Astana IT University

Design and Analysis of Algorithms

Tyulebayeva Arailym SE-2401

1. Algorithm Overview

Student A: Diana
Algorithm: Boyer-Moore Majority Vote

The Boyer-Moore Majority Vote algorithm is a linear-time solution to the *majority element problem* - identifying the element that appears more than ⌊n/2⌋ times in an array. It operates in a **single pass (O(n))**, making it highly efficient for large datasets.

The algorithm maintains a **candidate** and a **count** variable. As it iterates through the array, it adjusts the candidate and increments or decrements the count depending on whether the current element matches the candidate. After one pass, the candidate is guaranteed to be the majority element if one exists.

This approach is both **time- and space-efficient**, requiring only constant additional memory (O(1)) and a single linear scan.

2. Complexity Analysis

2.1 Theoretical Overview

The Boyer–Moore Majority Vote algorithm identifies the majority element — one appearing more than ⌊n/2⌋ times — in a single linear pass. It works by maintaining a candidate and a counter. When the current element matches the candidate, the counter is incremented; otherwise, it is decremented. If the counter reaches zero, the current element becomes the new candidate.

The strength of this approach lies in its simplicity and optimal efficiency. It requires no additional data structures and avoids expensive operations such as sorting or hashing. As a result, it achieves linear time and constant space performance.

2.2 Time Complexity

The algorithm performs a fixed set of operations per element: a comparison, an increment or decrement, and an occasional reassignment of the candidate. Regardless of input distribution, it always processes each element once.

- Best Case (Θ(n)): When the majority element dominates early, the counter rarely resets, but the algorithm still scans the entire array.
- Average Case (O(n)): The candidate may change several times, but the total operations per element remain constant.
- Worst Case (O(n)): Even if there is no majority or it appears late, the algorithm still performs one pass.

Thus, in all cases:

$$T(n) = c \cdot n = O(n)$$

This is the most efficient possible time complexity since every element must be examined at least once.

2.3 Space Complexity

Boyer–Moore maintains only two variables — a candidate and a counter — regardless of input size. It does not require extra memory for data structures or recursion.

$$S(n) = O(1)$$

This makes it especially suitable for memory-constrained environments and large streaming datasets.

2.4 Comparison with Other Approaches

Algorithm	Time Complexity	Space Complexity	Notes
Sorting + Scan	$O(n\log n)$	$O(1)$	Requires sorting
Hash Map Counting	$O(n)$	$O(n)$	Extra memory usage
Boyer–Moore	$O(n)$	$O(1)$	Most efficient overall

Boyer–Moore is the only algorithm that achieves both linear time and constant space simultaneously.

2.5 Recurrence (Conceptual)

Although iterative, the algorithm's recurrence relation can be expressed as:

$$T(n) = T(n-1) + O(1)$$

Solving this gives:

$$T(n) = O(n)$$

2.6 Summary

Case	Time Complexity	Space Complexity
Best	$\Theta(n)$	$O(1)$
Average	$O(n)$	$O(1)$
Worst	$O(n)$	$O(1)$

2.7 Interpretation

Boyer–Moore achieves optimal linear performance in all cases and uses constant space, outperforming common alternatives. Its predictable behavior and minimal memory usage make it ideal for large datasets and real-time applications.

3. Code Review

3.1. Strengths & Observations

Diana's implementation of the Boyer–Moore Majority Vote algorithm follows the fundamental logic of the algorithm correctly and demonstrates a solid understanding of its principles. The implementation performs the required linear-time majority detection in a single pass and maintains constant auxiliary space throughout execution.

Other notable strengths include:

Metrics tracking: The code collects detailed performance data such as comparisons and array accesses, which aligns with the assignment's empirical validation requirements.

Benchmark integration: A dedicated benchmark runner and CSV export functionality are included, facilitating performance analysis on arrays of varying sizes.

Clean algorithm structure: The main logic is divided into clearly defined steps — candidate selection and candidate verification — improving readability and conceptual clarity.

3.2. Issues & Improvement Suggestions

Despite these strengths, several aspects of the implementation can be improved to enhance correctness, robustness, and efficiency.

1. Lack of Candidate Verification Robustness

While the algorithm correctly identifies a candidate in a single pass, it initially assumes that a majority element always exists. This is not guaranteed for arbitrary input arrays. It is essential to verify the candidate in a second pass by counting its occurrences and ensuring that it appears more than n/2 times:

```
int count = 0;
for (int num : arr) {
    if (num == candidate) count++;
}
if (count <= arr.length / 2) {
    throw new NoSuchElementException("No majority element found");
}
```

2. Limited Error Handling

The original implementation throws an exception only if the array is empty, but it should also handle the case where no majority element is found. Throwing a NoSuchElementException or returning an Optional<Integer> would improve robustness and clarify algorithm behavior in such scenarios.

3. Metrics Overhead vs. Accuracy

While performance tracking is valuable, recording metrics inside tight loops can slightly skew benchmark results. A better approach is to minimize metrics updates or handle them outside of performance-critical sections to ensure that measurements more accurately reflect the true runtime.

4. Code Structure and Modularization

The algorithm could benefit from splitting core logic into smaller, well-defined methods, such as:

findCandidate(int[] arr)

verifyCandidate(int[] arr, int candidate)

runWithMetrics(int[] arr, PerformanceTracker tracker)

This would improve readability, simplify testing, and make future maintenance easier.

3.3. Optimization Impact & Justification

Optimization Description The initial implementation of the Boyer–Moore Majority Vote algorithm correctly identified the majority element using two passes: the candidate selection phase and the verification phase.

However, it performed unnecessary operations that could be optimized to improve performance. Original approach: • In the findCandidate() method, comparisons and assignments were performed for every element, even when they were redundant. • The checkCandidate() method always iterated through the entire array, even if the result was already mathematically determined before the end of the scan. Optimized approach: The updated version introduces two key improvements:

1. Simplified candidate selection loop:

• The candidate detection phase was rewritten using the enhanced for loop (for (int j : arr)), making the code cleaner and slightly reducing loop overhead.

• Logic was improved to avoid unnecessary operations when count is zero (now directly sets both candidate and count). 2. Early termination in verification:

• A significant optimization was introduced in the checkCandidate() phase.

• Instead of always scanning the entire array, the algorithm now checks whether it is mathematically possible for the candidate to still be a majority.

• If the remaining elements plus the current count cannot exceed n/2, the loop terminates early.

• This reduces the number of comparisons, especially in cases where the majority element is absent or appears early in the array.

4. Empirical Results

4.1 Experimental Setup

To validate the theoretical complexity analysis of Diana's Boyer–Moore Majority Vote algorithm, a series of empirical experiments were conducted. The algorithm was tested on randomly generated integer arrays of sizes ranging from 100 to 100,000. Each run was performed on the same machine under similar conditions to ensure consistent benchmarking results.
The following performance metrics were collected during execution:

- Comparisons: Number of element-to-candidate comparisons performed.
- Array Accesses: Number of times elements were accessed.
- Memory Allocations: Additional memory allocated beyond the input array.
- Execution Time: Total runtime measured in milliseconds.

These metrics were recorded and exported to CSV files to enable further analysis and visualization.

4.2 Performance Results

The table below shows the observed performance characteristics of the algorithm on increasing input sizes:

Input Size (n)	Comparisons	Array Accesses	Memory Allocations	Execution Time (ms)
100	200	200	0	0.033459
1,000	2,000	2,000	0	0.295375
10,000	20,000	20,000	0	1.843959
100,000	200,000	200,000	0	11.855916

4.3 Analysis of Results

The empirical results strongly confirm the theoretical predictions of the Boyer–Moore Majority Vote algorithm:

- Linear Time Complexity: Both the number of comparisons and array accesses scale linearly with input size. For example, when n increases by a factor of 10, comparisons and accesses also increase by approximately 10×. This confirms the $O(n)$runtime complexity derived analytically.

- Constant Space Complexity: Memory allocations remain constant (0 in all runs), verifying the $O(1)$auxiliary space usage. This is expected since the algorithm uses only a small number of variables regardless of input size.
- Execution Time Scalability: Execution time grows proportionally with the input size. For example, increasing n from 10,000 to 100,000 results in execution time increasing from ~1.84 ms to ~11.85 ms (~6.4×), which is consistent with linear behavior.

4.4 Validation of Complexity and Practical Performance

The measured data aligns precisely with the theoretical analysis:

- Time Complexity: $O(n)$— validated empirically by linear scaling of comparisons, accesses, and runtime.
- Space Complexity: $O(1)$— validated by zero additional memory usage.

Furthermore, the negligible overhead observed for metrics collection demonstrates that the implementation remains efficient in real-world conditions. The constant factors are small, and the performance is suitable even for large-scale arrays (e.g., 100,000 elements processed in ~11.8 ms).

4.5 Summary

The experimental results confirm that Diana's implementation of the Boyer–Moore Majority Vote algorithm achieves the expected linear performance with minimal memory usage. The algorithm scales predictably across input sizes and demonstrates strong alignment between theoretical analysis and real-world execution behavior.

5. Conclusion

The analysis of Diana's implementation of the **Boyer–Moore Majority Vote algorithm** demonstrates that it successfully meets the fundamental requirements of efficient linear-time majority detection. The algorithm achieves $O(n)$time complexity and $O(1)$space complexity, making it optimal for this problem domain. Empirical testing confirms the theoretical predictions: the number of comparisons and array accesses grows linearly with input size, while memory usage remains constant. These results verify both the correctness and scalability of the solution.

Despite these strengths, several areas for improvement were identified during the peer review. Most notably, the implementation assumes that a majority element always exists, and it does not verify the candidate after the initial pass. Adding a verification step and exception handling would make the algorithm more robust for all possible input scenarios. Furthermore, minor code-level optimizations—such as reducing branching and modularizing logic into smaller methods—could improve maintainability and potentially reduce constant factors in execution time.

The empirical benchmarks reinforce that the Boyer–Moore algorithm is not only theoretically efficient but also performs exceptionally well in practice, even on large inputs. With the proposed enhancements, the solution would reach production-level robustness and readability while preserving its optimal asymptotic properties. This exercise highlights the importance of combining theoretical understanding, careful code design, and empirical validation in algorithmic development.