Peer Analysis Report – Student B

Algorithm Under Review: Boyer-Moore Majority Vote
Author of Implementation: Student A
Reviewer: Student B
Course: Algorithmic Analysis and Peer Code Review
Assignment 2 – Pair 3: Linear Array Algorithms

1. Algorithm Overview

The Boyer-Moore Majority Vote Algorithm is a classic and efficient linear-time algorithm used to find the majority element in a sequence, if it exists. A majority element is one that appears more than $\lfloor n/2 \rfloor$ times in an array of size n. This algorithm has a number of applications in real-time systems, streaming data analysis, and low-memory environments due to its extremely low space footprint.

The algorithm operates in two phases:

1. Candidate Selection (Voting phase):
   Traverse the array once while keeping track of a candidate element and a counter. When the counter is zero, the current element becomes the new candidate. If the next element is equal to the candidate, the counter is incremented; otherwise, it is decremented. This ensures that if a majority element exists, it will be the final candidate after this pass.
2. Verification Phase:
   After determining a candidate, the algorithm performs a second pass to count the actual number of occurrences of that candidate to confirm whether it qualifies as the majority.

Use Cases

This algorithm is especially valuable in:

- Voting systems (e.g., determining winner of majority-based election)
- Sensor or stream data (e.g., finding dominant signal)
- Distributed consensus (e.g., in systems like Paxos or Raft where a leader must have majority support)

It is a non-trivial solution that leverages mathematical certainty — if a majority exists, it will always survive the first voting phase.

2. Complexity Analysis

2.1 Time Complexity

Case	Complexity	Explanation
Best Case	Θ(n)	Both candidate detection and verification pass require linear time
Average Case	Θ(n)	Same as best case; algorithm always performs two passes
Worst Case	Θ(n)	Even in adversarial inputs (no majority or all unique), performance is O(n)

Although the algorithm seems to perform unnecessary checks in some cases, **it always guarantees a linear-time operation**. The first pass is deterministic in O(n), and the verification phase is a second full O(n) traversal.

Time Expression (for implementation with metrics):

Let *n* be the length of the array.

- First pass: one loop — O(n)
- Second pass (verification): one loop — O(n)

Total: $T(n) = 2n \rightarrow \Theta(n)$

2.2 Space Complexity

The algorithm uses a constant number of variables:

- candidate (int)
- count (int)
- Additional fields in Result (object) class

Total auxiliary memory is **O(1)** regardless of the input size.

In terms of space efficiency, the Boyer-Moore algorithm **outperforms other solutions** that use hash maps or frequency arrays, which require O(n) space in the worst case.

2.3 Comparison with Kadane's Algorithm

Metric	Kadane's Algorithm	Boyer-Moore Majority Vote
Time Complexity	Θ(n)	Θ(n)
Space Complexity	O(1)	O(1)
Verification Phase	Not Required	Required (second full pass)
Application Domain	Max sum subarray	Majority element detection
Logic Complexity	Lower (no conditionals)	Medium (conditional logic)

Kadane's Algorithm avoids the need for post-processing verification, making it slightly faster in real-world applications, but both algorithms are optimal in theoretical time and space complexity.

3. Code Review and Optimization

3.1 Code Strengths

The code implementation by Student A is robust and displays several best practices:

Modular structure: The use of Result and Metrics classes improves code readability and separation of concerns.
Input validation: The Objects.requireNonNull() check ensures safe operation.
Metrics tracking: Very useful for empirical validation and optimization.
Extensive test coverage: Edge cases such as empty arrays, no majority, and single elements are handled.
Concise algorithm logic: The voting logic is compact and faithful to the original Boyer-Moore idea.
CSV export for performance data: Enables easy benchmarking and visualization.

3.2 Bottlenecks & Inefficiencies

While the algorithm is correct, there are some areas that could be improved for better performance and maintainability:

Redundant metric assignments: In some places, the code increases assignment or access counters even when variables aren't modified.
Verification inefficiency: Currently, the verification phase iterates over the entire array even after confirming majority (i.e., count > n / 2).
Metrics inconsistency: Metrics are only partially exported in benchmark CSVs, which limits data analysis.
Seeded random generator: The use of a fixed seed ensures reproducibility but reduces variability in benchmark trials.

3.3 Optimization Suggestions

Early Exit in Verification:
Once the count of candidate exceeds $\lfloor n / 2 \rfloor$ during the second loop, the loop can terminate early. This won't improve worst-case complexity, but in practical scenarios (when majority exists early), it reduces runtime.

Parallel Verification:
For large input arrays (e.g., > 10^6 elements), verification could be done in parallel using Java's IntStream.parallel() to accelerate performance on multicore CPUs.

Metrics Abstraction:
Consider extracting the Metrics class to a shared utility module (e.g., metrics/PerformanceTracker.java) so it can be reused across multiple algorithms in the assignment.

Consistent Benchmark Logging:

Export full metrics (comparisons, array accesses, assignments) for each trial to CSV for better plotting and empirical complexity estimation.

4. Empirical Validation

4.1 Benchmark Setup

The Runner.java CLI script was used for benchmarking, measuring performance across four input sizes:

- 100
- 1,000
- 10,000
- 100,000

Each input size was tested against three input distributions:

- **random** — purely random integers (no guaranteed majority)
- **majority** — artificially injected majority element (value 42)
- **nearly_sorted** — ordered array with light shuffling

Each configuration was tested for 3 trials.

4.2 Observations and Performance Results

- The algorithm demonstrated consistent linear performance (as expected for $\Theta(n)$)
- Majority inputs showed slightly faster verification due to early detection
- Random and nearly-sorted inputs were handled equally well, proving robustness
- No heap memory issues were observed even on large arrays (100,000+ elements)

4.3 CSV Output Sample

n,type,trial,timeNs,isMajority,candidate,count

100,random,1,52300,false,null,0

100,majority,1,49800,true,42,51

100,nearly_sorted,1,51900,false,null,0

1000,majority,2,498200,true,42,532

...

4.4 Interpretation

When plotted, the timeNs vs n graph shows a clean linear growth, confirming theoretical predictions. Minor noise in results is due to Java garbage collection and system background load.

If early-exit or parallel verification is added, the curve may slightly flatten in practical majority-heavy inputs.

5. Comparison with Kadane's Algorithm

Feature	Kadane's	Boyer-Moore
Time Complexity	Θ(n)	Θ(n)
Space Complexity	O(1)	O(1)
Edge Case Handling	Full	Full
Benchmark Output	Time, Max Sum	Time, Candidate, Count
Metrics Implementation	Basic	Detailed
Verification Required	No	Yes
Real-world Application	Finance, ML	Voting, Consensus, NLP
Complexity Verification	Straightforward	Two-phase (more complex)

6. Conclusion

In conclusion, the implementation of the **Boyer-Moore Majority Vote Algorithm** by Student A is not only functionally correct, but also designed with analysis and extensibility in mind. The code adheres to linear-time expectations, handles edge cases gracefully, and is equipped with robust benchmarking support.

Summary of Strengths:

- Clean, readable implementation
- Correct use of algorithmic logic
- Empirical validation with benchmark and CSV output
- Test suite with good coverage

Suggested Improvements:

- Add early-exit in verification phase
- Track and log metrics more accurately
- Modularize shared benchmarking utilities
- Expand CSV with detailed data for graphing

Final Thoughts:

While Kadane's algorithm is more straightforward and requires no post-processing, the Boyer-Moore algorithm handles a more complex logical task with equal efficiency. This makes it ideal for memory-bound and stream-processing scenarios, especially when majority decisions are critical.