**Individual Analysis Report: Boyer-Moore Majority Vote**

**Project Author: Sultanbek Mukashev  
Partner and report writer: Artyom Karmykov**

**Group: SE-2422  
Pair: 3  
Project: Assignment 2 — Algorithmic Analysis and Peer Code Review  
Date: 05.10.2025**

**Algorithm Overview**

Sultanbek’s project implements the problem of finding the "majority element" in an array. Given an array of integers of length , the goal is to determine if there is an element that occurs strictly more than  times, and to return it (or indicate absence if it doesn't exist).

The solution is based on the Boyer-Moore Majority Vote algorithm, which works in two passes:

1. The first pass (candidate selection) maintains a candidate and a counter. If the current element equals the candidate, the counter is incremented; otherwise, it's decremented. When the counter reaches zero, the candidate is updated to the current element.
2. The second pass verifies that the found candidate occurs more than  times in the array.

This method efficiently finds the majority element (if it exists) in just two linear scans of the array, using constant extra memory.

**Complexity Analysis**

**Time Complexity:**

* Each of the two scans is strictly .
* The first pass involves at most  comparisons and counter updates.
* The second pass makes  comparisons to count occurrences.
* Total:
* Lower bound : Less than  reads are impossible in the worst case.
* Therefore,

for all best/average/worst cases.

**Space Complexity:**

* The algorithm maintains only two variables (candidate, counter), independent of .
* Thus,

**Comparison to Alternatives:**

* Hash map:  time and  space (worse memory).
* Sorting:  time (slower for large ), memory cost depends on sorting algorithm.
* Other approaches are either more complex or less optimal.

For the majority element problem, Boyer-Moore provides both optimal time and space guarantees.

**Code Review**

**Strengths:**

* Implements the textbook two-pass logic with no unnecessary complications.
* Clear candidate selection loop.
* Handles edge cases and validates input.
* Avoids creating auxiliary data structures, respecting  space.

**Potential inefficiencies and suggestions:**

1. Code reuse: Candidate counting logic could be factored out, making further testing and extensions easier.
2. Early returns: Handle arrays of length 0 or 1 immediately before scanning.
3. Type optimization: Consider use of OptionalInt instead of Integer to avoid boxing/unboxing.
4. Testing: Expand test suite to include negative values, edge cases, and randomized arrays.
5. Benchmarking: Results are conveniently stored in CSV format and visualized for analysis.

**Opportunities for optimization:**  
Theoretical time and space are already optimal. Minor improvements are micro-optimization: reducing unnecessary comparisons, slightly restructuring branching, or speeding up random array generation for synthetic tests.

**Empirical Results**

The benchmark plot and table below match the expected linear scaling of execution time with respect to input size:

| Label | n | best\_ms | avg\_ms | worst\_ms | result |
| --- | --- | --- | --- | --- | --- |
| Comprehensive\_NoMajority\_n1000 | 1,000 | 0.044 | 0.045 | 0.053 |  |
| Comprehensive\_WithMajority\_n1000 | 1,000 | 0.046 | 0.047 | 0.053 | 320 |
| Comprehensive\_NoMajority\_n200800 | 200,800 | 0.162 | 0.667 | 1.607 |  |
| Comprehensive\_WithMajority\_n200800 | 200,800 | 0.686 | 0.723 | 0.784 | -75 |
| Comprehensive\_NoMajority\_n400600 | 400,600 | 0.263 | 0.271 | 0.312 |  |
| Comprehensive\_WithMajority\_n400600 | 400,600 | 1.394 | 1.428 | 1.610 | 995 |
| Comprehensive\_NoMajority\_n600400 | 600,400 | 0.395 | 0.399 | 0.418 |  |
| Comprehensive\_WithMajority\_n600400 | 600,400 | 2.098 | 2.144 | 2.381 | 58 |
| Comprehensive\_NoMajority\_n800200 | 800,200 | 0.526 | 0.530 | 0.538 |  |
| Comprehensive\_WithMajority\_n800200 | 800,200 | 2.795 | 2.812 | 2.862 | 710 |
| Comprehensive\_NoMajority\_n1000000 | 1,000,000 | 0.657 | 0.673 | 0.696 |  |
| Comprehensive\_WithMajority\_n1000000 | 1,000,000 | 3.492 | 3.536 | 3.765 | 339 |

**Plot Analysis:**

**Изображение выглядит как текст, снимок экрана, линия, График

Содержимое, созданное искусственным интеллектом, может быть неверным.**

* Execution time grows linearly as array size increases.
* With a majority element present, absolute times are slightly higher, but the order of growth is the same.
* No drastic jumps or slowdowns are observed, confirming theoretical analysis.

**Practical observation:**  
Even at arrays up to 1,000,000 elements, the average run time is below 4 ms, which is extremely fast for a high-level language.

**Conclusion**

Sultanbek’s Boyer-Moore Majority Vote implementation lives up to both the theoretical and empirical standards:

* The algorithm exhibits strict linear time and constant space complexity.
* No serious bottlenecks or inefficiencies were found in the code.
* Recommended: Keep this implementation as the default for majority element problems.
* For even larger datasets or frequent online queries, consider parallelizing runs or storing count maps for further queries.
* Automatic CSV export and visualization guarantee transparent and reproducible performance analysis - excellent practice for engineering and research reporting.