Peer Code Review Analysis Report

Course: Design and Analysis of Algorithms

Assignment: Assignment 2 - Algorithmic Analysis and Peer Review

Algorithm Analyzed: Boyer-Moore Majority Vote (single-pass majority element detection)

Implemented by: Student A (Nurasyl Asan)

Reviewed by: Student B (Saqyp Dias)

Date: October 5, 2025

Repository: https://github.com/Dericeda/designAndAnalysisOfAlgorithms2

---

1. Algorithm Overview

1.1 Problem Statement

The objective is to identify the majority element within a one-dimensional array of elements. A majority element is defined as an element that appears more than n/2 times, where n is the total number of elements in the array. The algorithm should efficiently find this element if it exists.

1.2 Algorithm Description

Student A has implemented the Boyer-Moore Majority Vote algorithm. This algorithm is highly efficient and operates in two distinct passes:

1. **Find a Candidate:** The first pass iterates through the array to find a single candidate for the majority element. It maintains a `candidate` element and a `count`. The count is incremented if the current element matches the candidate and decremented otherwise. If the count reaches zero, the current element becomes the new candidate.

2. **Verify the Candidate:** The second pass verifies if the identified candidate is truly the majority element by counting its total occurrences in the array and checking if this count is greater than n/2.

The implementation correctly encapsulates this logic within the `BoyerMooreMajorityVote.java` class, which uses a `PerformanceTracker` to measure its efficiency.

---

2. Complexity Analysis

2.1 Time Complexity

Best, Average, and Worst Case: Θ(n)

The algorithm's time complexity is linear, as it consists of two sequential passes through the array. The performance is not dependent on the distribution or ordering of the data.

- **Mathematical Justification:**
  - The first pass (finding a candidate) iterates through the array once: $T_1(n) = O(n)$.
  - The second pass (verifying the candidate) also iterates through the array once: $T_2(n) = O(n)$.
  - The total time complexity is $T(n) = T_1(n) + T_2(n) = O(n) + O(n) = \mathbf{O(n)}$.

2.2 Space Complexity

Auxiliary Space: Θ(1)

The algorithm is extremely memory-efficient, utilizing only a constant amount of extra space.

- It requires a few variables to store the `candidate`, `count`, and the `PerformanceTracker` object. This space requirement does not scale with the input size `n`, making the algorithm optimal for large datasets.

2.3 Comparison with Alternative Approaches

Algorithm	Time Complexity	Space Complexity	Notes
Brute Force	$O(n^2)$	O(1)	Two nested loops to count each element.
Hash Map	O(n)	O(n)	Stores element counts in a hash map.
Sorting	O(n log n)	O(1) or O(n)	Sorts the array and picks the middle element.
Boyer-Moore (Student A)	**O(n)**	**O(1)**	**Optimal time-space tradeoff.**

Экспортировать в Таблицы

Student A's implementation achieves the best possible time and space complexity for solving the majority element problem.

---

3. Code Review and Optimization Analysis

Strong Points:

1. **Correct Implementation:** The algorithm correctly identifies the majority element and handles cases where no majority element exists by returning an `Optional`.
2. **Comprehensive Testing:** The project includes a thorough test suite (`BoyerMooreMajorityVoteTest.java`) covering various scenarios, including empty arrays, null arrays, and different distributions of the majority element.
3. **Performance Metrics:** A `PerformanceTracker` class is well-integrated to measure key metrics like execution time, comparisons, and array accesses, which is excellent for empirical analysis.
4. **Code Clarity:** The code is well-structured and uses clear variable names (e.g., `candidate`, `count`), making the logic easy to follow. Javadoc comments are present and explain the functionality.
5. **Benchmarking:** A `BenchmarkRunner` is provided to systematically test the algorithm's performance across different input sizes and types, and to export the results to a CSV file.

Identified Inefficiencies & Potential Optimizations:

- **Issue #1: Redundant `candidate != null` Check**
  - **Location:** `BoyerMooreMajorityVote.java`, line 46
  - **Current Code:** `if (candidate != null && num == candidate)`
  - **Problem:** In the verification pass, the `candidate` will always have a value assigned from the first pass (unless the input array is empty, which is handled at the beginning). The `candidate != null` check is therefore redundant in every iteration of the second loop.
  - **Proposed Optimization:** Remove the `candidate != null` check from the second loop to slightly reduce the number of comparisons, although the impact on overall performance will be minimal.

---

4. Empirical Validation

4.1 Experimental Setup

- **Test Environment:**
  - CPU: Intel Core i3-10110U @ 2.10GHz
  - RAM: 8 GB DDR4
  - Java Version: JDK 17
- **Test Methodology:**
  - Input sizes: `n` = {1000, 10000, 100000, 500000}
  - 5 benchmark runs per input size.
  - Four different data distributions were tested: "Majority at Start", "Majority at End", "Random with Majority", and "Random (No Majority)".

4.2 Performance Results

The performance results were exported to `boyer_moore_results.csv` and visualized. The empirical data strongly supports the theoretical O(n) complexity.

Time Complexity Verification:

Input Size (n)	Avg Time (ms) (Random w/ Majority)
1,000	0.127
10,000	0.192
100,000	1.810
500,000	2.066

Экспортировать в Таблицы

As seen in the `time_vs_size.png` plot, the execution time grows linearly with the input size `n`, which is consistent with O(n) behavior.

Comparison Count Analysis:

Input Size (n)	Comparisons (Random w/ Majority)	Comparisons/n
1,000	1,977	~2.0
10,000	19,926	~2.0
100,000	199,241	~2.0
500,000	999,403	~2.0

Экспортировать в Таблицы

The number of comparisons is approximately `2n` (`n` for the first pass and `n` for the second). The data in the CSV file confirms this, with the ratio of comparisons to `n` being consistently close to 2.0. The `comparisons_vs_size.png` plot further illustrates this linear relationship.

Array Access Analysis:

Input Size (n)	Array Accesses (Random w/ Majority)	Accesses/n
1,000	2,000	2.0
10,000	20,000	2.0
100,000	200,000	2.0
500,000	1,000,000	2.0

Экспортировать в Таблицы

The number of array accesses is consistently `2n`, corresponding to the two full passes over the array. This is also visualized in the `array_accesses_vs_size.png` plot.

5. Comparison with Partner's Algorithm

Here is a comparison between Student A's Boyer-Moore algorithm and Student B's Kadane's Algorithm.

- **Similarities:**
  - **Optimal Complexity:** Both algorithms are optimal, with O(n) time and O(1) space complexity.
  - **Single Pass Nature:** Both solve their respective problems by iterating through the input array in a single, continuous sweep (though Boyer-Moore requires a second pass for verification).
  - **Dynamic Programming Principle:** Both algorithms build up the solution by making a decision at each element based on previously computed values.
- **Differences:**
  - **Problem Domain:** Boyer-Moore finds the *frequency* of a single element (majority), while Kadane's algorithm finds the maximum *sum* of a contiguous subarray.
  - **Verification Step:** Kadane's algorithm is a true single-pass algorithm. Boyer-Moore requires a second pass to verify its candidate, as the first pass does not guarantee that the candidate is the majority element.
  - **Number of Operations:** Due to the two passes, Boyer-Moore performs approximately `2n` array accesses and up to `2n` comparisons, whereas Kadane's algorithm performs roughly `n` accesses and `2n` comparisons.

---

6. Recommendations and Conclusion

6.1 Critical Optimizations (Low Priority)

1. **Remove Redundant Null Check:** As identified in Section 3, removing the `candidate != null` check in the verification loop would make the code slightly more efficient and clean, although the performance gain would be negligible.

6.2 Optional Enhancements

1. **Generics:** The current implementation is specific to `int[]`. It could be generalized using Java generics to find the majority element in an array of any object type, provided a proper `equals()` method is defined.
2. **Early Exit:** If, during the verification pass, the `count` of the candidate plus the number of remaining elements is less than `n/2`, the algorithm could terminate early. This would only be an optimization for specific data distributions where the majority element is sparse in the latter part of the array.

6.3 Final Assessment

Student A has delivered an exemplary implementation of the Boyer-Moore Majority Vote algorithm. The project is a model of good software engineering practices.

- **Correctness and Robustness:** The code is logically sound and handles all edge cases gracefully, as demonstrated by the comprehensive unit tests.
- **Performance:** The implementation is highly performant, achieving the optimal O(n) time and O(1) space complexity.
- **Documentation and Structure:** The project is well-documented, clearly structured, and easy to understand, build, and run.
- **Analysis:** The inclusion of detailed performance tracking and benchmarking tools demonstrates a deep understanding of algorithmic analysis.

The project is of high quality and reflects a thorough understanding of both the algorithm and the principles of software development and analysis.

Grade: A (94/100)

Reviewer Signature: Student B (Saqyp Dias)

Date: October 5, 2025

Review Status: Complete