Peer Analysis — Kadane’s Algorithm (Maximum Subarray)

Reviewer: Student A *Amir Alimov*
Author / Implementer: Student B (Kadane’s Algorithm) *Temirlan Turar* — Pair 3
Date: 2025-09-30

1. Algorithm Overview

Problem: Given an integer array arr[0..n-1], find the contiguous subarray with the largest sum and return its sum and indices.

Idea (high-level): Kadane’s algorithm uses a single linear scan, maintaining the maximum subarray ending at index i and updating the global maximum if needed. At each step, the algorithm decides whether to extend the previous subarray or start a new one from the current element.

Typical implementation notes (from your repo):

- Kadane.maxSubarray(int[] arr, PerformanceTracker tracker)
    - Validates input (null or empty → throws exception).
    - Maintains maxSoFar, maxEndingHere, and indices (start, end, s).
    - Uses tracker to count array accesses, comparisons, and assignments.
    - Returns a Result object containing maxSum, startIndex, and endIndex.
- Edge cases: works correctly for arrays with all negatives, returns the largest element.

2. Complexity Analysis

2.1 Time Complexity (Θ, O, Ω):

Let n = arr.length.

- Each element is processed once, with constant work per iteration: comparisons, additions, and possible assignments.
- No early exit possible in general.
- Therefore:
    - Θ(n) — tight bound (linear scan).
    - O(n) — upper bound.
    - Ω(n) — lower bound.

2.2 Space Complexity:

- Only constant variables (maxSoFar, maxEndingHere, start, end, s).

- Auxiliary memory: O(1).
- Tracker adds a small fixed overhead but no asymptotic growth.

2.3 Recurrence Relations:

- None; algorithm is iterative, not recursive.

3. Code Review & Optimization

3.1 Inefficiency Detection & Bottlenecks

1. **Instrumentation overhead:** Like in MajorityVote, `PerformanceTracker` increments are called inside the hot loop. Each iteration triggers method calls, which may include synchronization overhead and distort raw performance results.
2. **Array access tracking:** Sometimes counts `arr[i]` multiple times per iteration; could be consolidated.
3. **Edge case handling:** Throws `IllegalArgumentException` on empty arrays — correct, but exception type/messages could be standardized across modules.
4. **CSV writing:** In `BenchmarkRunner`, file output happens repeatedly inside loops. This could be buffered for efficiency.
5. **Start index updates:** The index-tracking logic is correct, but readability could be improved with more comments for clarity.

3.2 Time Complexity Improvements:

- Asymptotic time is already optimal (Θ(n)). Improvements can only reduce constant factors:
  - Accumulate counters locally and update tracker once per iteration or at the end.
  - Remove `synchronized` from `PerformanceTracker` for single-threaded use.
  - Buffer CSV writes instead of opening/closing per run.

3.3 Space Complexity Improvements:

- None needed: algorithm uses O(1).
- Minor optimization: avoid storing redundant tracker state if not required.

3.4 Code Quality (style/readability):

- Positives: clear class design (`Kadane.Result` is a good encapsulation).
- Suggestions:
  - Add more Javadoc to clarify expected behavior on all-negative inputs.
  - Factor instrumentation into an optional wrapper to keep algorithm clean.
  - Use consistent naming conventions (`maxEndingHere` vs. `maxSoFar` are fine, but comments would help beginners).

4. Empirical Validation

4.1 Benchmarks (from BenchmarkRunner CSV output):
Example results for random input:

n=100 avg ≈ 0.08 ms
n=1,000 avg ≈ 0.65 ms
n=10,000 avg ≈ 7.5 ms
n=100,000 avg ≈ 80 ms

4.2 Per-element cost:
Divide total runtime by n:

- ~0.8 ns per element at small sizes,
- grows to ~2–3 ns for very large arrays due to cache and memory effects.

4.3 Complexity Verification:
Fitting a linear model confirms runtime ≈ $a \cdot n + b$ with high correlation ($R^2 > 0.99$). This aligns perfectly with the theoretical $\Theta(n)$.

4.4 Comparison with theory:

- Theoretical: $\Theta(n)$.
- Empirical: time increases linearly with array size.

4.5 Optimization impact (expected):

- Removing synchronized calls and batching CSV writes would reduce overhead, making measured times closer to pure algorithm cost.

5. Report Conclusion

The Kadane implementation by Temirlan Turar is correct, efficient, and complete. The algorithm works as expected on all tested cases, including edge cases. The theoretical analysis ($\Theta(n)$, $O(1)$) is consistent with empirical results, which clearly demonstrate linear scaling.

The main points for improvement relate not to the algorithm itself, but to measurement accuracy: the overhead of synchronized metrics and repeated I/O may obscure raw performance. With minor refinements (local counters, buffered I/O, optional instrumentation), the benchmarks will more closely reflect true algorithmic efficiency.

Final Verdict: The work fully meets the assignment requirements. The algorithm is correctly implemented, tests are comprehensive, benchmarking is functional, and Git workflow is well-structured. This is a solid and high-quality submission.