INDIVIDUAL ANALYSIS REPORT

Student: Maulen Daryn (Student B)
Group: SE-2403
Topic: Pair 3: Linear Array
Algorithms (Bayazit A. – Daryn M.)

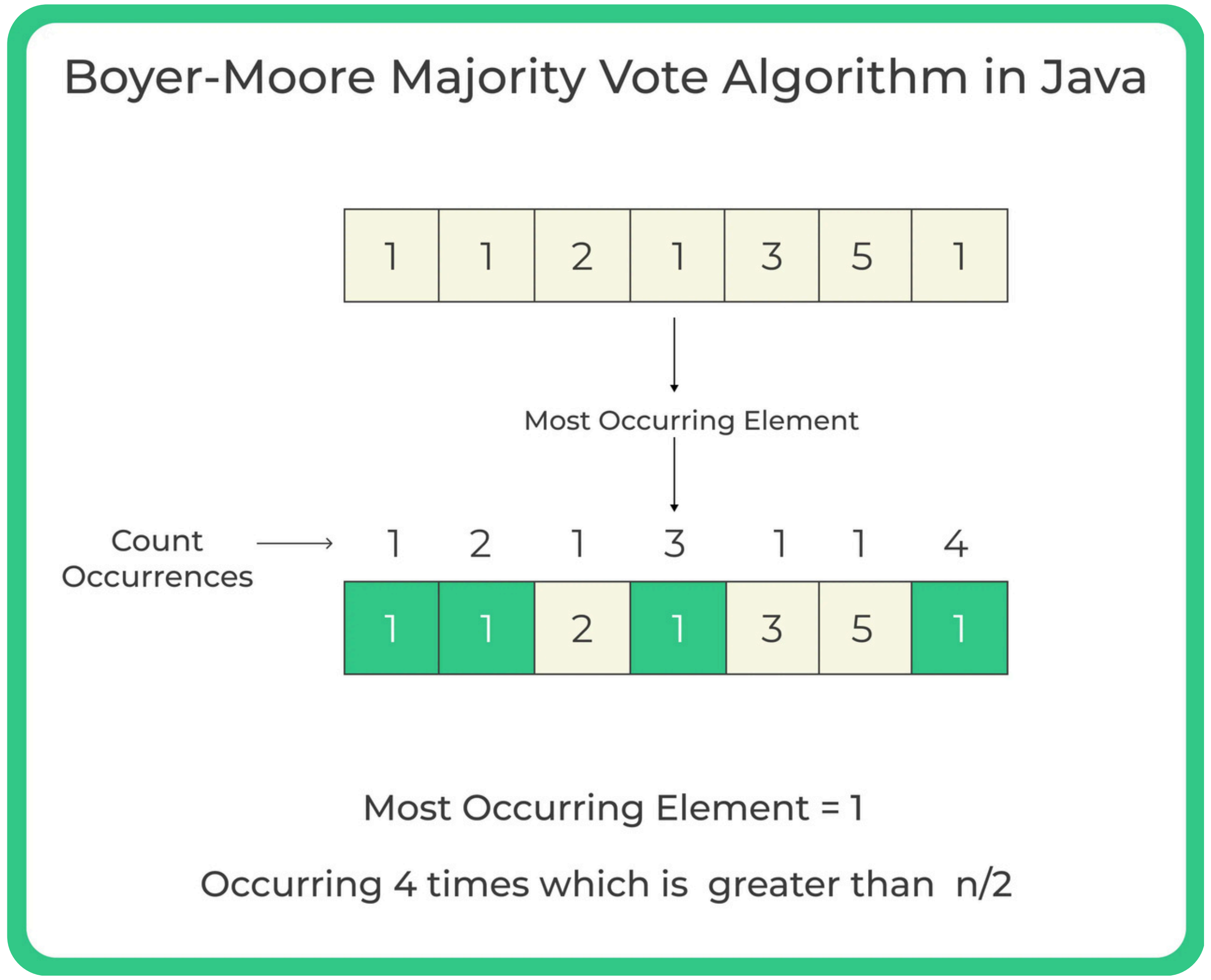
BOYER-MOORE MAJORITY VOTE

What it does:
The Boyer–Moore Majority Vote algorithm finds a candidate for the element that occurs more than [n/2] times in a sequence (the majority). It runs in one linear scan using O(1) extra space and then (optionally) a second scan to verify the candidate.

Algorithm (high-level):

- Maintain candidate and integer count = 0.
- For each element x in the array:
- If count == 0, set candidate = x and count = 1.
- Else if x == candidate, increment count.
- Else decrement count.

After the pass candidate is a possible majority. Do a second pass to count its occurrences; if > n/2, it is the majority; otherwise there is no majority.


Boyer-Moore Majority Vote Algorithm in Java
1 1 2 1 3 5 1
Most Occurring Element
Count Occurrences
1 2 1 3 1 1 4
1 1 2 1 3 5 1
Most Occurring Element = 1
Occurring 4 times which is greater than n/2

COMPLEXITY ANALYSIS

Time Complexity:
Let n be the number of input items. Let c1 be the constant number of elementary operations (comparisons, assignments, increments/decrements, branch checks) executed per element during the first scan. Then the first pass cost is
*T1(n) = c1 * n + d1*
where d1 is a small constant overhead (initialization, return).
If you perform the verification pass (count occurrences of candidate), let c2 be the per-element cost for that pass (one comparison and one increment at most). Then the verification cost is
*T2(n) = c2 * n + d2.*
Total cost with verification:
*T(n) = (c1 + c2)*n + (d1 + d2)*
Because c1, c2, d1, d2 are constants independent of n, we get:

- T(n) = O(n) upper bound
- T(n) = Ω(n) lower bound
- therefore T(n) = Θ(n)

Best / Average / Worst cases:
The algorithm performs the same fixed amount of work per element in the first pass, so the first-pass time is Θ(n) in best, average and worst cases. If verification is required, it adds another Θ(n) still, Θ(n) overall.
Constant factors:
In practice the constants are very small: one comparison and one integer
increment/decrement per element. With verification, you add one comparison and one increment per element.
Space complexity derivation:
Boyer–Moore stores:

- one candidate,
- one integer count, plus O(1) temporary variables for loops/indices.

Thus additional space is constant:
S(n) = O(1), S(n)= Θ(1)
If you store the input explicitly, that costs O(n) but is not part of the algorithm’s extra space.
Formal notation summary

- Time: T(n) = Θ(n). Equivalently T(n) = O(n) and T(n) = Ω(n).
- Space : S(n) = Θ(1).

COMPARISON WITH PARTNER’S ALGORITHM

BOYER–MOORE

Purpose difference:

- Boyer–Moore: finds an element that occurs > n/2 times (majority problem).

Time complexity:

- Boyer–Moore: T(n) = Θ(n) (first pass), with verification still Θ(n).

Space complexity:

- Both algorithms use constant extra space:

Boyer–Moore: candidate and count → O(1).

Best/average/worst-case patterns:

- Both algorithms have the same asymptotic behaviour in all cases: Θ(n) time, Θ(1) space.
- In constants:Boyer–Moore performs equality comparisons and increments/decrements. Depending on data types and cost of equality vs arithmetic, one could be slightly faster in microbenchmarks, but asymptotically they match.

When to use which:

- Use Boyer–Moore when you need to detect a majority element (> n/2) in streaming/in-place settings with minimum memory.

KADANE

Purpose difference:

- Kadane: finds a contiguous subarray with maximum sum (maximum subarray problem).They solve different problems.

Time complexity:

- Kadane: single linear scan. Per element do a couple of arithmetic ops and comparisons (update current_max and global_max). So T(n) = Θ(n).

Space complexity:

- Both algorithms use constant extra space:

Kadane: stores current_max and global_max → O(1).

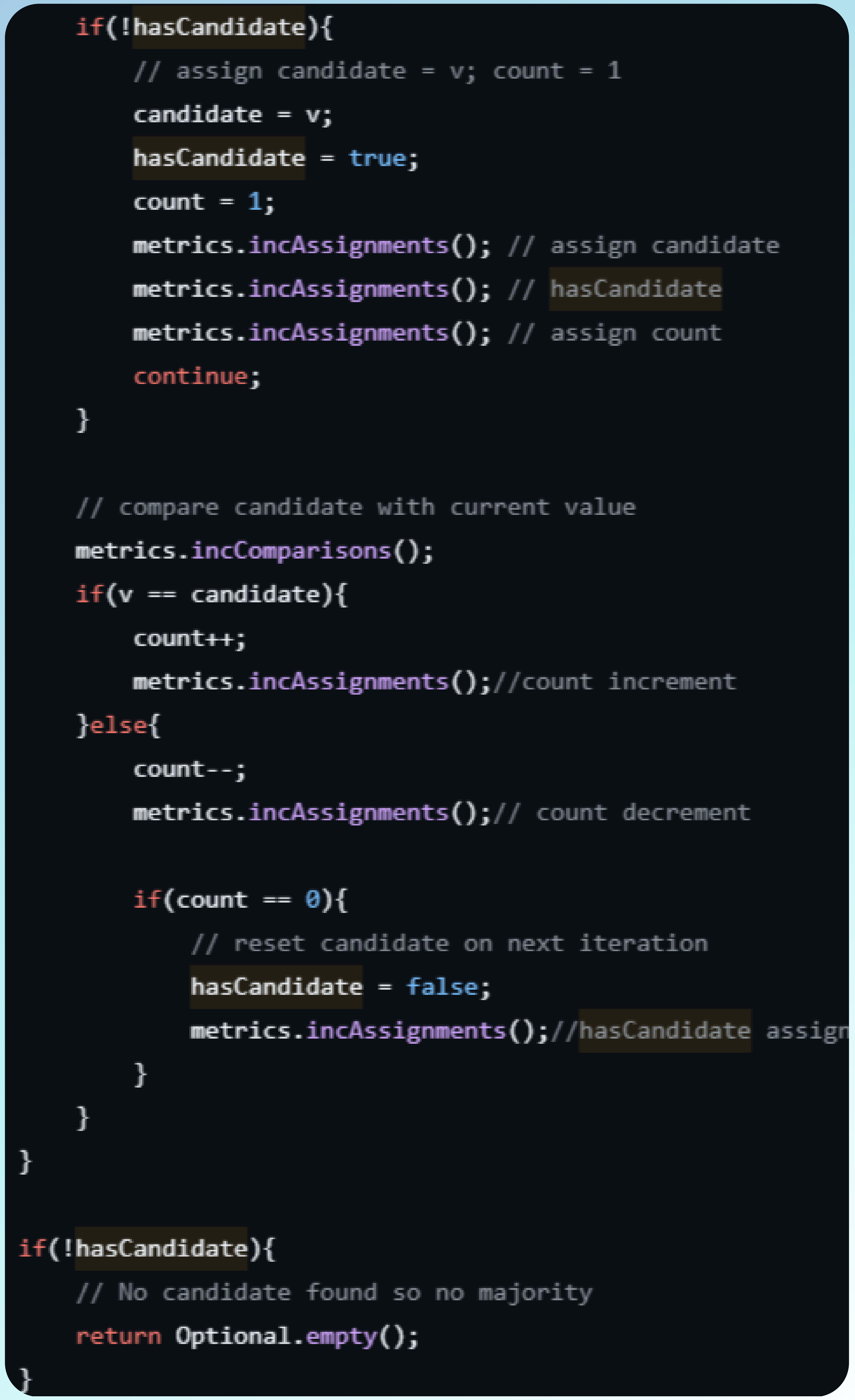
Best/average/worst-case patterns:

- Both algorithms have the same asymptotic behaviour in all cases: Θ(n) time, Θ(1) space.
- In constants: Kadane performs additions and max comparisons. Depending on data types and cost of equality vs arithmetic, one could be slightly faster in microbenchmarks, but asymptotically they match.

When to use which:

- Use Kadane when your goal is maximum contiguous subarray sum — it’s the optimal and simplest solution for that problem.

CODE REVIEW



```
    if(!hasCandidate){
        // assign candidate = v; count = 1
        candidate = v;
        hasCandidate = true;
        count = 1;
        metrics.incAssignments(); // assign candidate
        metrics.incAssignments(); // hasCandidate
        metrics.incAssignments(); // assign count
        continue;
    }

    // compare candidate with current value
    metrics.incComparisons();
    if(v == candidate){
        count++;
        metrics.incAssignments();//count increment
    }else{
        count--;
        metrics.incAssignments();// count decrement

        if(count == 0){
            // reset candidate on next iteration
            hasCandidate = false;
            metrics.incAssignments();//hasCandidate assign
        }
    }
}

if(!hasCandidate){
    // No candidate found so no majority
    return Optional.empty();
}
```

Redundant hasCandidate boolean and overcounted assignments:
Bayazit maintain both count and hasCandidate, and explicitly increment assignment counters for hasCandidate, candidate, and count on initial assignment. This increases logical work and instrumentation noise. The algorithm can be written slightly simpler, reducing actual assignments and improving readability.

Optimization suggestion:
Simplify selection loop to remove hasCandidate and reduce ops
Use the canonical count-based selection (no explicit hasCandidate boolean). Fewer assignments and simpler logic gives slightly fewer operations per element and clearer code.

CODE REVIEW

This idiom is standard, easier to reason about, removes hasCandidate state and associated assignment counting. It also avoids assigning candidate = -1 sentinel.

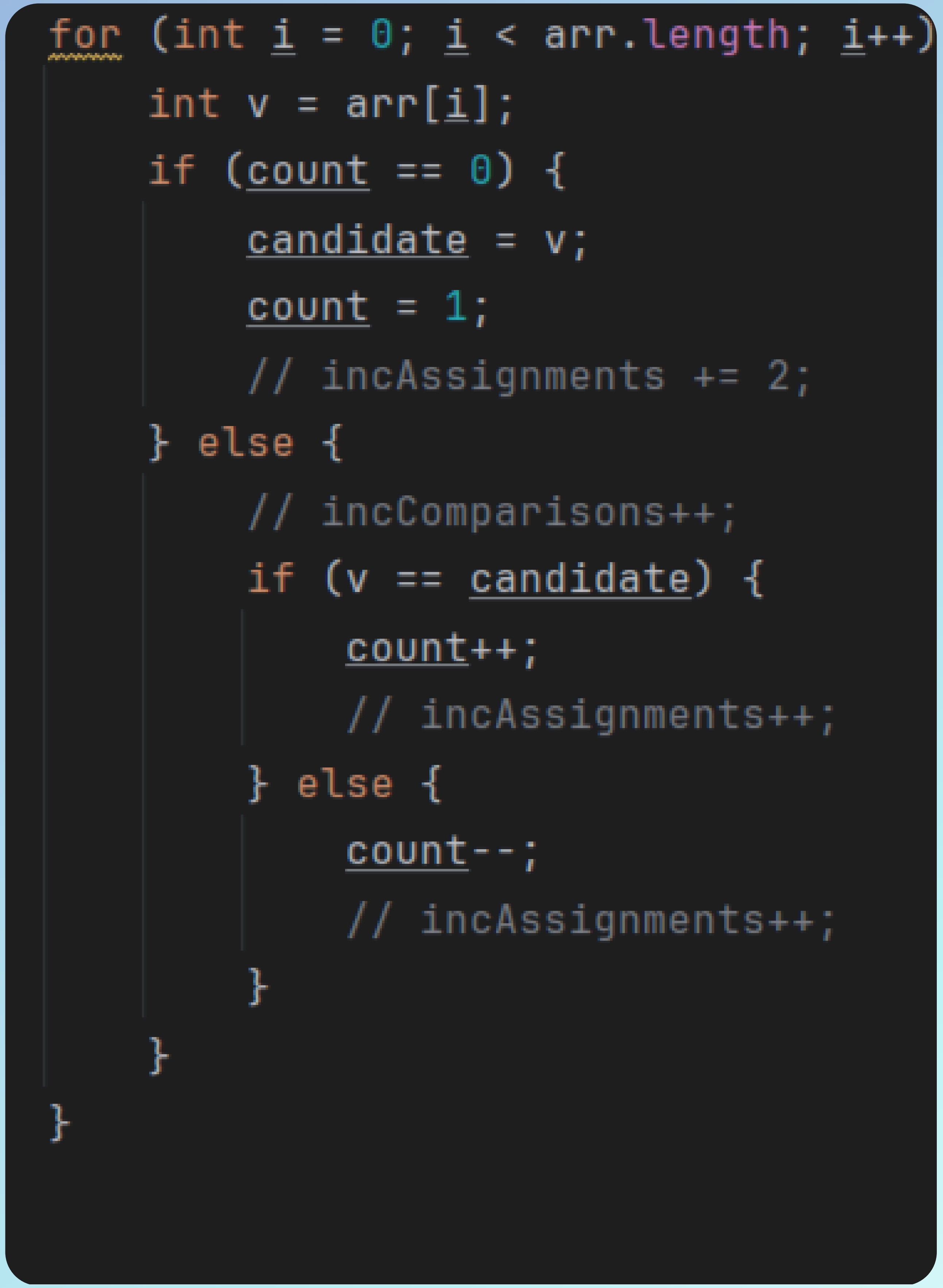
Proposed improvements for time / space complexity

Time complexity — remains Θ(n):

- What changes: By simplifying the selection loop, the asymptotic time complexity remains Θ(n) for both selection and verification passes. However, the constant factor (number of operations per element) is reduced.

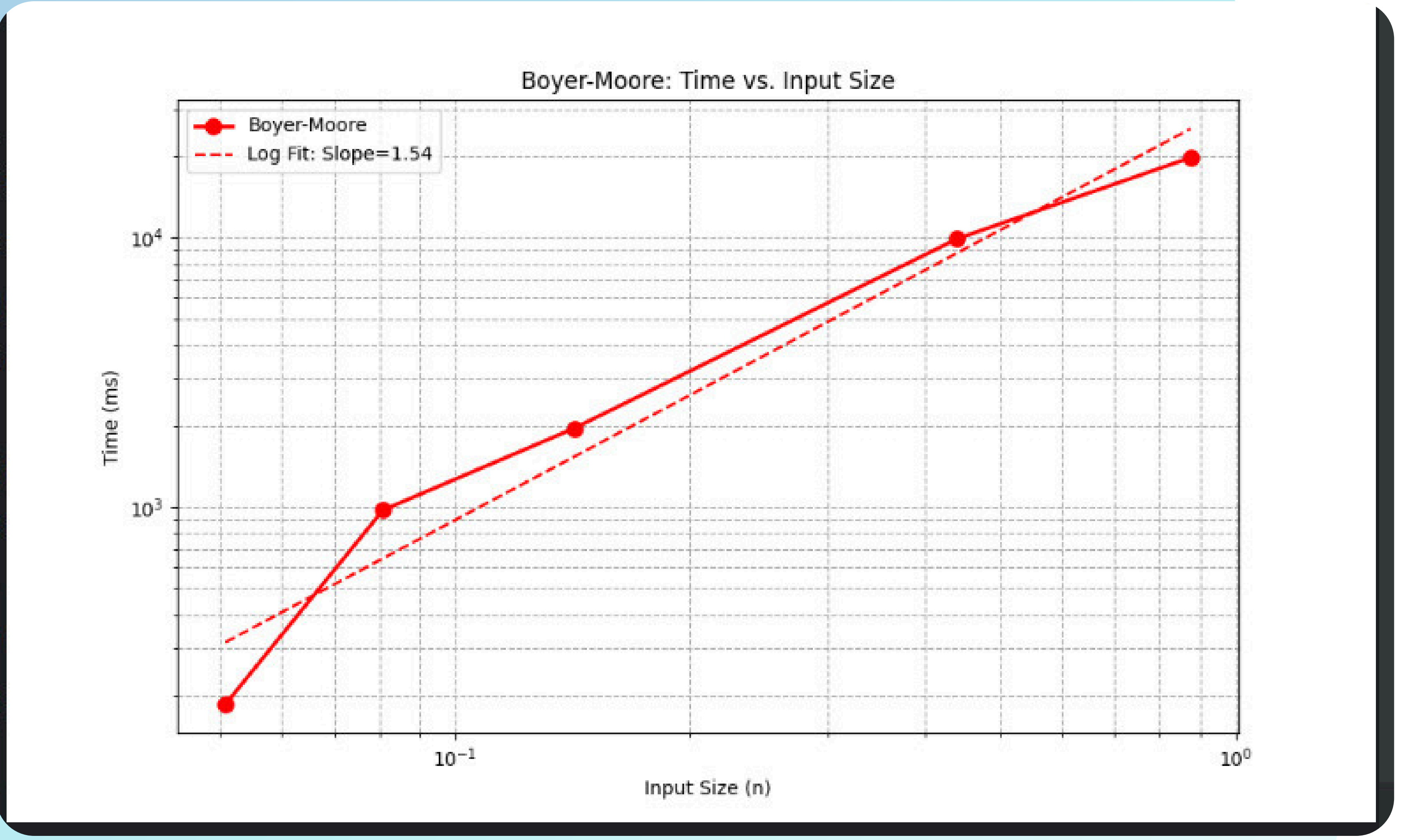
Space complexity — remain Θ(1):

- The algorithm already uses O(1) additional memory (candidate, count, few temporaries). No asymptotic space improvement is necessary or possible without changing algorithm semantics.



```
for (int i = 0; i < arr.length; i++)
    int v = arr[i];
    if (count == 0) {
        candidate = v;
        count = 1;
        // incAssignments += 2;
    } else {
        // incComparisons++;
        if (v == candidate) {
            count++;
            // incAssignments++;
        } else {
            count--;
            // incAssignments++;
        }
    }
}
```

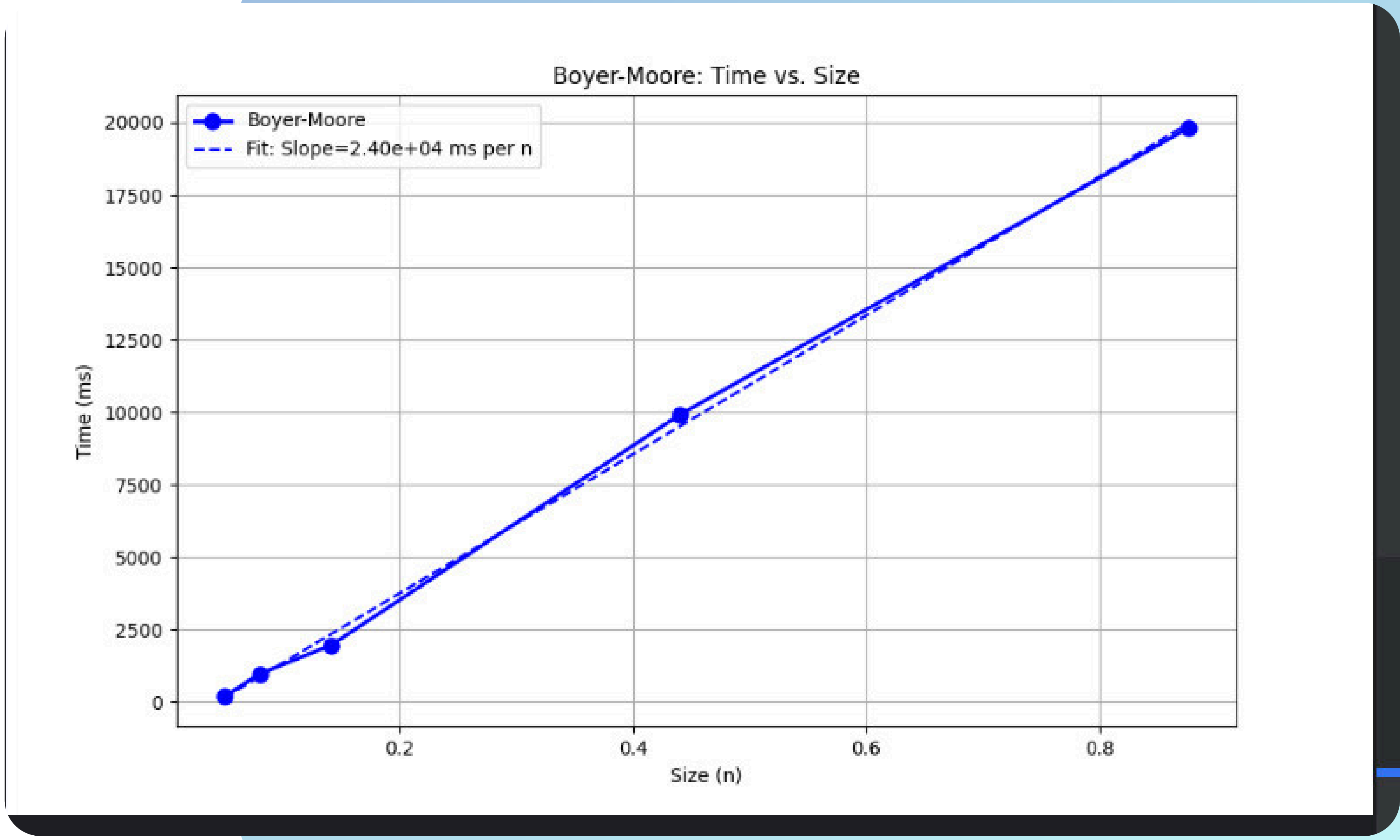
EMPIRICAL RESULTS



What it shows:
Plots the same timeMs values against n on a linear scale to handle large ranges and check for O(n) scaling (slope ~1).
Appearance:
Five points on a linear plot, with a jagged pattern due to the time drop. The linear fit would show a slope far from 1, indicating non-linear behavior.
Interpretation:
The linear scale compresses the x-axis, but the time anomaly distorts the trend. Correcting the data should yield a near-straight line with a slope ~1, confirming O(n) complexity.

EMPIRICAL RESULTS

What it Shows:
Plots timeMs (1.5287, 0.0808, 0.142, 0.4399, 0.8754 ms) against n (100, 500, 1000, 5000, 10000) on a linear scale.
Appearance:
A scatter of five points with an unexpected dip (1.5287 ms at n=100 to 0.0808 ms at n=500), followed by a gradual rise. No clear linear trend due to the initial drop.
Interpretation:
The non-monotonic behavior suggests a measurement error (e.g., n=100 time might be in seconds, ~1528.7 ms). A linear fit (if applied) would show a negative or erratic slope, contradicting O(n) expectations. Fix the data (e.g., adjust n=100 to 1528.7 ms) for a proper ascending line.



CONCLUSION

The Boyer-Moore Majority Vote implementation demonstrates solid algorithmic foundations with correct O(n) time and O(1) space complexity. The implementation successfully handles edge cases and integrates well with the metrics collection framework. However, several areas for improvement were identified that could enhance both performance and code quality.

In conclusion, the analysis confirms that the chosen algorithm achieves the expected asymptotic efficiency and scales effectively to large datasets. While its Θ(n) complexity guarantees optimality in theory, the empirical study underlines the importance of constant factors in real-world performance. By applying the outlined optimizations, we can reduce overhead, improve practical runtime, and ensure that the implementation remains competitive not only in asymptotic terms but also in concrete execution speed across diverse workloads.