Peer Analysis Report

Partner's algorithm: Boyer-Moore Majority Vote Algorithm
Language: Java
Partner: Mansur Serikbai

---

1. Algorithm Overview

1.1. Brief Description

The Boyer-Moore Majority Vote Algorithm is designed to find the majority element in an array, i.e., the element that appears more than half the times in the array. It operates in linear time and constant space, making it an optimal solution for this problem.

The algorithm works by making two passes over the array:

- **First pass**: It identifies a potential candidate for the majority element by using a counter variable to track the count of the current candidate. If the count reaches zero, the current element is chosen as the new candidate.
- **Second pass**: It verifies whether the candidate occurs more than half the times in the array.

The key metrics tracked by the algorithm are:

- **count**: the number of occurrences of the current candidate.
- **candidate**: the current element considered as the majority element.

This approach ensures the algorithm works efficiently in O(n) time with O(1) space complexity.

1.2. Theoretical Justification

The Boyer-Moore Majority Vote Algorithm is based on a simple inductive step. If the count of the current candidate reaches zero, the algorithm selects a new candidate. By the end of the first pass, the candidate with the highest count is retained.

- **Recurrence Relation**:
  - For any index `i`, if the current element matches the candidate, `count` is incremented. If it doesn't match, `count` is decremented.
  - If `count` reaches zero, a new candidate is selected.
- **Global Maximum Sum**:
  - The maximum sum is updated with the maximum of the current `count` and the global maximum value.

Since the algorithm doesn't rely on recursion and only uses a simple iterative approach, it achieves the highest possible efficiency with time complexity $\Theta(n)$ and space complexity O(1).

---

2. Complexity Analysis

2.1. Time Complexity

The time complexity is determined by two passes over the input array. In both passes, the algorithm examines each element exactly once.

Case	Θ (exact boundary)	O (upper)	Ω (lower)	Mathematical Justification
Best	Θ(n)	O(n)	Ω(n)	The work done is linear as every element is examined once.
Worst	Θ(n)	O(n)	Ω(n)	The worst-case scenario still operates in linear time.
Average	Θ(n)	O(n)	Ω(n)	The average complexity matches the best and worst cases, since no extra work depends on the data.

The total operating time T(n) can be expressed mathematically as:

$$T(n) = c_{\text{init}} + \sum_{i=1}^{n} c_{\text{loop}} + c_{\text{final}}$$

Where:

- C$\boldsymbol{init}$ is the time for initialization.
- C$\boldsymbol{loop}$ is the constant time for operations within the loop.
- C$\boldsymbol{final}$ is the time for final checks and returning the result.

Thus, the overall time complexity is Θ(n).

2.2. Space Complexity

The space complexity of the Boyer-Moore Majority Vote Algorithm is O(1). The algorithm only uses a fixed amount of space (for `count` and `candidate`), which does not depend on the input size.

Analysis	Auxiliary Space	Justification
Space Complexity	O(1)	The algorithm uses a fixed number of variables (e.g., `count`, `candidate`), so its memory requirements do not depend on the input size.
In-place Optimization	In-place	The algorithm operates in-place and does not require additional memory allocation.

Since the space complexity is constant, there are no further optimizations possible regarding space.

3. Code Review

3.1. Inefficiency Detection

While the Boyer-Moore Majority Vote Algorithm has optimal time and space complexity, the code implementation can be further optimized:

- **Excessive Comparisons**: The second pass is required to confirm the majority, but if there are optimizations available that could combine the two passes (e.g., by using a hash map or other auxiliary data structures), it might be beneficial.
- **Debugging Output**: A common inefficiency in many algorithms is the presence of unnecessary debugging output. While debugging output such as `System.out.println` is helpful during development, it can severely slow down the execution time if left in the code during runtime. Removing these lines can drastically improve performance.

3.2. Optimization Suggestions

Time Complexity Improvements:

- **Remove Debugging Output**: Eliminating any `System.out.println` or `System.out.printf` calls inside the loop will improve performance, particularly for smaller datasets where I/O operations dominate over computational steps.
- **Refactor to Canonical Form**: The Boyer-Moore Majority Vote Algorithm can be made even more efficient by using the canonical form, which does not require additional variables for handling special cases like all non-positive numbers. Simplifying the code reduces overhead and improves maintainability.

Space Complexity Improvements:

- **In-place Optimization**: The space complexity is already optimal (O(1)), so no further space optimizations are required.

3.3. Code Quality

- **Readability/Style**: The code is generally readable, but the inclusion of extra variables like `hasPositive` and `maxElement` adds unnecessary complexity to the logic. A cleaner, more standard implementation would make the code easier to understand for third-party developers.
- **Supportability**: The non-standard approach with additional checks for non-positive numbers makes the algorithm harder to support in the long term. Using the canonical Boyer-Moore approach would improve maintainability and ease of support.

---

4. Empirical Results

4.1. Performance Plots and Validation

Theoretical prediction: The time T(n) should grow linearly with the input size n ($T(n) \propto n$)

n	Random (Baseline)	Random (Optimized)	(T(n)/n) (Baseline, ns/element)
100	7,795,400 ns	4,106,500 ns	77,954
1000	3,176,900 ns	3,257,200 ns	3,177

n	Random (Baseline)	Random (Optimized)	(T(n)/n) (Baseline, ns/element)
10000	3,365,700 ns	3,391,200 ns	337
100000	7,114,800 ns	5,332,800 ns	71

The time complexity grows linearly as expected. The ratio T(n)/n (time per element) decreases as n increases, confirming that the algorithm behaves as expected for larger datasets.

4.2. Optimization Impact

Removing I/O operations from the loop shows a noticeable improvement in execution time.

n	Baseline time (ns)	Optimized time (ns)	Absolute Reduction	The effect
100	7,795,400	4,106,500	3,688,900	47.3% Decrease
10000	3,365,700	3,391,200	Insignificant	~0%
100000	7,114,800	5,332,800	1,782,000	25.0% Decrease

The optimizations significantly improve the performance, especially for smaller values of n.

5. Conclusion

5.1. Summary of Findings

The Boyer-Moore Majority Vote Algorithm is asymptotically optimal for the majority element problem:

- **Time Complexity**: $\Theta(n)$
- **Space Complexity**: $O(1)$

However, the practical performance and code quality could be improved:

- **Critical Factor (Performance)**: Excessive use of I/O operations within the loop introduces significant overhead.
- **Quality Factor (Maintainability)**: Use of non-standard logic for handling edge cases increases code complexity.

5.2. Optimization Recommendations

To achieve the best balance between theoretical and practical performance, the following steps are recommended:

- **Remove all I/O** operations (e.g., `System.out.println` calls) from the `findMajority` method to reduce execution time.
- **Refactor to the canonical form** of the Boyer-Moore algorithm to simplify the code and improve maintainability.