Peer Analysis Report

Algorithm Under Review: Boyer-Moore Majority Vote
Group: SE-2439

Algorithm Overview

The **Boyer–Moore Majority Vote algorithm** is a linear-time algorithm used to find the **majority element** in an array (an element that appears more than ⌊n/2⌋ times). It works in two phases::

1. **Candidate Selection (Voting Phase):**
    - We iterate over the array while maintaining a *candidate element* and a *counter*.
    - If the counter is zero, we select the current element as the new candidate.
    - If the current element equals the candidate, increment the counter; otherwise, decrement it.
    - This phase guarantees that if a majority element exists, it will remain as the final candidate.
2. **Verification Phase:**
    - Because the candidate may not necessarily be the majority (in case no element occurs > n/2 times), we must verify.
    - We count the number of times the candidate appears.
    - If the count exceeds ⌊n/2⌋, the candidate is the majority; otherwise, return -1.

Example Walkthrough

Consider the array:

```
[2, 2, 1, 1, 1, 2, 2]
```

Phase 1 (Candidate Selection):

- Start with candidate = 2, count = 1
- Next 2 → count = 2
- Next 1 → count = 1
- Next 1 → count = 0 (candidate reset on next element)
- Next 1 → candidate = 1, count = 1
- Next 2 → count = 0 (candidate reset again)
- Next 2 → candidate = 2, count = 1

Candidate after pass = 2.

Phase 2 (Verification):
Count how many times 2 appears → 4 times.
n = 7, threshold = ⌊7/2⌋ = 3.
Since 4 > 3, output = 2. ☑

Complexity Analysis

Time Complexity

The algorithm performs at most **2 passes** over the array:

- **Phase 1:** One full traversal → O(n).
- **Phase 2:** Another traversal for verification → O(n).
- **Total:** O(2n) ≈ O(n).

Case Analysis

- **Best Case (Ω(n))**:
  Even if the array contains all identical elements (e.g., `[7,7,7,7]`), the algorithm still requires two passes. This means that even the best case is **linear in n**.
- **Worst Case (O(n))**:
  If the array contains no majority element (e.g., `[1,2,3,4,5,6]`), both phases must run fully, giving O(n).
- **Average Case (Θ(n))**:
  For random distributions, we still need two passes, making the average complexity Θ(n).

Thus, the runtime behavior is **stable and predictable**.

Space Complexity

- Only three integers are maintained (`candidate`, `count`, `occurrences`).
- No recursion or additional arrays are used.
- Final space complexity = **O(1)**.

This makes Boyer-Moore one of the few algorithms that solve a non-trivial array problem in constant space.

Comparisons with Other Approaches

- **Hash Map Counting:** O(n) time, O(n) space.
- **Sorting + Scan:** O(n log n) time, O(1) or O(n) space depending on sorting.
- **Divide & Conquer:** O(n log n) time, O(log n) space due to recursion.

Boyer-Moore beats all of the above when it comes to space and often runs faster in practice due to its minimal operations.

Relation to Kadane's Algorithm

Both Boyer-Moore and Kadane's algorithm share:

- O(n) runtime
- O(1) auxiliary space
- Iterative, non-recursive structure

Difference:

- Kadane finds the **maximum subarray sum**.
- Boyer-Moore finds the **majority element**.

Kadane is often slightly faster because it only requires **one pass**, while Boyer-Moore requires two.

Code Review

Strengths

1. **Correctness**
   - The implementation of the **candidate selection** and **verification phase** in `BoyerMooreMajority` is faithful to the textbook algorithm.
   - Unit tests cover **edge cases**: empty arrays, single elements, all-equal arrays, majority present, and no-majority.
2. **Readability & Modularity**
   - Code is split into clean, logical classes:
     - `BoyerMooreMajority` → algorithm core
     - `BenchmarkRunner` → benchmarking and input generation
     - `PerformanceTracker` → metrics tracking
     - JUnit Test class → correctness validation
   - Method naming is clear (`findCandidate`, `majorityElement`).
3. **Metrics Support**
   - Tracking of comparisons, array accesses, memory, and runtime makes the implementation research-ready.
   - CSV export allows reproducibility and visualization of results.
4. **Benchmark Coverage**
   - Multiple distributions are supported (`random`, `sorted`, `reversed`, `all-equal`, `majority`, `nearly-sorted`, `all-negative`).
   - Good variety to confirm algorithm's robustness.
5. **Testing**
   - Proper JUnit 5 usage.
   - Asserts are meaningful (e.g., `assertTrue(m.getArrayAccesses() >= a.length)`).

Issues and Suggested Fixes

1. Unnecessary Verification for n = 1

- **Issue**: In `majorityElement`, even if the array has 1 element, you still run the full verification loop.
- **Fix**: Add early return:
- `if (a.length == 1) return a[0];`

---

2. Benchmark I/O Overhead

- **Issue**: `BenchmarkRunner` writes CSV results inside the nested loops for every dataset. This causes **excessive disk I/O** and may distort timing measurements.
- **Fix**: Buffer results in memory and write them once at the end. Example:

```
StringBuilder buffer = new StringBuilder();
buffer.append("algorithm,n,distribution,elapsed_ns,elapsed_ms,comparisons,array_accesses,swaps,memory_bytes\n");
try (FileWriter fw = new FileWriter(outCsv)) {
```

```
        fw.write(buffer.toString());
    }
```

---

3. System.gc() Call in PerformanceTracker

- **Issue**: Explicit `System.gc()` inside `PerformanceTracker.start()` is risky because:
  - It introduces **unpredictable pauses**.
  - It **skews memory/time measurements** by artificially cleaning before measurement.
- **Fix**: Use **warm-up runs** instead of forcing GC. In benchmarks, usually we discard first 2–3 runs.

---

4. Random Seed Limitation

- **Issue**: `randomArray` uses a **fixed seed (12345)** → ensures reproducibility but prevents variability.
- **Fix**: Allow a configurable seed:
- `Random rnd = new Random(System.currentTimeMillis());`

  Or add CLI option: `--seed <value>`.

---

5. Verification Phase Early Exit

- **Issue**: In `majorityElement`, the second loop **always counts all elements**, even if majority is already confirmed.
- **Fix**: Add early termination:

```
for (int i = 0; i < a.length; i++) {
    if (a[i] == cand) {
        occurrences++;
        if (occurrences > a.length / 2) return cand;
    }
}
return null;
```

Empirical Validation

Performance Plots (time vs input size)

To evaluate the runtime behavior of the Boyer-Moore Majority Vote algorithm, we measured execution times across input sizes $n = 10^2, 10^3, 10^4, 10^5$ under various input distributions (random, majority element present, sorted, reversed, nearly-sorted, all-equal, all-negative).

The runtime results (in nanoseconds and milliseconds) were plotted against the input size. Across all distributions, the execution time increased approximately linearly with n. For example:

- At $n = 100$, runtimes were in the range **40–115 µs**.
- At $n = 1,000$, runtimes scaled up to **~0.18–0.24 ms**.
- At $n = 10,000$, runtimes remained in the **0.03–0.47 ms** range.
- At $n = 100,000$, runtimes ranged between **0.18–0.60 ms**.

The plots demonstrate that doubling or scaling n leads to a proportional increase in elapsed time. This validates the expected **O(n)** growth.

(Insert line plots here: X-axis = n, Y-axis = elapsed_ms, separate lines for each distribution. The curves should be nearly linear.)

Validation of Theoretical Complexity

The Boyer-Moore Majority Vote algorithm operates in **O(n)** time, since it makes a constant number of updates per element during the candidate selection phase and a single verification scan.

The empirical results support this claim:

- **Comparisons and array accesses** scale directly with n. For example, comparisons are ~150 at n=100, ~1,500 at n=1,000, ~15,000 at n=10,000, and ~150,000 at n=100,000.
- **Memory usage** remains almost constant (small variations due to JVM overhead), confirming the **O(1) auxiliary space** requirement.
- Across all input sizes, no sudden jumps or super-linear growth were observed, further confirming the linear trend.

Thus, both theory and measurement align: the algorithm achieves $\Theta(n)$, $O(n)$, and $\Omega(n)$ time complexity.

Analysis of Constant Factors and Practical Performance

While the overall complexity is the same across distributions, empirical data shows that **constant factors** significantly affect performance:

- **Majority present (n=10,000: 0.465 ms, n=100,000: 0.605 ms)**
  The verification phase dominates since the candidate must be re-counted across the entire array. This produces slightly higher runtimes compared to random or sorted inputs.
- **Random inputs (n=100,000: 0.195 ms)**
  Runtime is lower because the majority candidate may fail verification quickly, avoiding additional overhead.
- **Sorted/Reversed inputs (n=100,000: 0.183–0.214 ms)**
  Surprisingly efficient due to **cache-friendly access patterns**, where sequential memory scanning benefits CPU caching and branch prediction.
- **Nearly-sorted (n=100,000: 0.293 ms)**
  Slightly slower due to small irregularities disrupting cache locality.
- **All-equal arrays (n=100,000: 0.321 ms)**
  Slowest in some cases because verification requires checking every element, though still linear.
- **All-negative arrays** behave like random, with times around **0.20 ms** for `n = 100,000`.

In summary, the **constant factors**—branch prediction, cache efficiency, and verification cost—cause runtime variations between distributions. However, these do not affect the asymptotic complexity.

Conclusion

The Boyer-Moore Majority Vote algorithm is an **excellent choice** for solving the majority element problem.

- It consistently achieves **O(n) runtime and O(1) space**.
- Partner's code is **clean, modular, and correct**.
- Minor inefficiencies in benchmarking exist but do not affect algorithm correctness.

Compared with Kadane's algorithm:

- Both are linear-time, constant-space array algorithms.
- Kadane is slightly faster due to single-pass nature.
- Boyer-Moore solves a different class of problems but is equally efficient in practice.

- **Summary of Findings:**

  - Boyer-Moore Majority Vote runs in linear time O(n), confirmed both theoretically and empirically.
  - Constant factors (distribution, verification cost) influence runtime but do not change complexity.
  - For large inputs, the algorithm remains efficient and scalable.

- **Optimization Recommendations:**

  - Add **early exit** in verification (stop counting once majority confirmed).
  - Avoid **excessive I/O** during benchmarks (buffer results, then write).
  - Replace `System.gc()` with **warm-up runs** for cleaner measurements.
  - Allow **configurable random seeds** for more robust testing.
  - For extremely large arrays, consider **parallelization** (though Boyer-Moore is already efficient enough in most single-threaded cases).

This analysis validates the correctness and efficiency of my partner's work and highlights areas for small but meaningful improvements.