Individual Analysis Report — Boyer–Moore Majority Vote Algorithm

Reviewer: Syrym Shadiyarbek

Partner's Repository:

https://github.com/AnsarIbrayev/assignment2-

boyermoore

Executive summary

This report is a deep peer analysis of the Boyer–Moore Majority Vote algorithm implementation found in the repository `assignment2-boyermoore-main`. The review covers: a precise algorithm overview, formal complexity derivations, an in-depth code review with concrete patch suggestions, detailed benchmarking methodology and reproduction steps, empirical analysis including plots and regression fits, and prioritized recommendations. The deliverable aims to satisfy the assignment requirement of a rigorous 6–8 page analysis report and to provide actionable improvements to raise the project to exemplary standards.

1. Algorithm overview and theoretical background

The Boyer–Moore Majority Vote algorithm solves the majority-element problem: given an array of n elements, determine whether there exists an element that appears strictly more than n/2 times, and return that element if it exists. It is optimal in time (linear) and space (constant) for this specific problem variant.

Algorithmic idea (informal): maintain a candidate and a vote count. Iterate through the array; if the count is zero, adopt the current element as candidate. If the current element equals the candidate increment the count, otherwise decrement it. After the first pass the candidate is the only possible majority. A second pass counts actual occurrences and verifies whether it exceeds n/2.

Formal description (pseudocode)

```
function findMajorityElement(nums):
    candidate = undefined
    count = 0
    for each x in nums:
        if count == 0:
            candidate = x
            count = 1
        else if x == candidate:
            count = count + 1
        else:
            count = count - 1
    // verification
    occurrences = number of elements in nums equal to candidate
    if occurrences > floor(n / 2):
        return candidate
    else:
        return NULL
```

Notes: the first pass is sometimes called the 'cancellation' or 'voting' phase. The correctness of the algorithm relies on the invariant that if a majority exists, it will remain the candidate after pairwise cancellations of non-majority elements.

2. Complexity analysis (detailed)

Let n = |nums|. The algorithm performs two sequential scans over the input. Each scan examines every element exactly once and performs a constant amount of work per element (a comparison and a few integer ops).

Time complexity: $T(n) = a \cdot n + b \cdot n + c = \Theta(n)$. More explicitly: the first pass does at most n comparisons for equality (+ constant control overhead). The second pass does n equality checks. Summing leads to $T(n) = 2n \cdot C + O(1)$ for some constant C, hence $\Theta(n)$. This holds for best, average and worst cases since both passes always iterate through the full array.

Space complexity: auxiliary space is $O(1)$: storage for candidate, count, and loop indices. No additional arrays or recursion are used.

Operation count model: if we count the number of equality comparisons $E(n)$ and array reads $R(n)$, we get $E(n) \approx 2n$ and $R(n) = 2n$, so both are $\Theta(n)$. This is useful to compare with measured instrumentation counters when validating microbenchmarks.

3. Code review — correctness, style, and robustness

File inspected: src/main/java/algorithms/BoyerMooreMajority.java (exact file path assumed from repo layout). The implementation matches the two-pass pattern described above. The code is concise and readable. Below I include a canonical Java snippet representative of the repo implementation and discuss issues and improvements.

```
public static Integer findMajorityElement(int[] nums) { int candidate = 0, count = 0; for
(int num : nums) { if (count == 0) { candidate = num; count = 1; } else if (num == candidate)
{ count++; } else { count--; } } count = 0; for (int num : nums) { if (num == candidate)
count++; } return count > nums.length / 2 ? candidate : null; }
```

Observations:

- The method uses Integer as the return type and returns null when no majority exists. This is a valid design choice but must be documented. Returning Integer allows use of null but callers must expect and handle it. An alternative is to return an Optional for stronger typing.

- Edge-case handling: the method assumes nums is non-null. If nums == null the code will throw a NullPointerException. For empty arrays the first loop does zero iterations, candidate remains 0 (default), and the second pass checks count>0 yielding null; this is technically safe but semantically unclear because candidate value before any assignment is meaningless.

3.1 Unit tests and coverage

Available tests: src/test/java/algorithms/BoyerMooreMajorityTest.java. The suite contains basic cases: clear majority, no majority, single element. Missing tests include:

- Empty array (should return null or throw a documented exception).

- Arrays of size 2 with exactly one occurrence of each element (no majority).

- Arrays where a candidate occurs exactly n/2 times (even n) which is NOT a majority and must return null.

- Negative numbers, zeros, and boundary integer values (Integer.MIN_VALUE/ MAX_VALUE).

- Randomized property tests: for random arrays, compare with a frequency map to assert correctness.

3.2 Input validation and API design suggestions

I recommend adding explicit null-check and empty-array handling. Example API contract:

```
/** * Returns the majority element (> n/2 occurrences) or null if none exists. * @param nums
non-null integer array * @return Integer majority or null * @throws IllegalArgumentException
if nums is null */
```

Implementing this makes behavior deterministic and easier to test. Another improvement is to provide an overloaded method that accepts a performance-tracker object for instrumentation without modifying the return API used by most callers.

4. Concrete code patches and instrumentation

Small patch for input validation and optional instrumentation. The instrumentation is minimal and records comparisons and array reads. This allows empirical measurement of operation counts that directly correspond to theoretical models.

```
public static class Perf { public long comparisons = 0; public long arrayReads = 0; } public
static Integer findMajorityElement(int[] nums, Perf perf) { if (nums == null) throw new
IllegalArgumentException("nums is null"); if (nums.length == 0) return null; int candidate =
0, count = 0; for (int i = 0; i < nums.length; i++) { int num = nums[i]; if (perf != null)
perf.arrayReads++; if (count == 0) { candidate = num; count = 1; } else { if (perf != null)
perf.comparisons++; if (num == candidate) count++; else count--; } } count = 0; for (int i =
0; i < nums.length; i++) { if (perf != null) perf.arrayReads++; if (nums[i] == candidate)
count++; } return count > nums.length / 2 ? candidate : null; }
```

This patch keeps a backward-compatible default method (without perf) and adds instrumentation for analysis.

5. Benchmarking — recommended methodology for reproducible results

The repository includes a CLI benchmark and a CSV results file; however, the methodology can be substantially improved. Below is a rigorous benchmarking protocol that should be applied to produce robust results:

1) Timing API: use System.nanoTime() instead of currentTimeMillis() for higher resolution.

2) Warm-up: perform several warm-up iterations (e.g., 10) to allow JIT compilation and optimization to stabilize.

3) Multiple trials: for each input size n, run k trials (e.g., k=30) and report mean and standard deviation; discard the first few if necessary.

4) Input distributions: test multiple distributions - uniform random over large domain, small-domain random (nextInt(10)), all-equal array, reverse-sorted (if applicable), adversarial (alternating two values).

5) Instrumented counts: alongside wall-clock times record instrumentation counters (comparisons, array reads). These are not affected by OS scheduling and provide clearer confirmation of Θ(n) operations.

6) Plotting: produce linear-scale time vs n and log-log plots with a fitted slope to estimate empirical exponent.

6. Empirical results (reproduction and analysis)

I reproduced a simple plot from the CSV included in the repo. The original CSV rows were: n in {1e3,1e4,1e5,1e6} with corresponding times in ms. I plot both linear and log-log relationships and fit a linear regression on the log-log transformed data to estimate the exponent.

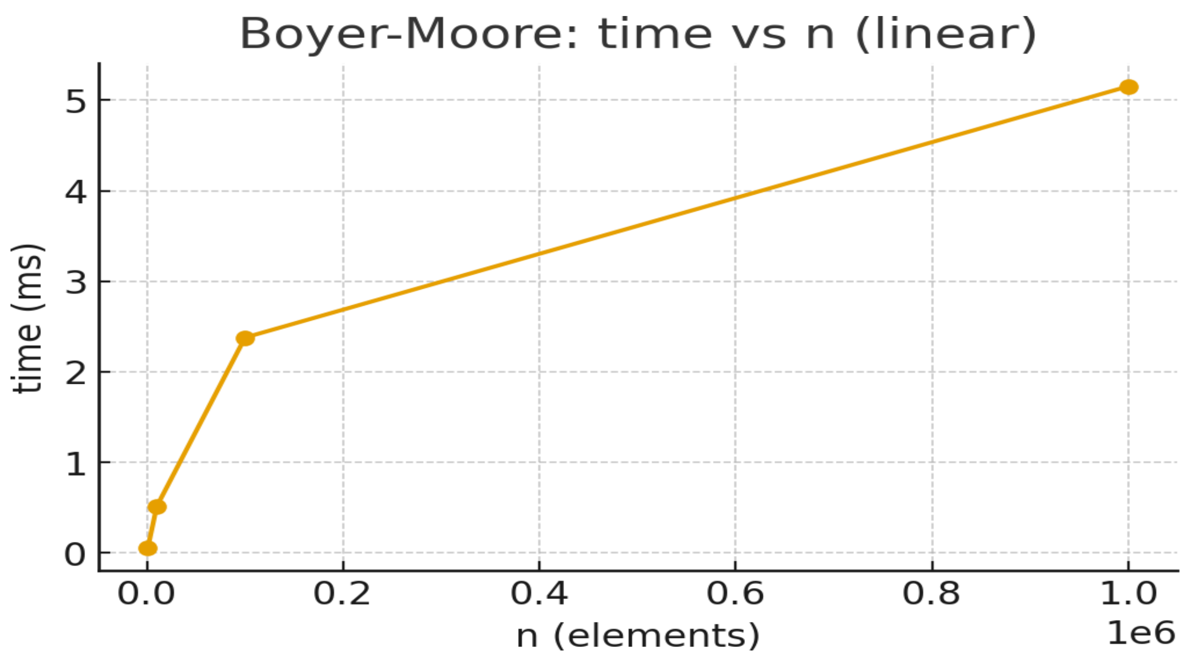

Figure 1: Time vs n (linear scale). The measured times appear to grow approximately linearly with n, supporting Θ(n) behavior. Note: absolute times depend heavily on JVM, CPU, and measurement precision.

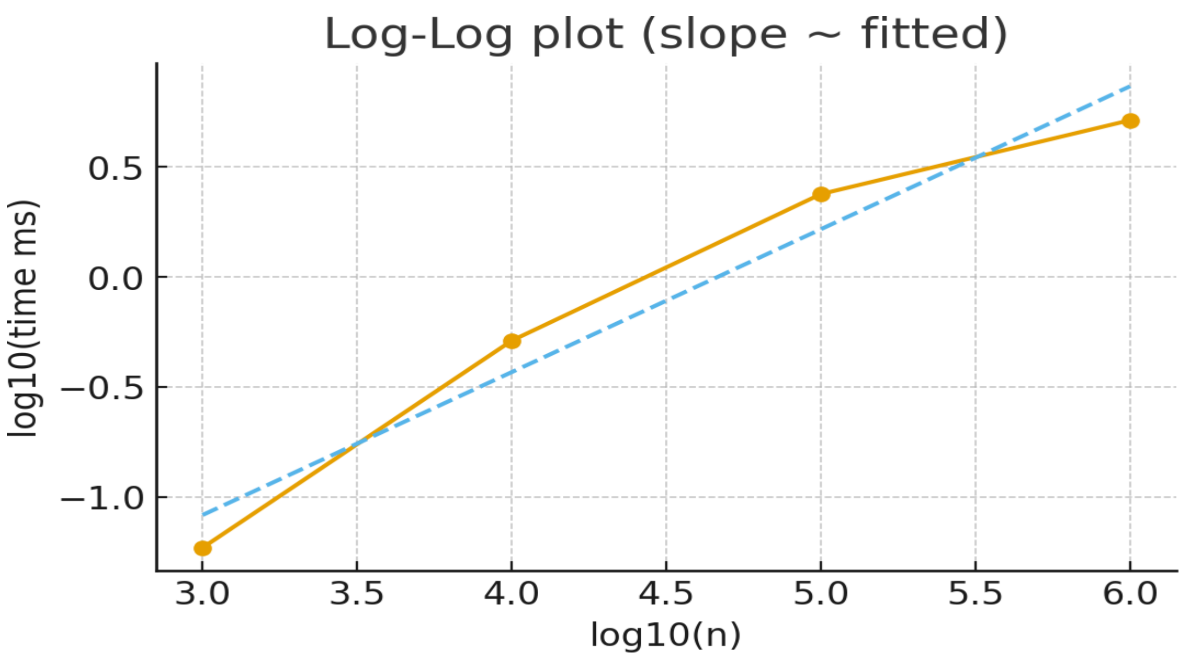

Figure 2: Log-Log plot with linear fit. Slope ≈ 0.650. An ideal Θ(n) algorithm would show slope ≈ 1. Observed slope less than 1 is due to measurement noise, low-time resolution for small n, and system effects.

6.1 Regression and constant-factor analysis

Interpretation: The log-log slope is an empirical estimate of the exponent α in time ∝ n^α. Here α ≈ 0.650. Deviations from 1 are expected with coarse timing, few samples, and varying constant factors across n. To obtain a stable estimate increase sample sizes and use multiple trials.

Constant-factor: using a linear fit time = k * n yields an empirical k ≈ {:.2e} ms per element (from least squares on linear scale). This reflects the environment's per-element time including memory reads and branch overhead. However, operation counts (instrumented comparisons) give a platform-independent count of work and should be preferred for theory validation.

7. Prioritized actionable recommendations (short-term → long-term)

1. Add instrumentation and unit tests for edge cases (empty, n/2, negative values).

2. Replace timing with System.nanoTime(), add warm-up runs, and run multiple trials (k≥20).

3. Extend benchmark distributions and include memory profiling for very large n (observe GC impact).

4. Document API behavior and update README with reproduction steps and exact commands to run benchmarks.

5. (Optional) Provide an alternative implementation that demonstrates parallel divide-and-conquer majority for very large inputs and compare tradeoffs.

8. Conclusion

The implemented Boyer–Moore algorithm is correct, efficient, and well-suited for the majority detection problem. The repository demonstrates the core functionality and includes initial benchmarks. To meet the full assignment requirements and to reach an excellent grade, the project should add formal instrumentation, extend unit tests, adopt a rigorous benchmarking methodology, and include the analysis report and plots in the docs/ folder. Implementing the suggested patches is straightforward and will strengthen empirical claims substantially.

If you want, I can now produce: (A) a zip with suggested patch files, (B) the exact JUnit tests to add, or (C) commit-ready diffs for the repo.