**Assignment 2: Algorithmic Analysis and Peer Code Review**

**Analysis Report**

**Boy er-Moore Algorithm**

**PAIR 3: Linear Array Algorithms**

**Student A: Aiymzhan Abilgazay** (Boy er-Moore Majority Vote (single-pass majority element detection))

**Student B: Marzhan Tulebayeva** (Kadane's Algorithm (maximum subarray sum with position tracking))

**Peer Analysis — Majority Vote Algorithm (Boyer–Moore Majority Vote)**  
**Student Name:** Marzhan Tulebaeva

**1. Algorithm Overview**

The **Boyer–Moore Majority Vote Algorithm** efficiently identifies an element that appears more than ⌊n/2⌋ times in an array, if such an element exists. It operates in two conceptual steps:

1. **Candidate Selection (single pass)**: Iterate over the input array, maintaining a candidate and a counter. If the counter is zero, the current element becomes the candidate. If the current element equals the candidate, increment the counter; otherwise, decrement it. After this pass, the candidate is a potential majority element.
2. **Verification (optional second pass)**: Count the occurrences of the candidate in the array and verify if it occurs more than ⌊n/2⌋ times. This ensures correctness if a majority element may not exist.

**Use Case:** This algorithm is ideal when minimal extra memory is available, as it finds the majority element in **linear time** and **constant space**, making it a standard solution for majority detection.

**Implementation Summary:**

The MajorityVote class provides two overloads:

* int majorityElement(int[] nums) — a wrapper constructing a MajorityMetrics object.
* int majorityElement(int[] nums, MajorityMetrics metrics) — main routine recording metrics:
  + metrics.incrementArrayAccesses() per array read
  + metrics.incrementComparisons() per comparison
  + Verification pass counts occurrences of the candidate

Edge cases are handled: nums == null or nums.length == 0 returns -1, nums.length == 1 returns the single element.

public int majorityElement(int[] nums, MajorityMetrics metrics) {

if (nums == null || nums.length == 0) return -1;

if (nums.length == 1) return nums[0];

int candidate = 0, count = 0;

for (int num : nums) {

metrics.incrementArrayAccesses();

if (count == 0) candidate = num;

count += (num == candidate) ? 1 : -1;

metrics.incrementComparisons();

}

// Verification pass

count = 0;

for (int num : nums) {

metrics.incrementArrayAccesses();

if (num == candidate) count++;

metrics.incrementComparisons();

}

return (count > nums.length / 2) ? candidate : -1;

}

**2. Complexity Analysis**

**2.1 Time Complexity**

Let **n** be the number of elements in the input array.

**Candidate selection pass:**

* Loop iterates over n elements. For each element:
  + 1 array access
  + 1 comparison
  + O(1) updates
* Cost: Θ(n)

**Verification pass (optional):**

* Another loop over n elements to count occurrences
* Cost: Θ(n)

**Total:** Θ(n) + Θ(n) = Θ(n)

**Worst/Best/Average Cases:**

* Worst-case: Θ(n) (linear scan with verification)
* Best-case: Θ(n) (even if majority is early, full scan required)
* Average-case: Θ(n)

**Formal notation:**  
T(n) = c1*n + c2*n + O(1) = Θ(n)

**2.2 Space Complexity**

* Only a few integers (candidate, count, countCheck) and the metrics object → O(1)
* Algorithm is **in-place**, does not modify or copy input
* **Auxiliary space:** Θ(1)

|  |  |  |
| --- | --- | --- |
| Resource | Usage | Complexity |
| candidate, count, countCheck | 3 integers | O(1) |
| MajorityMetrics object | 1 object | O(1) |
| Total Auxiliary Space | - | Θ(1) |

**3. Code Review & Optimization**

**3.1 Quality and Maintainability**

**Positive Points:**

* Clear separation of wrapper and main routine
* Edge cases handled
* Metrics consistently recorded

**Areas for Improvement:**

* Metrics calls inside the hot loop add overhead
* Redundant comparisons in verification pass
* Minor style inconsistencies (braces, spacing)

**3.2 Inefficiency Detection**

* Two passes versus single-pass verification: if a majority is guaranteed, verification is redundant.
* Metrics overhead may dominate benchmarks; uninstrumented mode recommended for pure performance evaluation.
* Wrapper allocation: constructing a new metrics object each call could be optimized with a NOOP metrics singleton.

**3.3 Optimization Suggestions**

1. Avoid per-iteration metric calls by using local counters and committing after loops.
2. Remove verification pass if majority is guaranteed.
3. Cache frequently used fields locally in hot loops.
4. Use a NOOP metrics object for uninstrumented runs.
5. Simplify verification comparisons (only count once per element).
6. Test with adversarial inputs (alternating arrays, arrays with no majority).

**4. Empirical Validation**

**Measurement Plan:**

* Input sizes: n = 100, 1,000, 10,000, 100,000
* Warmup: 5 iterations, Measurement: 5 iterations, Forks: 1
* Modes:
  1. Algorithm-only (NOOP metrics)
  2. Instrumented (full metrics)
  3. Optimized variant (local counters, reduced method calls)
* Metrics: time (ns/op), array accesses, comparisons, allocations, GC pauses

**Expected Plots:**

* Time vs n: linear
* Comparisons vs n: ~2n with verification
* Array accesses: ~2n
* Allocations: constant

**Interpretation:**

* Execution time scales linearly with n
* Comparisons ≈ k\*n, k ≈ 2 (selection + verification)
* Array writes = 0
* Metrics overhead measurable; micro-optimizations reduce constant factor without changing asymptotics

## Example Empirical Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n | Time (ms) | Array Accesses | Comparisons | Allocations |
| 100 | 0.1 | 200 | 200 | 1 |
| 1,000 | 1.0 | 2,000 | 2,000 | 1 |
| 10,000 | 10 | 20,000 | 20,000 | 1 |
| 100,000 | 95 | 200,000 | 200,000 | 1 |

**5. Summary & Conclusions**

**Main Findings:**

* Boyer–Moore Majority Vote is asymptotically optimal: Θ(n) time, Θ(1) space
* Implementation is correct, robust, and includes metrics instrumentation
* Practical concern: method-call overhead in hot loop

**Recommendations:**

1. Keep verification unless majority is guaranteed
2. Benchmark instrumented and uninstrumented modes
3. Apply micro-optimizations (local caching, reduced method calls)
4. Add randomized and adversarial tests

**Reproducibility:**

* Use JMH for measurements
* Report averages, comparisons, array accesses, and plots
* Linear regression confirms Θ(n) scaling