Analysis Report: Kadane's Algorithm

Partner Implementation Review

Reviewer: Ahmetov Rasul

Partner: Usen Asylan

Algorithm: Kadane's Algorithm (Maximum Subarray Sum)

Date: October 2025

1. Algorithm Overview

Hey Asylan! Really impressed with your Kadane's implementation. This algorithm solves the maximum subarray problem - finding the contiguous subarray with the largest sum. It's a classic dynamic programming problem with applications everywhere from stock trading to data analysis.

How It Works

Your implementation uses a clever single-pass approach with dynamic programming:

Core Logic: At each position, you make a smart decision - either:

- **Extend** the current subarray (add current element to running sum)
- **Start fresh** from the current element (if previous sum is negative)

The genius is that you maintain:

- `currentSum`: Best sum ending at current position
- `maxSum`: Global best found so far
- Position tracking: Start and end indices of the winning subarray

```
currentSum = Math.max(arr[i], currentSum + arr[i]);
maxSum = Math.max(maxSum, currentSum);
```

Why It Works: If the running sum becomes negative, it can only hurt future subarrays, so you restart. The maximum sum ending at position i is either just the element itself or the previous sum extended.

2. Complexity Analysis

2.1 Time Complexity

Best Case: $\Theta(n)$ *with your optimization*

Scenario: All positive numbers `[3, 7, 2, 9, 5]`

Your early termination optimization:

```
// Phase 1: Check if all positive
for (int num : arr) {
    if (num <= 0) break;
}
// If all positive, return sum immediately
```

Operations:

- First pass: n comparisons (check all positive)
- Early return: $O(1)$
- **Total: $n + c = \Theta(n)$**

Note: Even with optimization, still $O(n)$ since you need to verify all elements are positive!

Worst Case: $\Theta(n)$

Scenario: All negative numbers `[-5, -2, -8, -1]` or alternating signs

Operations:

- Single pass: n iterations
- Per iteration: 2-3 comparisons (currentSum update + maxSum check)
- **Total: ~2n comparisons = $\Theta(n)$**

Your benchmark proves this: 10,000 elements = 19,998 comparisons ≈ 2n ✓

Average Case: $\Theta(n)$

Scenario: Mixed positive/negative `[5, -3, 8, -2, 1]`

Operations per element:

- 1 array access: `arr[i]`
- 1-2 additions/comparisons: `Math.max(arr[i], currentSum + arr[i])`
- 1 comparison: `if (currentSum > maxSum)`
- 0-1 position updates

Total: ~2-3 operations per element = $\Theta(n)$

Mathematical Justification:

```
T(n) = c₁ (initialization)
     + c₂·n (main loop with constant work per iteration)
     + c₃ (finalization)
     = Θ(n)

Lower Bound Ω(n): Must examine every element at least once
Upper Bound O(n): Never more than 3n operations
Tight Bound Θ(n): Always linear ✓
```

Your algorithm is optimal! You can't solve maximum subarray faster than $O(n)$ without additional constraints.

2.2 Space Complexity: $\Theta(1)$

Variables used:

- `currentSum`, `maxSum`: 8-16 bytes
- `start`, `end`, `tempStart`: 12 bytes
- `comparisons`, `arrayAccesses`: 8 bytes (metrics)
- Loop variable `i`: 4 bytes

Total: ~40 bytes regardless of n - excellent constant space!

Comparison with alternatives:

Approach	Time	Space
Brute Force	$O(n^3)$	$O(1)$
Prefix Sum	$O(n^2)$	$O(n)$
Divide & Conquer	$O(n \log n)$	$O(\log n)$
Your Kadane's	**$\Theta(n)$**	**$\Theta(1)$**

Result: Optimal in both dimensions! ✓

3. Code Review

3.1 What You Did Great!

1. Clean Position Tracking

```
// Love how you track the actual subarray indices
if (currentSum < arr[i]) {
    currentSum = arr[i];
    tempStart = i;  // Smart! Reset start position
}
```

Most implementations just return the sum. Yours returns the actual subarray location - way more useful!

2. Professional Git Workflow

Your commit history is textbook perfect:

- ✓ feat/algorithm - baseline implementation
- ✓ feat/metrics - performance tracking
- ✓ feat/cli - benchmark runner
- ✓ feat/optimization - early termination
- ✓ Proper PRs with merge commits

This is exactly what the assignment asked for. Really professional work!

3. Comprehensive Testing

23+ test cases covering:

- ✓ Edge cases: empty, single element, null
- ✓ All positive/negative arrays
- ✓ Mixed sequences
- ✓ Performance validation

Your test suite is solid!

4. Performance Metrics Integration

```
tracker.incrementComparisons();
tracker.incrementArrayAccesses();
```

Built-in benchmarking is super valuable for empirical analysis. Smart design!

5. CLI with Configurable Sizes

```
mvn exec:java -Dexec.args="5000"
```

Makes it easy to test different input sizes. Nice touch!

3.2 Issues Found (Minor)

Issue #1: Early Termination Trade-off ⚠️

Location: Pre-check for all-positive arrays

```
// Current approach:
boolean allPositive = true;
for (int num : arr) {          // First pass: O(n)
    if (num <= 0) {
        allPositive = false;
        break;
    }
}
if (allPositive) {
    return totalSum;            // O(1) return
}
// Then main algorithm runs    // Second pass: O(n)
```

Problem: For non-all-positive cases (the common case), you're doing EXTRA work:

- Added n comparisons checking for positives
- Extra loop iteration
- For random data: ~50% overhead on first phase

Impact: Your metrics show ~2n comparisons. Without pre-check, might be ~1.5n.

Suggestion 1 - Remove it:

```
// Just run main algorithm directly
// It's already O(n), no need to check first
```

Suggestion 2 - Integrate into main loop:

```
boolean sawNegative = false;
for (int i = 0; i < arr.length; i++) {
    if (arr[i] <= 0) sawNegative = true;
    // ... rest of algorithm
}
if (!sawNegative) maxSum = totalSum;  // Adjust at end
```

Expected improvement: 5-10% speedup for typical cases.

Issue #2: Integer Overflow Risk ⚠️

Location: Sum calculations

```
long currentSum = 0;    // You use int
long maxSum = Long.MIN_VALUE;
```

Scenario that breaks:

```
int[] arr = {Integer.MAX_VALUE/2, Integer.MAX_VALUE/2, 100};
// Sum overflows Integer.MAX_VALUE
```

Your code:

```
currentSum = currentSum + arr[i];  // Can overflow!
```

Fix:

```
// Change to long
long currentSum = 0;
long maxSum = Long.MIN_VALUE;
```

Impact: Edge case, but important for robustness. For production code, always use `long` for sums.

Issue #3: Metrics Overhead in Production 🎯

Current: Metrics always enabled

```
tracker.incrementComparisons();  // Every iteration
tracker.incrementArrayAccesses();
```

Suggestion:

```
// Add flag to disable in production
if (METRICS_ENABLED) {
    tracker.incrementComparisons();
}
```

Or use Java's `-ea` assertions:

```
assert tracker.incrementComparisons() || true;  // Only runs with -ea
```

Expected gain: 5-10% when disabled.

3.3 Code Quality Score

Overall Grade

A-

(93/100)

Strengths:

- ✓ Crystal clear variable names
- ✓ Single responsibility methods
- ✓ Professional error handling
- ✓ Comprehensive test coverage
- ✓ Excellent documentation
- ✓ Maven structure perfect

Minor Improvements:

- Extract magic numbers
- Use `long` for sums
- Consider removing early termination
- Make metrics optional

Really solid implementation! The issues are all minor optimization opportunities.

4. Optimization Suggestions

4.1 Code-Level Improvements

1. Remove Pre-check Overhead (Priority: High)

- Before: ~3.5n operations for mixed arrays
- After: ~2.5n operations
- Time saved: ~30% on first phase

2. Use Long for Overflow Safety (Priority: High)

```
// Just change type:
long currentSum = 0;
long maxSum = Long.MIN_VALUE;
```

3. Conditional Metrics (Priority: Medium)

```
private static final boolean METRICS_ENABLED = false;  // Toggle

public Result findMaxSubarray(int[] arr) {
    PerformanceTracker tracker = METRICS_ENABLED ?
        new PerformanceTracker() : new NoOpTracker();
    // ... rest of code
}
```

4.2 Time/Space Complexity - Already Optimal! ✓

Your algorithm is **theoretically optimal**. You literally cannot do better than:

- **Time: $\Theta(n)$** - must read every element
- **Space: $\Theta(1)$** - only constant variables

Any improvement would be in constant factors, not asymptotic complexity. Well done!

5. Empirical Validation

5.1 Your Benchmark Results

Input Size	Time (ms)	Comparisons	Array Accesses	Comp/n Ratio
100	0.011	198	~200	1.98
1,000	0.133	1,998	~2,000	1.998
10,000	0.548	19,998	~20,000	1.9998
50,000	2.779	99,998	~100,000	1.99996

5.2 Analysis

Perfect linear growth! ✓

The ~2n ratio is crystal clear:

- 1 comparison: `Math.max(arr[i], currentSum + arr[i])`
- 1 comparison: `if (currentSum > maxSum)`
- Total: 2 comparisons per element

Scaling verification:

- 10× input (100→1,000): 10.09× comparisons ✓
- 10× input (1K→10K): 10.01× comparisons ✓
- 5× input (10K→50K): 5.00× comparisons ✓

Theoretical match: Perfect!

Time shows sub-linear scaling (12.1×, 4.12×, 5.07×) - this is due to:

- CPU cache warming up
- JVM JIT compilation
- Better cache locality with larger arrays
- Fixed initialization overhead becoming negligible

This is **expected** and actually shows your implementation is cache-friendly!

5.3 Comparison: Kadane's vs. Boyer-Moore

Metric	Kadane's (You)	Boyer-Moore (Me)
Time Complexity	$\Theta(n)$ - 1 pass	$\Theta(n)$ - 2 passes
Space Complexity	$\Theta(1)$	$\Theta(1)$
Passes Required	1	2 (vote + verify)
Comparisons/n	~2	~2-3
Must Verify	No	Yes (correctness)
Position Tracking	Start + End	First + Last