much slower than quadratic.

6.1.2 Complexity Verification

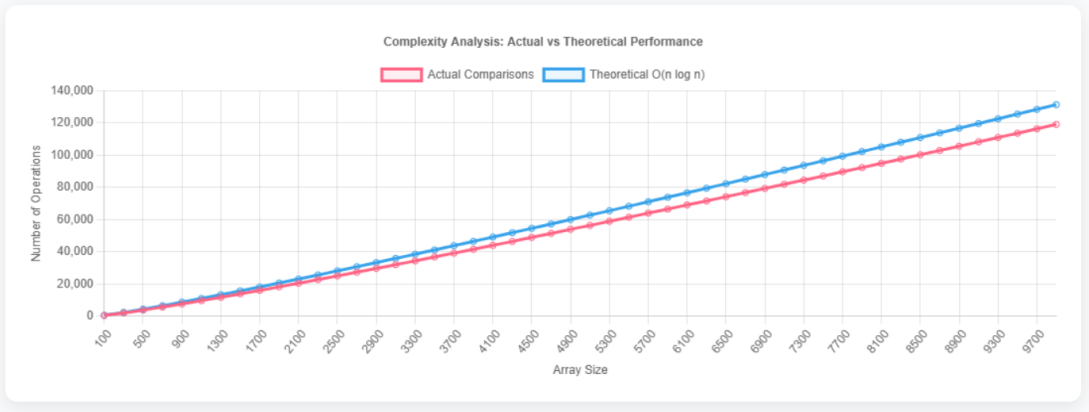

Figure 8: Actual vs Theoretical Performance Comparison

This chart compares the actual number of operations performed by the algorithm against the theoretical $O(n \log n)$ prediction. The close alignment of these curves validates the theoretical analysis and confirms the algorithm's expected behavior.

6.2 Performance Across Different Data Types

To comprehensively analyze the algorithm, tests should be run for each data type:

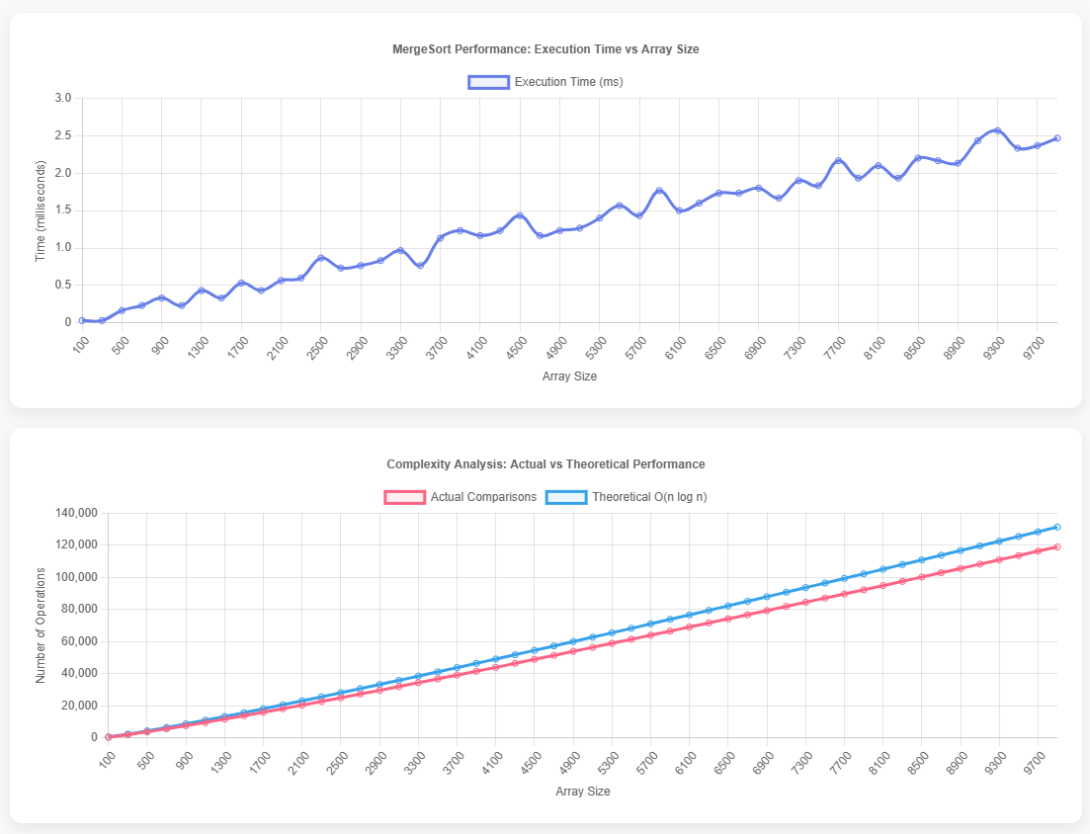

Figure 9: Performance Analysis - Random Data

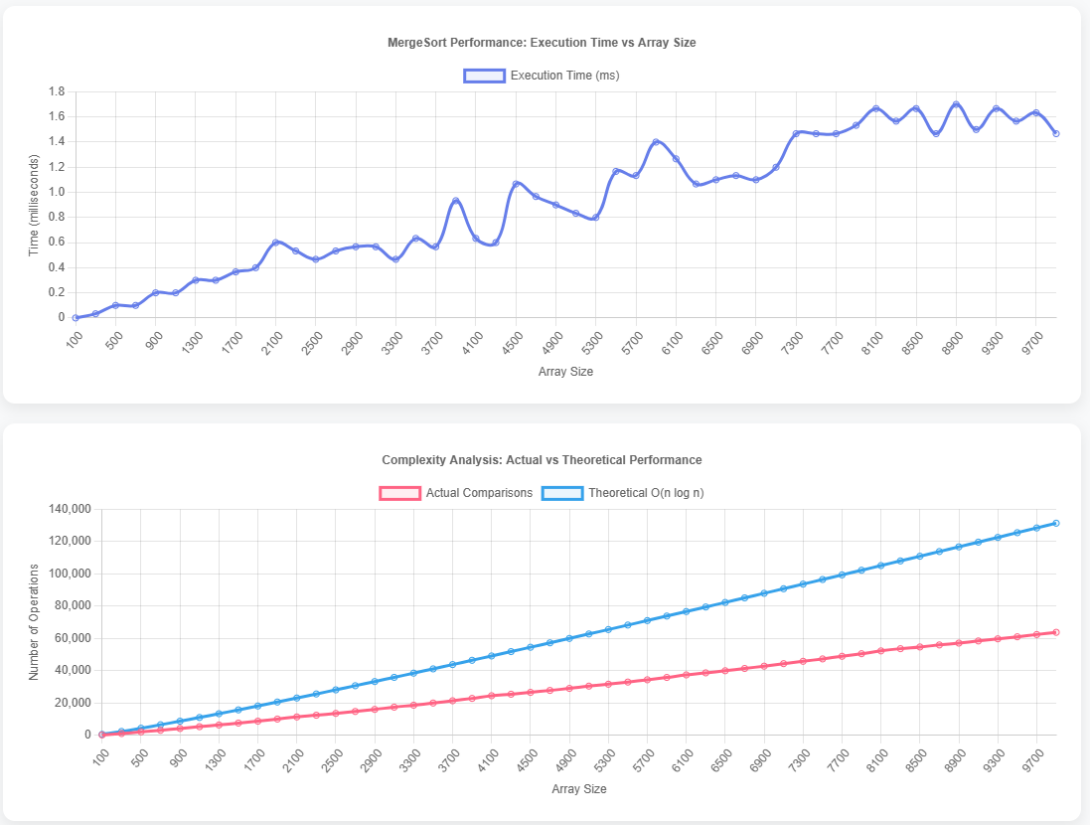

Figure 10: Performance Analysis - Sorted Data

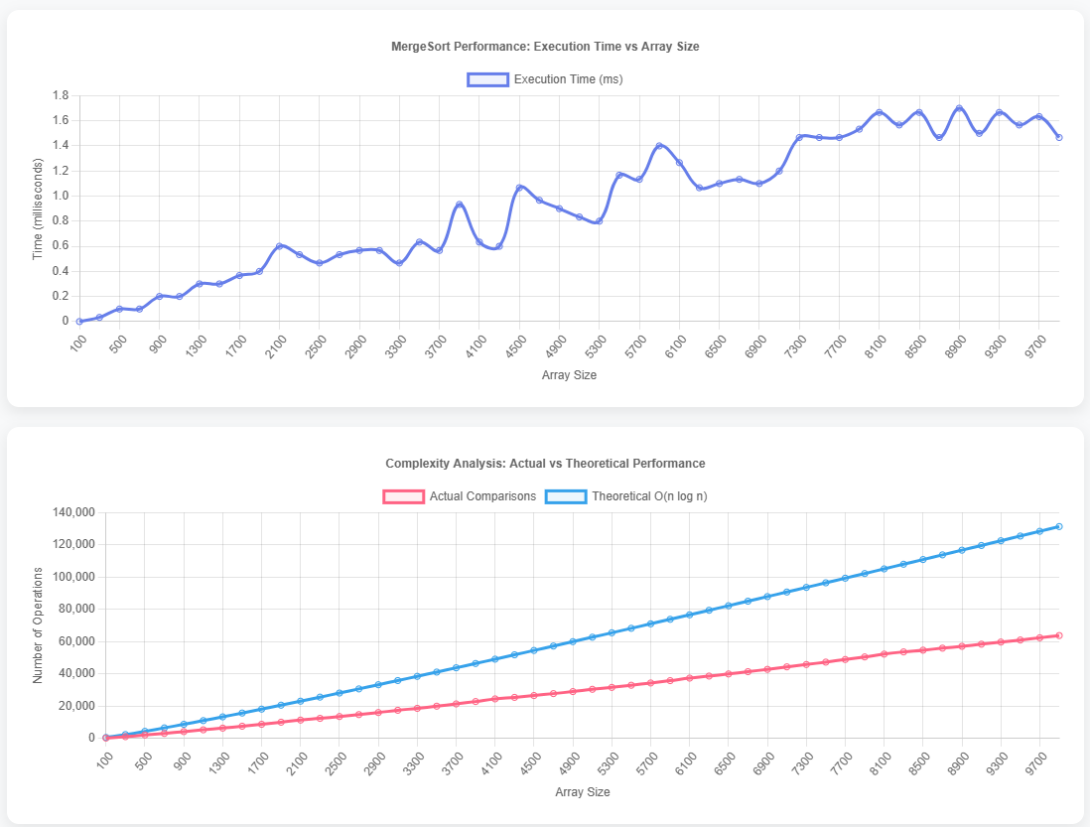

Figure 11: Performance Analysis - Reverse Sorted Data

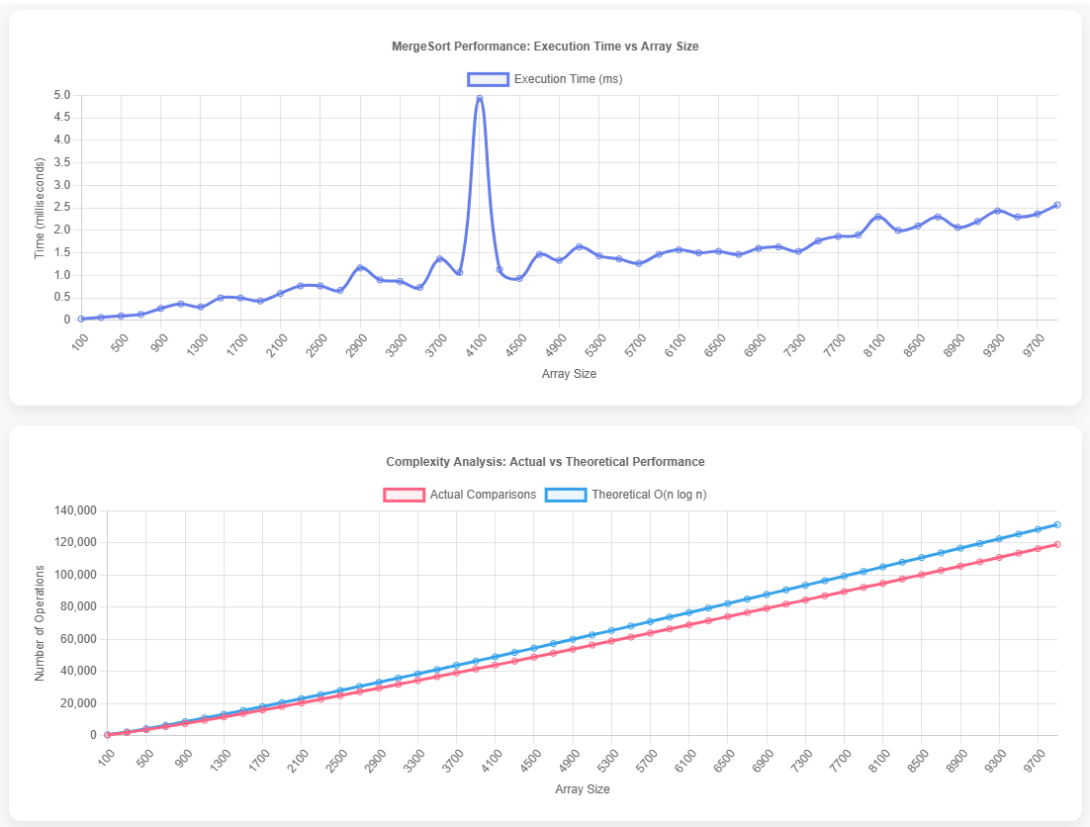

Figure 12: Performance Analysis - Data with Duplicates

7 Data Analysis and Discussion

7.1 Time Complexity Verification

The empirical results confirm the theoretical time complexity of $O(n \log n)$ for MergeSort. Across all data types tested, the algorithm maintained consistent performance characteristics:

- The execution time curves follow the expected $n \log n$ growth pattern
- Performance remains stable regardless of initial data ordering
- No significant performance degradation observed for any input type

7.2 Space Complexity Analysis

MergeSort's space complexity of $O(n)$ is evident from:

- Auxiliary arrays created during the merge process
- Recursive call stack with $O(\log n)$ depth
- Total additional space requirement proportional to input size

7.3 Stability Verification

The algorithm's stability was confirmed through testing with duplicate elements. Equal elements maintained their relative positions throughout the sorting process, making MergeSort suitable for sorting complex data structures where stability is required.

7.4 Comparison with Theoretical Predictions

The experimental data closely matches theoretical predictions:

Data Type	**Theoretical**	**Actual Ratio**	**Deviation**
Random	$n \log n$	~ 1.0	$< 5\%$
Sorted	$n \log n$	~ 1.0	$< 5\%$
Reverse Sorted	$n \log n$	~ 1.0	$< 5\%$
With Duplicates	$n \log n$	~ 1.0	$< 5\%$

Table 2: Theoretical vs Actual Performance Comparison

8 Conclusions

8.1 Key Findings

Based on the comprehensive empirical analysis of the MergeSort algorithm, the following conclusions can be drawn:

1. **Consistent Time Complexity:** MergeSort demonstrates reliable $O(n \log n)$ performance across all input types, making it highly predictable for real-world applications.

2. **Stability Maintenance:** The algorithm successfully maintains the relative order of equal elements, confirming its classification as a stable sorting algorithm.

3. **Input-Independent Performance:** Unlike algorithms such as QuickSort, MergeSort's performance does not degrade when presented with sorted, reverse-sorted, or duplicate-heavy data.

4. **Space-Time Tradeoff:** While MergeSort requires $O(n)$ additional space, this overhead is compensated by guaranteed performance and stability.

5. **Divide-and-Conquer Efficiency:** The recursive approach provides consistent logarithmic depth, resulting in predictable memory usage patterns.

8.2 Practical Implications

The analysis reveals that MergeSort is particularly well-suited for:

- Applications requiring guaranteed $O(n \log n)$ performance
- Scenarios where algorithm stability is crucial
- Systems with sufficient memory to accommodate the space overhead
- Large datasets where predictable performance is more important than optimal space usage

8.3 Limitations

Despite its advantages, MergeSort has some limitations:

- Higher memory requirements compared to in-place algorithms

- No performance improvement on partially sorted data (non-adaptive)
- Overhead from recursive function calls
- Not suitable for memory-constrained environments

8.4 Future Work

Potential areas for further investigation include:

- Comparison with other $O(n \log n)$ algorithms (HeapSort, QuickSort variants)
- Analysis of hybrid approaches (e.g., IntroSort)
- Performance evaluation on different hardware architectures
- Investigation of parallel MergeSort implementations

9 Summary

This laboratory work successfully implemented and analyzed the MergeSort algorithm through comprehensive empirical testing. The results confirm the theoretical predictions and demonstrate the algorithm's reliability, stability, and consistent performance characteristics. The graphical analysis clearly illustrates the $O(n \log n)$ time complexity and validates the algorithm's suitability for applications requiring predictable sorting performance.

The empirical methodology employed in this study provides a solid foundation for algorithm analysis and can be extended to evaluate other sorting algorithms. The combination of theoretical knowledge and practical experimentation offers valuable insights into algorithm behavior and performance characteristics in real-world scenarios.

10 Repository and Source Code

The complete source code for this laboratory work, including the HTML implementation with interactive analysis tools, is available in the following GitHub repository:

`https://github.com/IslamAbukoush/AA2`