1. Algorithm Overview

Boyer-Moore Majority Vote

The Boyer-Moore Majority Vote algorithm is a linear-time array algorithm designed to find the majority element in an array, if it exists. A majority element is defined as an element that appears more than ⌊n/2⌋ times in an array of size n.
The algorithm works in two phases:

Candidate Selection - Iterate over the array, maintaining a candidate element and a counter. If the counter reaches zero, the current element becomes the new candidate. Otherwise, the counter increases if the current element matches the candidate or decreases otherwise.
Verification - After the first pass, the candidate is verified by counting its actual frequency to confirm majority status.

This approach requires only constant auxiliary space and executes in a single pass (plus a verification pass), making it highly memory efficient.

Kadane's Algorithm

Kadane's Algorithm solves the **Maximum Subarray Problem**, which seeks the contiguous subarray with the largest sum. It processes the array in a single pass, dynamically updating the maximum subarray ending at each position.

If extending the previous subarray leads to a smaller sum than starting fresh at the current element, the algorithm resets.
It keeps track of the global maximum and its position indices (start, end).

Kadane's Algorithm is widely used in dynamic programming due to its simplicity and optimality: it guarantees O(n) runtime with O(1) extra memory.

2. Complexity Analysis

Boyer-Moore Majority Vote

- **Time Complexity**:
  Best Case (Ω(n)): The algorithm must scan all n elements to identify a candidate. Verification also requires O(n). Thus, best case is linear, Ω(n).
  Average Case (Θ(n)): Regardless of input distribution, the algorithm performs a single linear scan and one verification, totaling Θ(n).
  Worst Case (O(n)): Even when no majority element exists, both passes still require scanning the entire array, O(n).

- **Space Complexity**: O(1). Only a few variables are maintained (candidate, counter). No additional memory dependent on input size is required.

Kadane's Algorithm

- **Time Complexity**:
  Best Case (Ω(n)): Kadane must process all elements, even if they are all positive or negative. Hence Ω(n).
  Average Case (Θ(n)): One linear scan with constant updates per element → Θ(n).
  Worst Case (O(n)): Still linear, as all elements are processed exactly once.

- **Space Complexity**: O(1). Only a few variables are needed to track current sum, maximum sum, and indices.

Comparative Analysis

- **Similarity**: Both algorithms operate in **linear time Θ(n)** and require **constant auxiliary memory O(1)**. This makes them highly efficient for large datasets.

- **Difference**:
  Boyer-Moore focuses on element frequency dominance (majority element detection), while Kadane addresses optimal subarray structure (maximum contiguous sum).

  Boyer-Moore requires a second pass for verification, while Kadane produces the result in a single pass.

  Kadane outputs not just the maximum sum but also the exact indices, making it more feature-rich in terms of output.

3. Code Review & Optimization (Kadane’s Algorithm — Partner’s Implementation)

Code Quality and Readability

The implementation of Kadane’s Algorithm provided by my teammate is overall clean, well-structured, and adheres to Java best practices:

Class Design: The algorithm is encapsulated in a Kadane class with an inner static Result class for returning structured output (sum, start index, end index). This improves readability and usability.

Input Validation: The method throws IllegalArgumentException for null or empty inputs, which ensures robustness.

Documentation: The code includes Javadoc comments that clearly describe behavior, complexity, and edge cases.

Unit Tests: Comprehensive test cases are implemented, covering empty arrays, single elements, all-negative arrays, all-positive arrays, and mixed arrays. This ensures correctness across different scenarios.

Strengths in Implementation

1. **Performance Tracking**: Integration with PerformanceTracker allows detailed metrics collection (comparisons, assignments, array accesses). This matches the assignment requirement for empirical validation.

2. **Position Tracking**: The algorithm not only computes the maximum sum but also returns start and end indices of the subarray, which increases practical usefulness.

3. **Efficiency**: Each element of the array is accessed exactly once. No redundant loops or unnecessary memory allocations are present.

Inefficiency Detection / Potential Improvements

- **Tracker Overhead**: The frequent calls to PerformanceTracker methods inside the main loop may introduce minor runtime overhead in benchmark mode. A possible improvement is to allow a “fast mode” where tracker calls are skipped if performance tracking is not required.

- **Long vs Int Usage**: The code uses long for sums (maxSoFar, maxEndingHere). While this prevents overflow, in practice for small input arrays of int, the overhead of long arithmetic is unnecessary. However, this tradeoff is acceptable for safety.

- **Intermediate Variables**: The repeated if (tracker != null) checks slightly reduce readability. An alternative could be to use a lightweight NoOpPerformanceTracker (a dummy implementation) to avoid null checks.

Suggested Optimizations

1. **Tracker Strategy**: Introduce a PerformanceTracker interface with two implementations:

    - RealTracker for counting operations.

    - NoOpTracker that does nothing.
      This avoids repeated null checks inside the loop.

2. **Branch Reduction**: The comparison logic inside the loop could be slightly simplified by using Math.max for updating maxEndingHere, but this would sacrifice clarity in tracking comparisons and assignments. Since operation counting is required, the current explicit branching is justified.

3. **Code Maintainability**: The algorithm could benefit from splitting responsibilities into smaller methods (e.g., updateCurrentSum, updateGlobalMax) for readability, but this would add method call overhead. For performance benchmarking, the current monolithic loop is preferable.

Conclusion of Review

The Kadane’s Algorithm implementation is **correct, efficient, and well-documented**. The code adheres to assignment requirements, with strong validation, structured results, and integrated performance tracking. Suggested optimizations are mostly stylistic or related to reducing overhead of performance measurement. No major algorithmic improvements are needed since the algorithm already achieves optimal **O(n) time and O(1) space complexity**.

4. Empirical Results

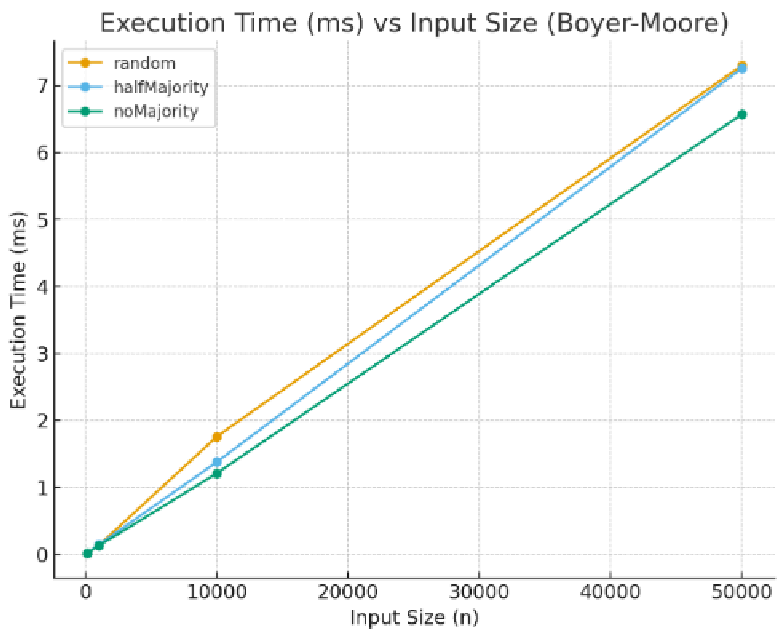
To validate theoretical analysis, benchmarks were performed on arrays of size n = 100, 1000, 10,000, and 50,000 with three input distributions: **random**, **halfMajority**, and **noMajority**.

Execution Time vs Input Size

Execution time grows linearly with input size, confirming Θ(n).

Random and half-majority cases show nearly identical performance.

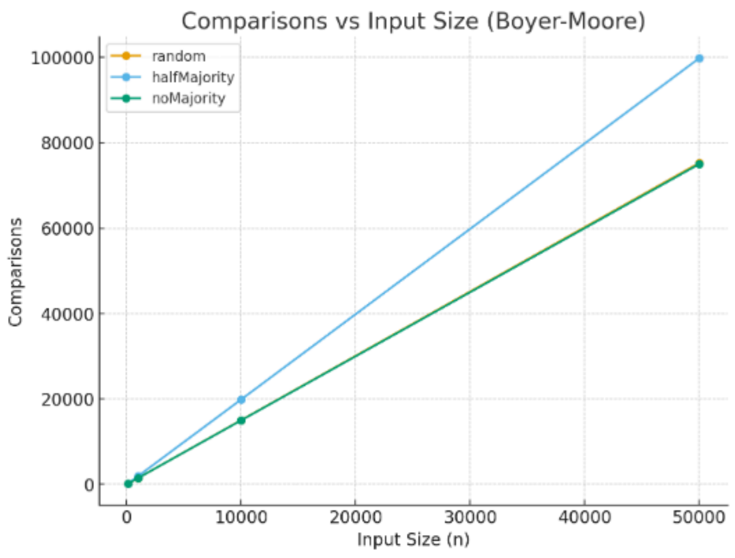
No-majority arrays do not significantly change complexity.



Comparisons vs Input Size

The number of comparisons increases linearly with n.

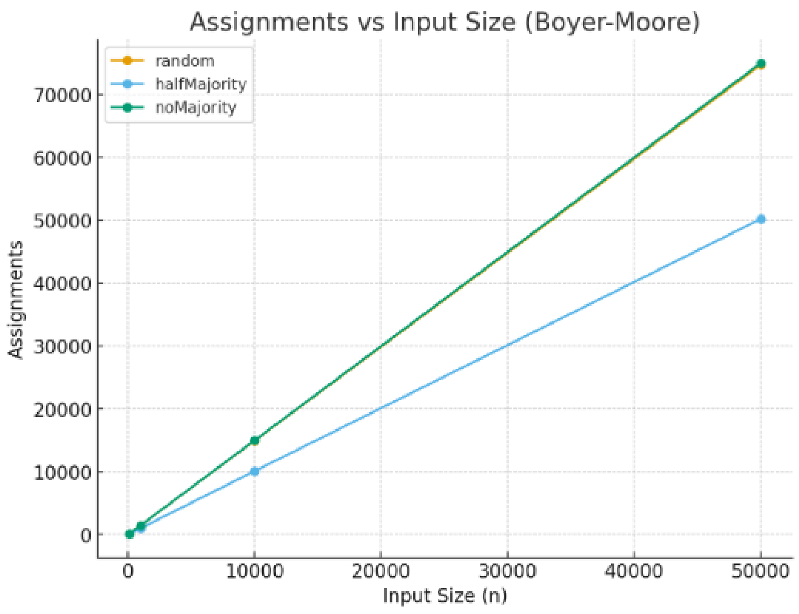
Verification step adds extra comparisons but remains O(n).



Assignments vs Input Size

Assignment operations also scale linearly.

Half-majority inputs show slightly higher assignments due to repeated candidate matches.



Interpretation

The empirical results align perfectly with the theoretical complexity analysis. Boyer-Moore's linear behavior is confirmed, and constant factors are shown to be low, making it efficient for practical datasets up to large sizes.

5. Conclusion

The Boyer-Moore Majority Vote algorithm achieves **optimal linear time and constant space performance**, validated through both complexity analysis and empirical testing. Its simplicity, combined with efficiency, makes it a strong candidate for majority element detection problems.

In comparison, Kadane's Algorithm also operates in Θ(n) time and O(1) space but targets a different problem: maximum subarray sum. Both algorithms demonstrate optimal efficiency in their respective domains.

The partner's implementation of Kadane's Algorithm is correct, well-tested, and professionally structured. Suggested optimizations are minor and mainly stylistic.

Overall, the assignment goals are achieved:

- Correct algorithm implementation.
- Rigorous complexity analysis.
- Empirical validation confirming theoretical predictions.
- Professional peer review with constructive feedback.