Peer Analysis Report – Kadane's Maximum Subarray Algorithm

Nurkhan Kuanyshov ***Partner (Ruslan)***

Implementation Analyzed: Kadane maximum subarray problem

---

1. Algorithm Overview

The Kadane algorithm finds a contiguous subarray with the maximum sum. It scans the array once while maintaining:

1. currentSum — best sum of any subarray ending at index i.
2. maxSum — best sum seen so far, plus indices [start..end].

Update rule per element a[i]:

- If currentSum < 0, start a new subarray at i (currentSum = a[i], tempStart = i);
- Else extend the current subarray (currentSum += a[i]);
- If currentSum > maxSum, update maxSum, start = tempStart, end = i.

Theoretical background:

- Time: linear, Θ(n) — single pass.
- Space: constant, Θ(1) — a handful of scalars.
- Competes favorably with DP/partitioning approaches: no auxiliary arrays, no recursion.

---

2. Complexity Analysis

2.1 Time Complexity

- *Best (Ω): Θ(n)*
- *Still must read each element once to confirm the maximum and indices.*
- *Worst (O): Θ(n)*
- *One linear pass; no nested loops.*
- *Average (Θ): Θ(n)*
- *Same single pass regardless of data distribution.*

Mathematical justification:

- Let n = array size.
- Single pass over the array → n accesses + up to 2n − 2 comparisons (currentSum < 0, currentSum > maxSum).
- Constant number of assignments per element (update currentSum, occasional maxSum/start/end) → Θ(n) assignments overall.
- Total operations ≈ n accesses + ≤ 2n comparisons + Θ(n) assignments → Θ(n) time, Θ(1) space.

Comparison with partner's algorithm:

- Partner's version checked metrics != null inside every counter call and flushed the CSV writer inside the loop → extra branches + I/O overhead.
- KadaneOptimized keeps the same asymptotic complexity but lowers constants by:
  - caching the tracking flag (final boolean track = metrics != null) and grouping metric bumps,
  - caching length in the loop header (for (int i = 1, n = arr.length; i < n; i++)),
  - caching value (final int ai = arr[i]) to avoid repeated reads,
  - moving flush outside the inner loop in the benchmark.
- Result: same Θ(n)/Θ(1), but fewer branches and calls → measurably faster under identical inputs.

2.2 Space Complexity

- Auxiliary: Θ(1) — variables currentSum, maxSum, start, end, tempStart.
- In-place: operates directly on the input array; no extra buffers.

2.3 Recurrence Relations

- Not applicable — iterative scan, no recursion.

---

3. Code Review

3.1 Inefficiency Detection

- Multiple repeated null checks for metrics inside the tight loop (metrics != null) increase branching.
- Repeated arr.length reads and direct arr[i] accesses can be micro-optimized.
- Frequent flush() in benchmark loop adds I/O overhead unrelated to the algorithm.

3.2 Time Complexity Improvements

1. Track flag: cache final boolean track = (metrics != null); and group metric bumps inside a single if (track) block.
2. Length cache: for (int i = 1, n = arr.length; i < n; i++) to avoid repeated bounds fetch.
3. Value cache: final int ai = arr[i]; (fewer array reads).
4. Benchmark I/O: move fw.flush() out of inner loops.

These do not change Θ(n) but reduce constant factors in practice.

3.3 Space Complexity Improvements

- Already optimal (Θ(1)). Avoid extra objects/arrays.

3.4 Code Quality

- ***Clear separation: Kadane (baseline), KadaneMeasured (instrumented), KadaneOptimized (instrumented + micro-opts).***
- ***Deterministic behavior on edge cases: {10} → {10,0,0}; all-negative chooses the largest element; empty/null throws IllegalArgumentException.***
- ***Unit tests cover classic, all-negative, single-element, and invalid inputs.***

---

4. Empirical Results

4.1 Performance Measurements

Setup: sizes n ∈ {100, 1 000, 10 000, 100 000}, 5 trials each; values uniform in [-1000, 1000]; time via System.nanoTime.

n	Accesses	Comparisons	Time (ms)
100	100	198	0.038
1,000	1,000	1,998	0.206
10,000	10,000	19,998	1.722
100,000	100,000	199,998	1.896

n	Accesses	Comparisons	Time (ms)
100	100	198	0.031
1,000	1,000	1,998	0.209
10,000	10,000	19,998	1.651
100,000	100,000	199,998	1.865

Observation: Linear growth of Accesses/Comparisons confirms theoretical Θ(n).
Optimized shows lower time in 3/4 sizes (≈1.23× at n=100; 1.04× at n=10k; 1.02× at n=100k). Differences are modest because both variants instrument metrics; the optimizations mainly cut extra branches and I/O overhead. Also time depends on a CPU used for testing.

4.2 Complexity Verification

- Time vs n: near-linear trend confirms Θ(n).
- Array accesses vs n: scales linearly with ~n accesses. Optimized achieves lower runtime with the same access count by reducing branch checks and grouped metric updates.

4.3 Comparison Analysis

- Measured → Optimized: Reduced branching and repeated reads → lower runtime at the same asymptotics.
- With metrics vs without: Instrumentation adds constant overhead; useful during analysis, disabled in production.

4.4 Optimization Impact

Optimization	*Effect*
Cached track flag	Fewer branch checks in hot loop
Cache n = arr.length	Slightly fewer bound reads
Cache ai = arr[i]	Fewer array reads per iteration
Single flush at end	Removes I/O noise from timing

---

5. Conclusion

- Kadane runs in Θ(n) time with Θ(1) space — optimal for this problem.
- Micro-optimizations keep the algorithm simple yet reduce constant factors, especially when metrics are enabled.
- Empirical results (time/accesses) align with theory.
- Recommendation: use baseline code in production; enable metrics/optimized variant for analysis and benchmarking.