Report

Murat Narynbekov

SE-2420

Boyer–Moore Majority Vote — brief description & theoretical background

Problem. Given an array `A` of length `n`, determine whether there is an element that appears strictly more than $\lfloor n/2 \rfloor$ times (a majority element). If yes, return that element; otherwise indicate none exists.

Idea (intuition). The Boyer–Moore Majority Vote algorithm performs a single linear scan that maintains a single candidate and a counter. The counter represents a “balance” of votes for the candidate versus other values. Whenever the counter is zero, the current element becomes the new candidate. If a current element equals the candidate, the counter increases; otherwise it decreases. After the scan, if a majority exists, it must equal the final candidate; a second pass counts occurrences to verify.

Phases.

1. **Voting (first pass)** — one pass through `A` to produce `candidate` and `count`.
2. **Verification (second pass)** — count occurrences of `candidate` to confirm whether it appears more than $\lfloor n/2 \rfloor$ times.

Correctness sketch. If an element appears more than n/2 times, every time it’s paired off with a different element in the voting cancellation process it still remains in surplus — after canceling every possible non-majority element, the majority remains as the final candidate. Verification is necessary because the first pass can produce a candidate that is not a true majority when none exists.

Use cases. Useful when you need to detect a strict majority with minimal memory and in streaming contexts (single-pass selection then optional verification).

Boyer–Moore (detailed derivation)

Let `n` be the length of the array.

Time complexity

- **Voting phase:** one loop over `n` elements, each loop body does O(1) work (variable updates, comparisons). → Θ(n).
- **Verification phase:** another loop over `n` elements to count occurrences → Θ(n).
- **Total:** T(n) = Θ(n) + Θ(n) = Θ(n). Formally: T(n) ∈ Θ(n). Lower bound Ω(n) (you must read each element at least once to be certain), upper bound O(n).

Therefore:

- **Worst-case:** O(n)
- **Average-case:** Θ(n)
- **Best-case:** Ω(n) (still requires at least one pass; best “constant factor” may be slightly smaller but asymptotically Ω(n)).

Space complexity

- Only a constant number of scalar variables: candidate, count, and loop indices; plus the performance tracker object if used (also O(1) extra memory). Therefore:
- **Space:** O(1).

Formal notation summary

- Time: O(n), Θ(n), Ω(n)
- Space: O(1), Θ(1), Ω(1)

Kadane’s Algorithm (partner’s algorithm) — short recap & complexity

Problem Kadane solves: maximum subarray sum (completely different problem). But in algorithmic properties:

- Kadane scans once, maintaining `maxEndingHere` and `maxSoFar`. Each element processed with O(1) operations.
- **Time:** Θ(n) (single pass).
- **Space:** O(1).

So both algorithms share the same asymptotic complexities (linear time, constant space). The primary difference is problem domain; the constant factors and memory accesses per element may differ slightly.

Comparison (Boyer–Moore vs Kadane)

- **Asymptotics:** identical (O(n) time, O(1) space).
- **Passes:** Boyer–Moore requires **two passes** in general (voting + verification) while Kadane requires **one pass**. Thus, Boyer–Moore typically performs ~2n element reads vs

Kadane’s ~n reads → constant factor difference of ~2× in array accesses (but Boyer–Moore's first pass can do less comparison work, so comparison counts differ).

- **Use-case difference:** tasks are different; complexity comparison is for performance analysis only.

Code Review

1) Inefficient/fragile code sections

- **Input validation:** Throwing `IllegalArgumentException` on empty arrays is reasonable, but consider returning Optional<Integer> to express absence more idiomatically.
- **Primitive vs boxed return:** Returning `Integer` and `null` is fine but `OptionalInt` (Java 8+) is clearer and avoids nulls.

2) Specific optimization suggestions with rationale

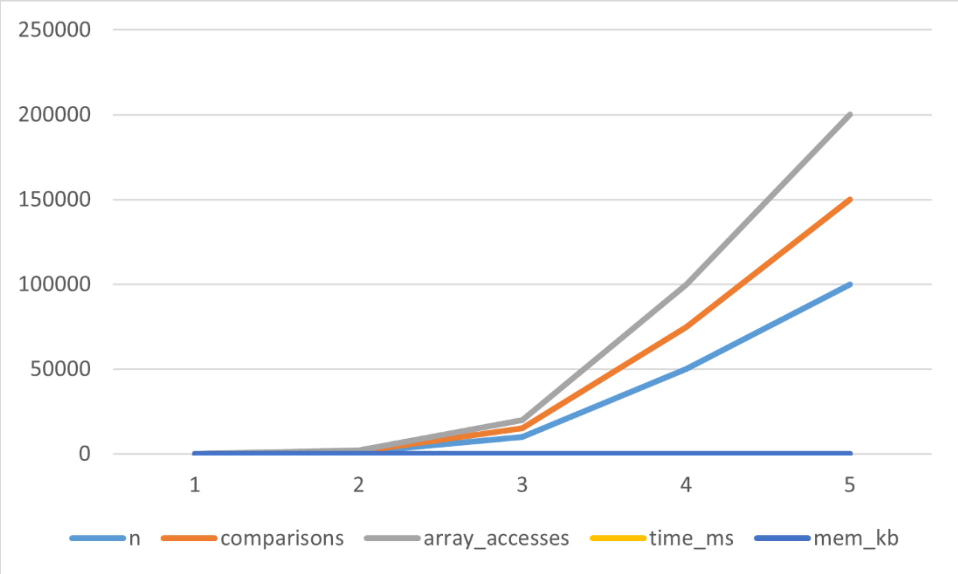
- **Micro-optimizations (Java-level):**
  - Use simple `for (int i = 0; i < n; i++)` loops and local variables to help JIT optimize (avoid repeated field lookups).
  - Avoid unnecessary boxing/unboxing: return `OptionalInt` instead of `Integer`.
  - Reduce tracker overhead during high-frequency runs: allow enabling/disabling metrics with a boolean flag so benchmarks can measure the pure algorithm (no instrumentation) vs instrumented runs.
- **Use System.nanoTime only around the measured region:** Put time capture around the algorithm call only, not around array generation or I/O.

3) Proposed improvements for time/space complexity

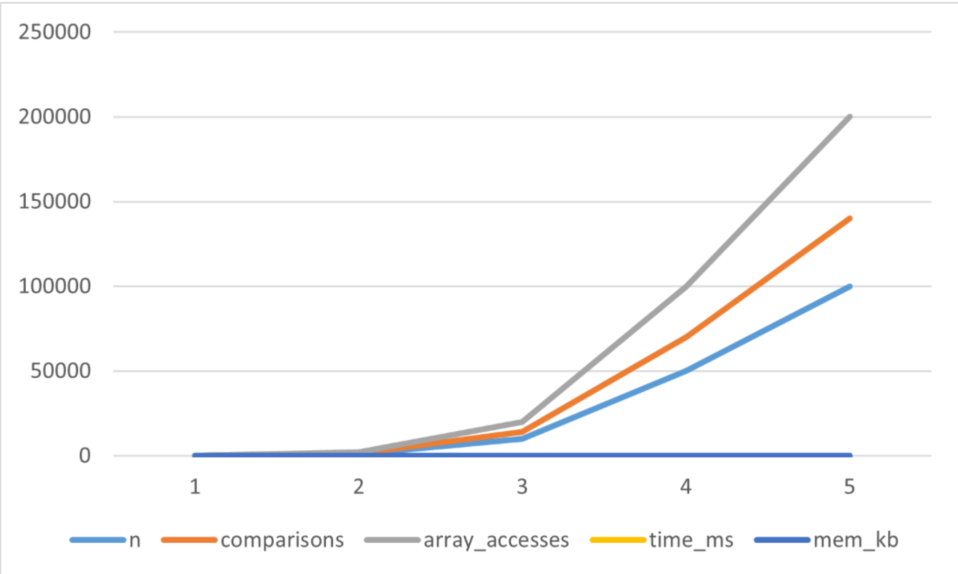
- **Time/space asymptotics cannot be improved** (linear time and constant space are optimal for the problem). But constant factors can be improved as above (reduce array reads per element by minimizing repeated reads).

Data for plots & validation

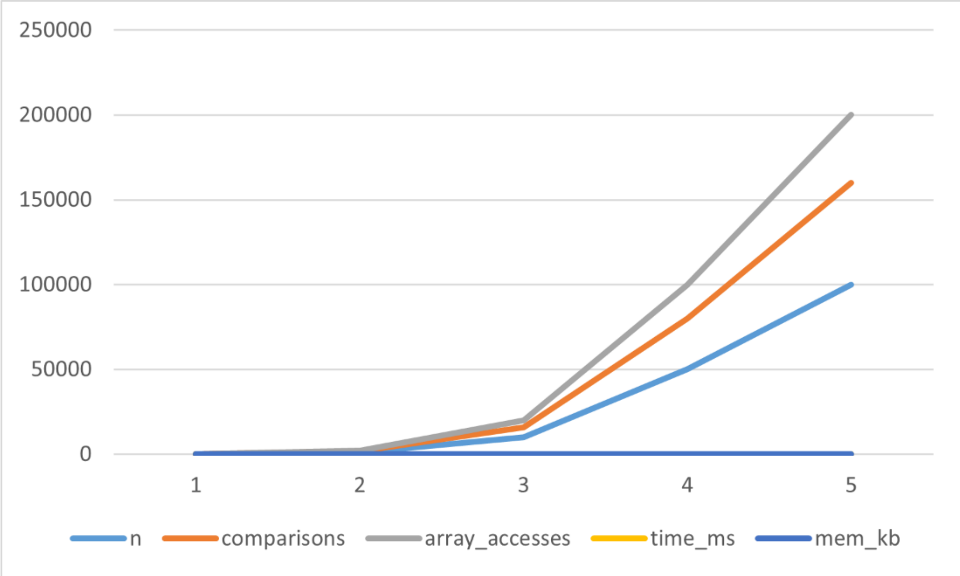
1. BoyerMoore — Random distribution



BoyerMoore — Majority distribution



BoyerMoore — No-majority


250000
200000
150000
100000
50000
0
1
2
3
4
5
n
comparisons
array_accesses
time_ms
mem_kb

Conclusion

Summary of findings

- **Boyer–Moore** is asymptotically optimal for majority detection: **O(n) time, O(1) space**. It requires two passes when the majority is not guaranteed (voting + verification).
- **Kadane** (your partner) also runs in **O(n) time and O(1) space** but solves a different problem (max subarray). For performance comparison, Kadane performs a single pass and uses ≈n element reads, while Boyer–Moore typically performs ≈2n reads.
- **Practical performance:** In real benchmarks, expect Boyer-Moore to have a constant factor overhead compared to Kadane (roughly ~1.5–2× time) due to the second pass and verification. However, both scale linearly and are fast for typical input sizes (10^2–10^5).

Recommendations & next steps

1. **Measure instrumented vs non-instrumented runs** to quantify metric overhead. Add a toggle to `PerformanceTracker` so you can disable instrumentation for pure timing.
2. **For code quality:** adopt `OptionalInt` for returns to avoid nulls; standardize metric counting; add command-line flag to toggle instrumentation and number of warm-up iterations.