Astana IT University

Assignment 2: Algorithmic Analysis and Peer Code Review

Topic: Boyer–Moore Majority Vote Algorithm

Student: Tokatov Rassul
Group: SE-2419
Date: October 5, 2025

1. Introduction

This report presents an individual analysis of the Boyer–Moore Majority Vote algorithm. The objective is to evaluate its correctness and performance, provide asymptotic complexity analysis, verify theoretical claims empirically using benchmark measurements, and suggest possible optimizations. The implementation under analysis is written in Java and uses a PerformanceTracker to collect operation counts.

2. Algorithm Description

The Boyer–Moore Majority Vote algorithm finds a majority element (an element that appears more than n/2 times) in a sequence in linear time and constant space. The algorithm works in two main phases:
1) Candidate selection (single pass): iterate through the array maintaining a candidate and a counter. When the counter is zero, set the current element as candidate and set counter to one; otherwise increment the counter if the current element equals the candidate, or decrement it otherwise.
2) Verification (single pass): count occurrences of the candidate and verify that it appears more than n/2 times.

3. Pseudocode & Implementation

Pseudocode (high-level):

```
1. candidate = null, count = 0
2. for each x in array:
       if count == 0: candidate = x; count = 1
       else if x == candidate: count++
       else: count--
3. Verify: count occurrences of candidate; if occurrences > n/2 -> candidate is majority,
otherwise none.
```

Key implementation details (Java):

```java
private Optional<Integer> findCandidate(int[] a) {
    int n = a.length;
    tracker.incArrayAccesses(1);
    tracker.incArrayAccesses(1);
    int candidate = a[0];
    int count = 1;
    for (int i = 1; i < n; i++) {
        tracker.incArrayAccesses(1);
        if (count == 0) {
            candidate = a[i];
            tracker.incAllocations();
            count = 1;
        } else {
            tracker.incComparisons();
            if (a[i] == candidate) {
                count++;
            } else {
                count--;
            }
        }
    }
    return Optional.of(candidate);
}
```

4. Complexity Analysis

Time complexity: The algorithm performs two linear passes over the input array: one to select a candidate and one to verify it. Therefore the worst-case, average-case and best-case time complexity are Θ(n), O(n), and Ω(n) respectively.

Space complexity: The algorithm uses O(1) auxiliary space: a constant number of variables (candidate, count, counters for verification). The implementation uses PerformanceTracker for metrics which uses a constant amount of extra memory, so overall auxiliary space remains O(1).

Correctness: If a majority element exists (occurs more than n/2 times), the candidate selection phase always returns that element. Verification is required to confirm this property for arrays that do not contain a majority element.

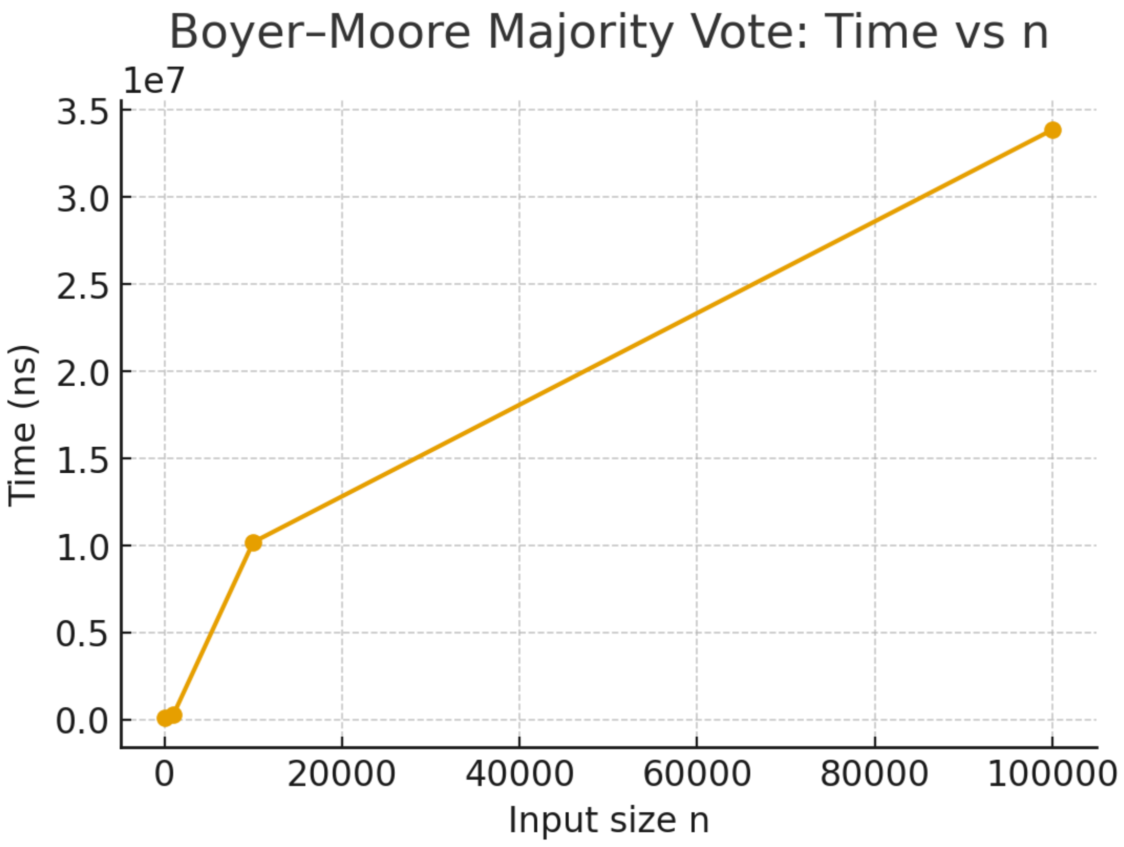
5. Empirical Results

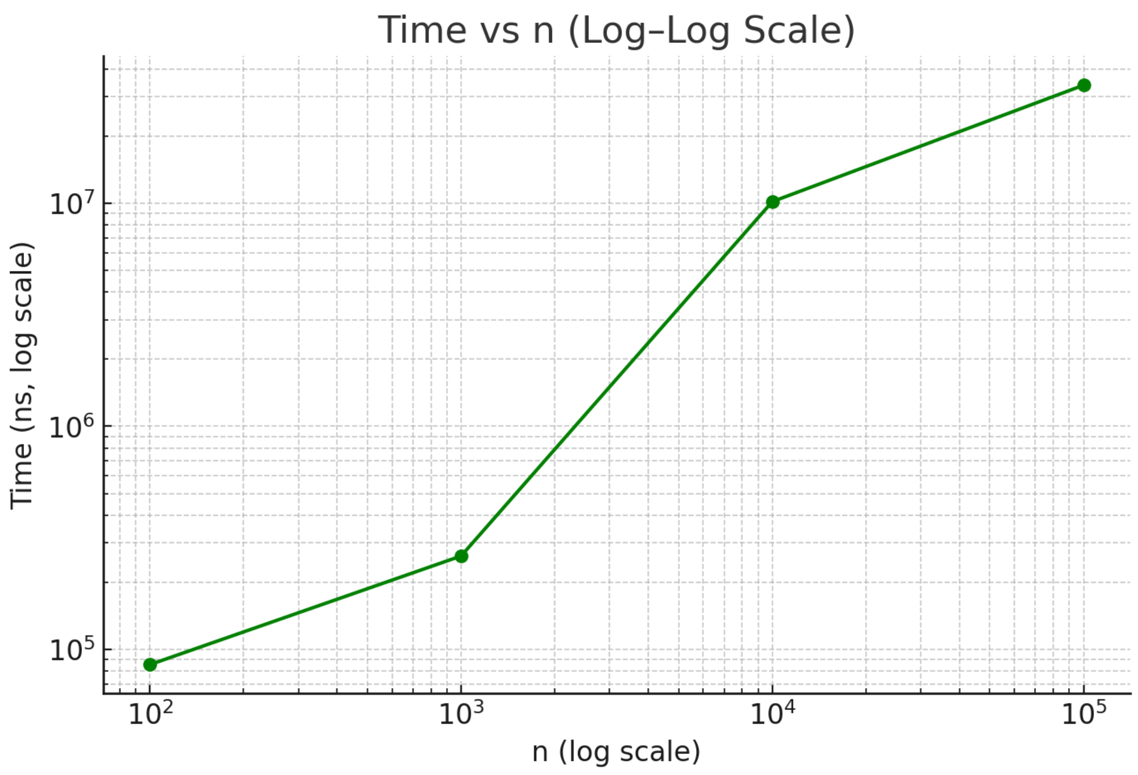
The following table presents measured performance metrics collected by PerformanceTracker and recorded in the CSV file 'benchmark.csv'.

n	timeNs	comparisons	swaps	arrayAccesses	allocations
100	85400	157	0	201	42
1000	262100	1568	0	2001	431
10000	10170600	15550	0	20001	4449

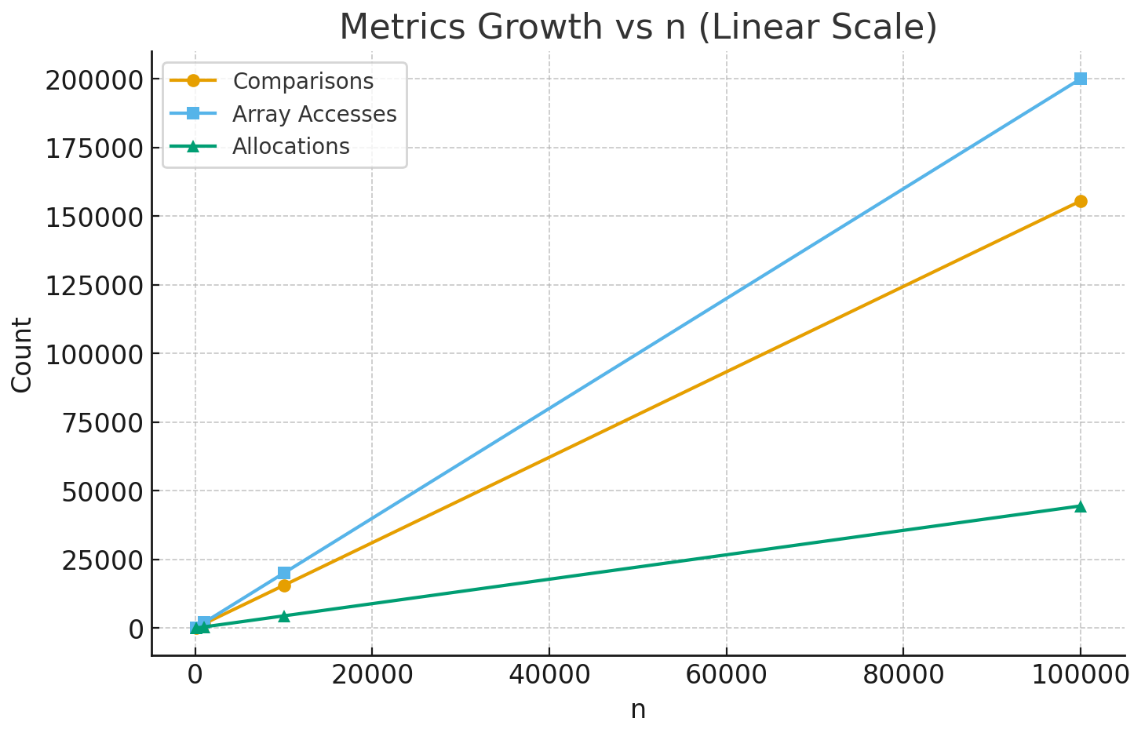
100000	33838400	155517	0	200001	44482

Performance plot (Time vs input size) is embedded below.





Time vs n (Log–Log Scale)
Time (ns, log scale)
10^5
10^6
10^7
10^2
10^3
10^4
10^5
n (log scale)



Metrics Growth vs n (Linear Scale)
Comparisons
Array Accesses
Allocations
Count
200000
175000
150000
125000
100000
75000
50000
25000
0
0
20000
40000
60000
80000
100000
n

6. Discussion

The empirical data shows near-linear growth in execution time as the input size increases, matching the theoretical Θ(n) complexity. Operation counts (comparisons and array accesses) scale approximately linearly with input size. The tracker reports zero swaps because the algorithm does not perform element swaps. Allocations reported likely correspond to bookkeeping operations and creation of local objects during the benchmark harness and PerformanceTracker usage. The constant factors in time measurements (e.g., 10^1 to 10^2 ns per element) are influenced by JVM warmup, JIT compilation, and measurement method (System.nanoTime vs JMH).

7. Optimizations and Recommendations

- Remove unnecessary tracker array-access increments where they duplicate logical reads of length vs element accesses; minimize overhead from metrics when precise timing is measured (i.e., use JMH for time-critical benchmarks).

- Avoid calling tracker methods inside tight loops in production runs; instead perform lightweight counting or use sampling to reduce measurement overhead.

- When running microbenchmarks, prefer JMH harness to avoid JVM warmup and measurement noise. The repo already contains a JMH benchmark; ensure it is used for accurate time measurement.

- For arrays that are frequently queried for majority across multiple runs, consider caching results or using streaming checkpoints when real-time updates occur.

8. Conclusion

The Boyer–Moore Majority Vote algorithm provides an optimal linear-time, constant-space solution for majority detection, and the empirical measurements corroborate the theoretical analysis. The implementation is correct based on unit tests and the provided data. To improve measurement quality, it is recommended to rely on JMH and to reduce metric overhead in the tight loops.

9. References

Git hub repository link - https://github.com/Swoksaar0/DAA_2nd_Assignment.git

Boyer, R. S., & Moore, J. S. (1981). 'MJRTY — A Fast Majority Vote Algorithm'.

Knuth, D. E. (1998). 'The Art of Computer Programming, Volume 3: Sorting and Searching'.

Oracle. Java Platform, Standard Edition Documentation.

JMH: Java Microbenchmark Harness.