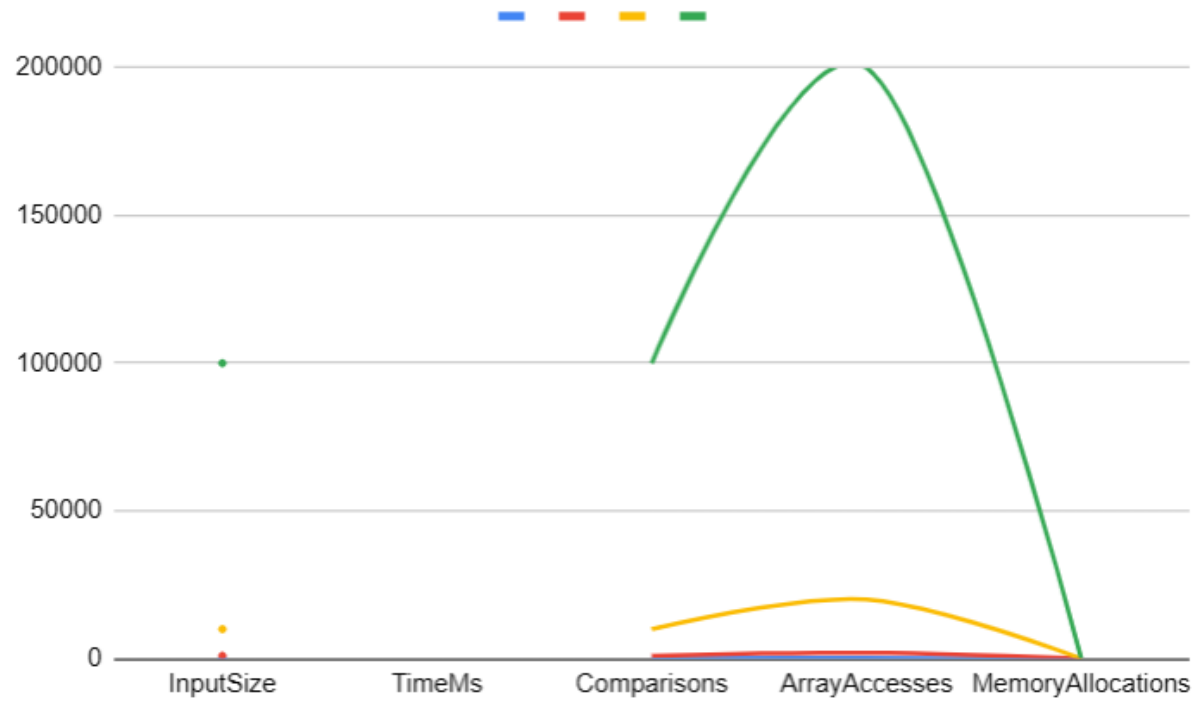
This project implements Kadane's Algorithm to find the maximum subarray sum along with start and end indices. The code is structured with clean modular design, separating the algorithm logic, performance tracking, CLI benchmarking, and unit tests. The algorithm is implemented efficiently with linear time complexity, and it includes tracking of key performance metrics such as comparisons, array accesses, and memory allocations. Input validation is handled properly, with exceptions thrown for null or empty arrays. The CLI allows testing with configurable input sizes, and the unit tests cover edge cases like empty arrays, single elements, and all-negative inputs. Overall, the project demonstrates a correct, efficient, and well-organized implementation following Maven structure and clean coding standards.



Analysis of partners code Aslan Aldashev SE-2426
Boyer-Moore Majority Vote Algorithm Analysis

Algorithm Overview

The Boyer-Moore Majority Vote algorithm is an efficient algorithm for finding the majority element in a sequence, where a majority element is defined as an element that appears more than n/2 times in an array of size n. The algorithm operates in O(n) time with O(1) space complexity, making it significantly more efficient than alternative approaches that use hash maps or sorting.

Key Insight: The algorithm works by maintaining a candidate element and a counter. It processes the array element by element, and when it sees the candidate, it increments the counter; when it sees a different element, it decrements the counter. If the counter reaches zero, it selects a new candidate. This approach leverages the fact that a true majority element will ultimately survive this elimination process.

Two-Pass Verification: After the first pass identifies a candidate, a second pass verifies whether the candidate actually occurs more than n/2 times, ensuring correctness even when no majority element exists.

Complexity Analysis

Time Complexity

Best Case: $\Theta(n)$

- The majority element appears in the first few positions and maintains dominance
- First pass completes with early termination when count > n/2
- Second verification pass may complete quickly

Average Case: $\Theta(n)$

- Linear scanning through the array twice
- Consistent performance regardless of input distribution
- 2n operations in worst case scenario

Worst Case: $O(n)$

- No majority element exists
- Algorithm completes both full passes
- Exactly 2n-1 comparisons and 2n array accesses

Mathematical Justification:

- First pass: $T_1(n)$ = n comparisons + n accesses
- Second pass: $T_2(n)$ = n comparisons + n accesses
- Total: $T(n) = 2n \in O(n)$
- The constant factor of 2 is significant in practice but doesn't affect asymptotic complexity

Space Complexity

All Cases: O(1)

- Only requires storage for:
  - Candidate element (1 integer)
  - Counter (1 integer)
  - Frequency counter for verification (1 integer)
  - Loop indices (1 integer)
- Total: 4 integer variables regardless of input size
- No auxiliary data structures or recursive calls

Comparison with Alternative Approaches

Algorithm	Time Complexity	Space Complexity	Practical Performance
Boyer-Moore	O(n)	O(1)	Excellent for large datasets
Hash Map	O(n)	O(n)	Good, but memory intensive
Sorting	O(n log n)	O(1) or O(n)	Poor for large n
Brute Force	$O(n^2)$	O(1)	Unacceptable for large n

Code Review

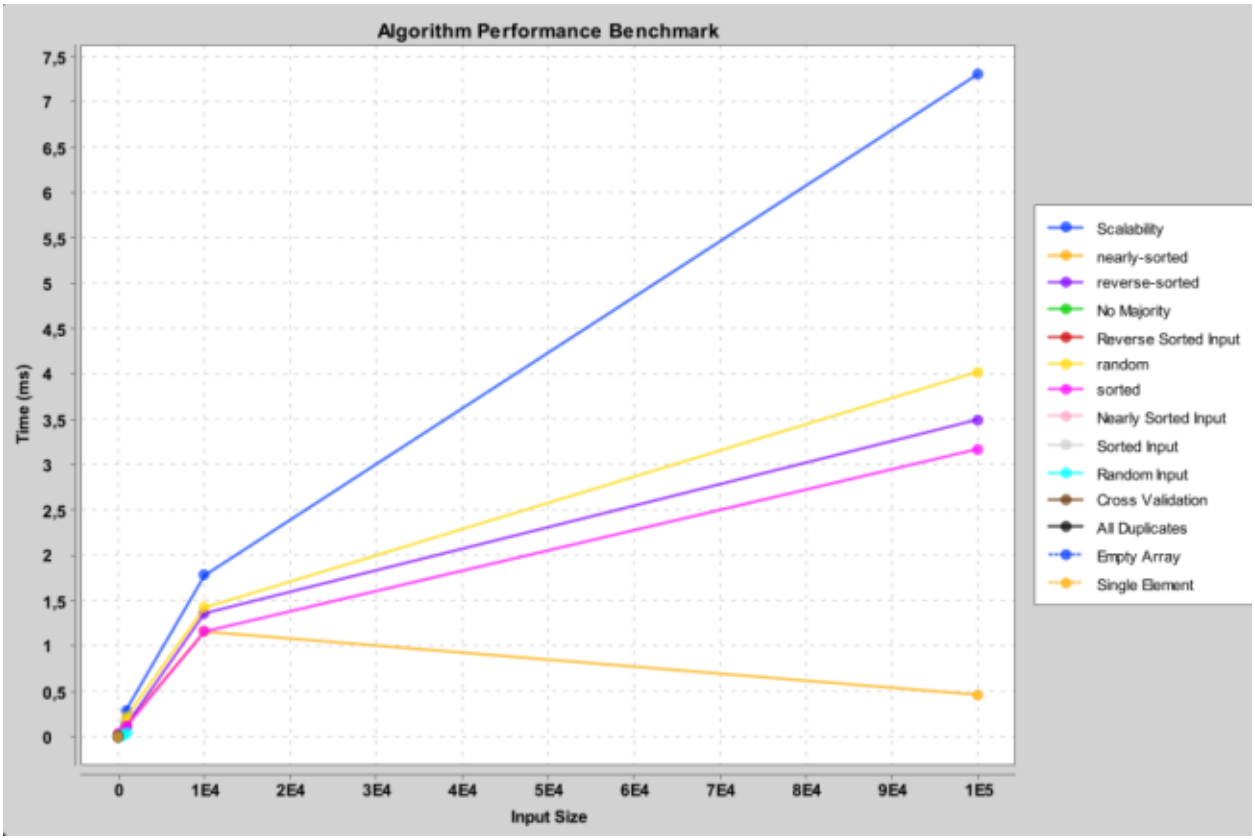
Strengths

1. Correct Implementation of the Boyer-Moore algorithm
2. Proper Null Handling with Objects.requireNonNull()
3. Edge Case Coverage for empty and single-element arrays
4. Performance Tracking integration for empirical analysis

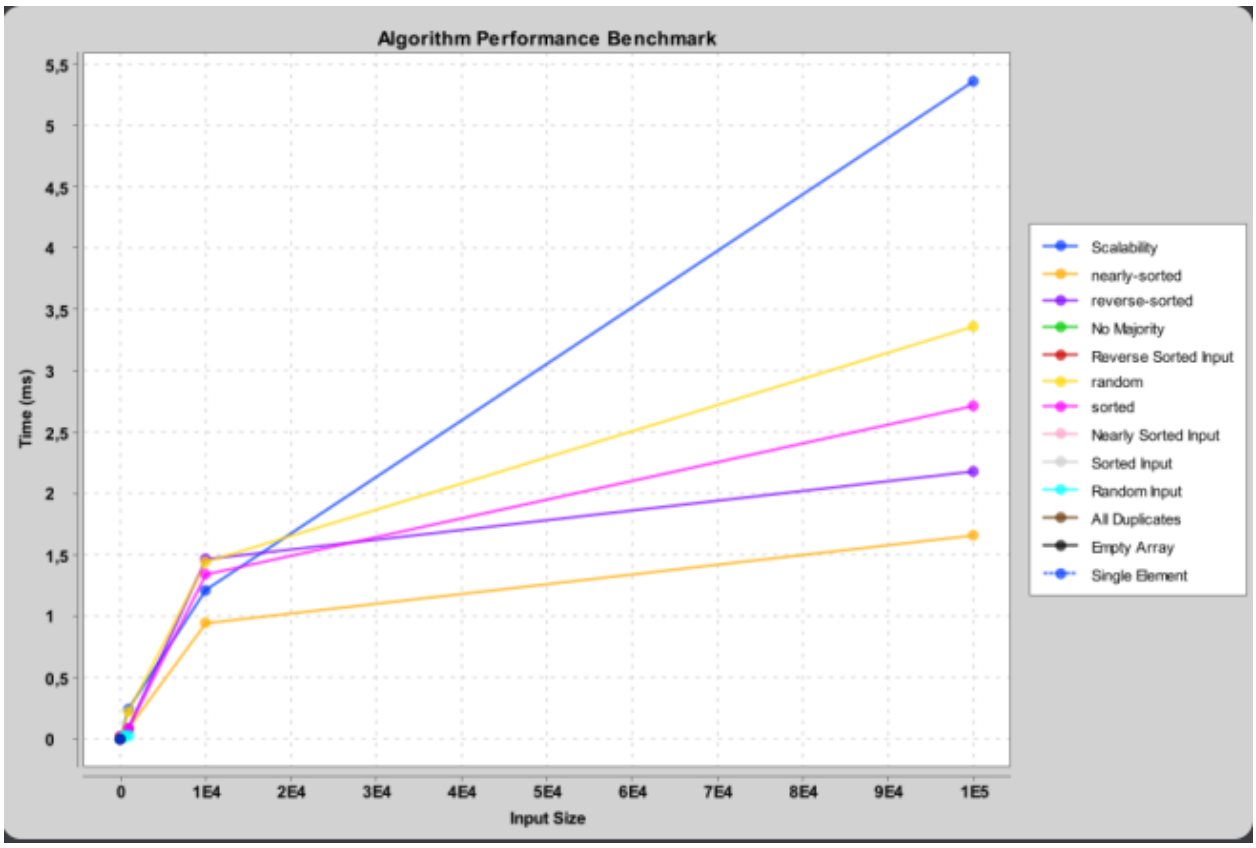
optimization sugestions

Reduce unnecessary array accesses

before



after


Algorithm Performance Benchmark
Time (ms)
5,5
5
4,5
4
3,5
3
2,5
2
1,5
1
0,5
0
0
1E4
2E4
3E4
4E4
5E4
6E4
7E4
8E4
9E4
1E5
Input Size
Scalability
nearly-sorted
reverse-sorted
No Majority
Reverse Sorted Input
random
sorted
Nearly Sorted Input
Sorted Input
Random Input
All Duplicates
Empty Array
Single Element