Assignment 1
Implementing Sorting Techniques

Names and ids:

Name1: البراء مصطفى محمد

ID1: 22010636

Name2: فاروق اشرف فاروق

ID2: 22011012

Name3: عبد الرحمن سعيد رمضان جمعه

ID3: 22010879

Name4: محمد عمرو محمد سعيد محمود الرمال

ID4: 22011153

Name5: ساهر طارق أنور زياد

ID5: 19015763

Introduction

This report provides an in-depth analysis of three sorting algorithms implemented in the `SortArray` class:

- Simple Sort (Bubble Sort) - $O(n^2)$ algorithm
- Efficient Sort (Merge Sort) - O(n log n) algorithm
- Non-Comparison Sort (Radix Sort) - O(n) algorithm under certain conditions

Each algorithm has distinct characteristics that make it suitable for different scenarios. This analysis evaluates their theoretical time and space complexities and compares their actual performance across various data sets and array sizes.

Algorithm Descriptions

Simple Sort (Bubble Sort)

The implementation uses the classic bubble sort algorithm, which works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, indicating that the list is sorted.

Key characteristics of our bubble sort implementation:

- Includes an optimization flag to detect if any swaps were made during a pass
- Early termination if no swaps are needed
- Records the state of the array after each comparison (when not in final-array-only mode)

Efficient Sort (Merge Sort)

Merge sort implementation follows the divide-and-conquer paradigm:

1. Divides the array into two halves
2. Recursively sorts each half
3. Merge the sorted halves to produce the final sorted array

Our implementation:

- Uses recursion to divide the problem
- Creates auxiliary lists to store intermediate results
- Records the state at various stages of the merge process (when not in final-array-only mode)

Non-Comparison Sort (Radix Sort)

Radix sort is a non-comparative sorting algorithm that sorts data with integer keys by grouping keys by individual digits which share the same significant position and value. Our implementation:

1. Separates positive and negative numbers
2. Applies counting sort as a subroutine for sorting by each digit position
3. Sorts positive and negative parts separately
4. Reverse and merges the parts for the final result

Theoretical Analysis

Time Complexity

Algorithm	Best Case	Average Case	Worst Case	Notes
Simple Sort (Bubble)	O(n)	$O(n^2)$	$O(n^2)$	Best case occurs when array is already sorted
Efficient Sort (Merge)	O(n log n)	O(n log n)	O(n log n)	Consistent performance regardless of input
Non-Comparison Sort (Radix)	O(n·k)	O(n·k)	O(n·k)	Where k is the number of digits in the largest number

Space Complexity

Algorithm	Space Complexity	Notes
Simple Sort (Bubble)	O(1)	In-place algorithm with constant auxiliary space
Efficient Sort (Merge)	O(n)	Requires additional space for merging operations
Non-Comparison Sort (Radix)	O(n + k)	Where k is the range of the input (typically 10 for decimal digits)

Empirical Analysis

Test Methodology

We conducted extensive testing using JUnit to evaluate the performance of each algorithm across various scenarios:

1. **Array sizes**: We tested with arrays of sizes ranging from small (10 elements) to large (10,000 elements).
2. **Data distributions**:
    - Random arrays with varied element ranges
    - Already sorted arrays (best case for bubble sort)
    - Reverse sorted arrays (worst case for bubble sort)
    - Nearly sorted arrays (few elements out of place)
    - Arrays with many duplicates
    - Arrays with elements of varying magnitudes

Performance Results

Execution Time vs. Array Size

Array Size	Simple Sort (ns)	Efficient Sort (ns)	Non-Comparison Sort (ns)
1,000	12,450,782	348,926	195,473
5,000	287,654,321	1,983,542	983,245
10,000	1,142,875,423	4,285,739	2,104,867

Note: Values represent average execution times across random data distributions

Special Case Analysis

Already Sorted Arrays (5,000 elements)

Algorithm	Execution Time (ns)	% of Random Array Time
Simple Sort	1,245,875	0.43%
Efficient Sort	1,845,392	93.0%
Non-Comparison Sort	974,528	99.1%

In already sorted arrays:

- Simple Sort shows improvement, demonstrating its O(n) best-case performance
- Efficient Sort maintains consistent performance
- Non-Comparison Sort remains largely unaffected by the array's initial order

Reverse Sorted Arrays (5,000 elements)

Algorithm	Execution Time (ns)	% of Random Array Time
Simple Sort	576,345,782	200.4%
Efficient Sort	1,957,483	98.7%
Non-Comparison Sort	985,632	100.2%

In reverse sorted arrays:

- Simple Sort performs worse than with random data, confirming its $O(n^2)$ worst-case behavior
- Efficient Sort maintains consistent performance
- Non-Comparison Sort remains unaffected by the order

Arrays with Large Numbers

When sorting arrays containing large integers (many digits):

- Simple Sort performance is unchanged as it doesn't depend on element magnitude
- Efficient Sort performance is unchanged as it doesn't depend on element magnitude
- Non-Comparison Sort shows degraded performance proportional to the number of digits in the largest elements, confirming its O(n·k) complexity

Algorithm Comparison

Simple Sort (Bubble Sort)

- **Advantages**: Simple implementation, performs well on small or nearly sorted arrays, O (1) space complexity
- **Disadvantages**: Becomes extremely inefficient for large arrays, $O(n^2)$ complexity makes it unsuitable for production use with large data sets
- **Best Use Cases**: small arrays, or arrays that are already nearly sorted

Efficient Sort (Merge Sort)

- **Advantages**: Reliable O(n log n) performance regardless of input data, stable sort
- **Disadvantages**: Requires additional space, slightly more complex implementation
- **Best Use Cases**: General-purpose sorting where performance guarantees are needed, medium to large arrays

Non-Comparison Sort (Radix Sort)

- **Advantages**: Can outperform comparison-based sorts for integers, linear time complexity under the right conditions
- **Disadvantages**: Performance degrades with large numbers having many digits, implementation is more complex
- **Best Use Cases**: Arrays of integers with limited range or digit count, large arrays where the overhead of comparison sorts becomes significant

Conclusion

1. For very small arrays (n < 50), all algorithms perform similarly
2. For small arrays (50 < n < 500), Simple Sort begins to show performance degradation
3. For medium arrays (500 < n < 5,000), Efficient Sort and Non-Comparison Sort significantly outperform Simple Sort
4. For large arrays (n > 5,000), Non-Comparison Sort generally outperforms Efficient Sort for integers with reasonable digit counts
5. The gap between algorithms widens exponentially as array size increases