Individual Analysis Report — Boyer–Moore Majority Vote

Author: Kenzhebayev Madiyar (Student B)
Partner: Aldiyar Amangeldi Implementation by Student A (Boyer–Moore Majority Vote)
Course: DAA — Assignment 2

1. Algorithm Overview

Problem. Given an array A of length n, determine whether there exists an element (the *majority*) that occurs strictly more than n/2 times; if so, return it.

Algorithm (Boyer–Moore Majority Vote).
The algorithm runs in two phases:

1. **Candidate selection (single pass):** keep a candidate and a count. Traverse the array; when count == 0, set candidate = A[i], count = 1. Otherwise, increment count when A[i] == candidate, else decrement count. At the end the candidate is a possible majority (if one exists).
2. **Verification (single pass):** count occurrences of candidate and check if count > n/2. The implementation uses an *early termination*: as soon as the verification count reaches the threshold ⌊n/2⌋+1, it returns true.

Practical notes from the provided implementation.

- The implementation uses Integer[] (boxed integers) and Objects.equals() for comparisons.
- A Metrics object tracks comparisons, arrayAccesses, memoryAllocations, and wall-clock time (nanoseconds).
- Utilities include generateTestArray, shuffleArray, CLI benchmarking and export facilities.
- The code uses early termination in verification and collects per-run PerformanceResult information for statistical summaries.

Summary. The implementation faithfully follows the standard two-pass Boyer–Moore approach with useful engineering additions (metrics, data generators, CLI). The algorithm is conceptually simple and attractive because it is linear-time and constant-space.

2. Complexity Analysis

2.1 Time complexity – derivation & justification

Let n be the array length.

Phase 1 — Candidate selection: single loop across all n elements. Each iteration does O(1) work (assignments, one comparison of candidate with A[i], increment/decrement). Therefore phase 1 costs T1(n) = c1 * n + c0 for constants c1,c0.

Phase 2 — Verification: another single loop across n elements, likewise O(1) per element, but the implementation may terminate early when count reaches the threshold. Worst case (no early termination) is T2(n) = c2 * n + c0'.

Total time T(n) = T1(n) + T2(n) = (c1 + c2) * n + c' = Θ(n).

- **Worst case:** Θ(n). Example: no majority present or majority spread in a way that prevents early-termination in verification → both passes traverse full array.
- **Best case:** Θ(n). Even if candidate found quickly, verification still must at least visit elements until confirming or disproving (implementation may short-circuit early in verification, but asymptotically a single pass remains Θ(n)). For constant-factor improvement, best-case constant factors are smaller, but asymptotic is still Θ(n).
- **Average case:** Θ(n) — for random inputs typical performance close to 2n operations (two passes). Early termination reduces constant factor but does not change Θ.

Big-O, Θ, Ω summary:

- Time: O(n), Θ(n), Ω(n).

2.2 Space complexity

Auxiliary space: the algorithm uses a constant number of variables (candidate, count, counters in Metrics) irrespective of n → O(1) auxiliary space.

Caveat: the implementation uses Integer[] (boxed values) as input and sometimes generates new arrays; when testing/generating arrays, memory for test arrays is O(n), but that's required for input; algorithmic auxiliary memory remains O(1).

2.3 Recurrence relations

Not applicable — algorithm is iterative (two linear passes), no recursion or divide-and-conquer recurrence.

3. Code Review & Optimization

3.1 Correctness & edge cases

- **Null input:** handled: returns Result with error message — good.
- **Empty array:** returns null majority with hasMajority=false. Implementation ends timer and returns. Good.
- **Single-element array:** correctly returns that element as majority. Good.
- **Candidate selection correctness:** Implementation of findCandidate follows classical algorithm; however one minor detail: comparisons counter only increments when count != 0, so metrics are inconsistent (see metrics section).
- **Verification correctness:** Uses majorityThreshold = array.length / 2 + 1 and early termination when count >= threshold; returns count > n/2 otherwise. Correct and consistent.

3.2 Efficiency issues and performance bottlenecks

1. **Use of Integer[] (boxing) rather than int[].**
    a. Autoboxing/unboxing significantly increases runtime and memory overhead and affects cache behavior.
    b. Each access to Integer may allocate or dereference object headers; this inflates arrayAccesses measured and real runtime.
    c. **Impact:** For large n ($\geq 10^4$), cost of boxing can dominate constant factors.
2. **Use of Objects.equals(candidate, array[i]).**
    a. For boxed integers, this is safe (handles nulls), but when data is primitive int, == is cheaper and clearer.
    b. Objects.equals may be slightly slower due to null checks and method overhead.
3. **Metrics overhead built-in to algorithm.**
    a. The code increments counters (metrics.incrementArrayAccesses(), metrics.incrementComparisons()) inside the hot loops. If benchmarking for algorithmic speed, metric collection changes the runtime constants.
    b. **Suggestion:** Provide *instrumented* and *non-instrumented* variants. Non-instrumented version (no metrics) should be used for timing/benchmarks.
4. **generateTestArray for "no majority" case:**

    a. Uses i % (size / 2 + 1) — for small sizes this pattern may accidentally produce majority-like distributions (e.g., small size values with modulus generating duplicates). Prefer using random data without bias.
5. **Randomness in shuffleArray using Math.random():**
    a. Math.random() is thread-safe but slower; prefer java.util.Random or ThreadLocalRandom.current() for speed and reproducibility.
    b. Also, the shuffle uses Math.random() with implicit global seed; for reproducibility supply Random with fixed seed as optional parameter.
6. **Metrics 'memoryAllocations' counter only increments once for the Metrics object.**
    a. This is misleading if other allocations occur (e.g., Result object, arrays created in generator). For accurate allocation accounting use Runtime or a profiler (instrumentation API); manual counters are fragile.
7. **Potential micro-bugs in metrics accounting:**
    a. Some code increments arrayAccesses for length checks and null checks; mixing logical checks with metrics makes metrics noisy and hard to interpret. Define consistently what counts as an "array access".

3.3 Maintainability & style

- Overall code is readable and modular (separated findCandidate, verifyCandidate, utilities, CLI).
- Suggestions:
    - Add Javadoc for public methods and classes.
    - Rename Metrics fields and methods to be explicit (e.g., incComparison(), incArrayAccess()).
    - Use logger (SLF4J) for CLI output in production code; keep System.out for homework/demo.
    - Avoid long CLI methods — split responsibilities (parsing, executing, printing).
    - Add proper unit tests with assertions rather than relying on demos in Main.

3.4 Specific code-change suggestions (small patches)

A. Prefer int[] API

Change public signature:

public static Result findMajorityElement(int[] array)

and helper generators to produce int[]. This eliminates boxing overhead.

B. Provide non-instrumented fast path

Add:

```
public static int findMajorityElementFast(int[] array) {
   if (array == null || array.length == 0) throw ...
   int candidate = 0, count = 0;
   for (int v : array) {
      if (count == 0) { candidate = v; count = 1; }
      else if (candidate == v) count++; else count--;
   }
   return candidate;
}
```

And separate verify function for final check.

C. Replace Objects.equals with == when using primitives.

D. Use ThreadLocalRandom and seeded Random in tests:

```
Random rnd = new Random(seed);
```

E. Improve generateTestArray randomness

For no-majority arrays:

```
int[] arr = new int[size];
for (int i=0;i<size;i++) arr[i] = rnd.nextInt(bound);
```

Ensure bound sufficiently large to avoid accidental majority.

4. Empirical Validation (≈ 2 pages)

4.1 Experimental setup (how to run)

- Use non-instrumented (fast) variant for timing (findMajorityElementFast + verifyFast) or run findMajorityElement but disable metrics collection option if available.
- Input sizes: n = 100, 1,000, 10,000, 100,000.

- For each n and scenario (with majority / without majority), run R = 10 independent trials; measure:
    - wall-clock time per run (use System.nanoTime() and compute ms)
    - number of comparisons and array accesses (from Metrics) — *only if instrumentation is enabled*
- For reproducibility use fixed seeds for RNG.
- Commands:
    - via CLI: java -jar DAA_assignment_2.jar benchmark (or use the provided BenchmarkRunner with options)
    - via Maven: mvn -DskipTests=false -Dexec.mainClass=org.example.Main -Dexec.args="benchmark"

4.2 Data collection and expected scaling

- **Expected:** runtime ∝ n (linear). If we plot time_ms vs n on linear axes, we expect a near-straight line through origin with slope ≈ k (constant factor).
- **Verification metric:** compute best-fit slope k = time_ms / n and check stability across n.
- **Comparing instrumented vs non-instrumented:** instrumented runs will have larger k; measure factor f = k_instrumented / k_fast.

4.3 Example table template (fill with measured values)

n	mode	avg_time_ms	std_ms	avg_comparisons	avg_array_accesses
100	withMajority	0.05	0.01	200	200
100	noMajority	0.06	0.02	200	200
1,000	withMajority	0.4	0.03	2000	2000
10,000	noMajority	4.1	0.2	20000	20000
100,000	optimized	35.6	1.1	200000	200000

4.4 Plotting & complexity verification

- Plot 1: Time (ms) vs n (linear). Fit a line y = k*n + b and report R^2. For Boyer–Moore expect $R^2 \approx 1$.

- Plot 2: Time per element time_ms / n vs n — should be roughly constant.
- Plot 3: Comparisons and array accesses vs n — should scale linearly, slope ≈ 2 (two passes), but early termination may reduce slope < 2 for some inputs.

4.5 Optimization-impact measurement

- **Before:** instrumented runtime k1.
- **After (int[] + non-instrumented):** runtime k2.
- Compute speedup S = k1 / k2. Expect S > 1; using primitives often yields measurable speedups (2×–5× depending on JVM, GC, and n).
- Provide tables with both pre/post values and relative improvement.

5. Conclusion

Summary of findings

- The implementation correctly implements Boyer–Moore Majority Vote and handles edge cases (null, empty, single-element) properly. The early-termination optimization in verification is a sensible constant-factor improvement.
- The algorithm's theoretical complexity is Θ(n) time and O(1) auxiliary space — the implementation preserves these properties.
- Main practical drawback: heavy use of boxed Integer[] and Objects.equals() increases constant factors significantly and inflates measured metrics. The metrics counters interleaved with hot loops also change practical timings, so

separate instrumented and non-instrumented paths are recommended.

Recommendations

1. **Switch public API to int[]** to remove boxing overhead. Keep a thin wrapper overload that accepts Integer[] if required for backward compatibility.
2. **Provide two modes:** instrumented (collect metrics) and fast (no metrics) so benchmarks measure pure runtime.
3. **Use reproducible RNG seeds** in benchmarks and ThreadLocalRandom/Random with seeds in tests.
4. **Improve testing:** add randomized cross-validation comparing algorithm result to a brute-force hashmap counting (property-based tests).
5. **Clarify and standardize metrics definitions** (what constitutes an array access, whether comparisons count null checks) and avoid counting non-algorithmic checks (array length, null check) inside performance counters.
6. **Document** the API with Javadoc and provide usage examples in README.

Final assessment. The partner's work is well-structured, thoroughly instrumented and ready for empirical study; with the few suggested code changes it will be production-quality for this assignment and give significantly more reliable and faster benchmarks.