Algorithm Analysis Report: Kadane's Algorithm Implementation

Student: Vladislav Kutsenko
Reviewer: Madiyar Sarbayev
Date: October 6, 2025

---

1. Algorithm Overview

Kadane's Algorithm solves the maximum subarray problem: finding the contiguous subarray with the largest sum in a one-dimensional array. This implementation extends the classic algorithm by tracking the exact position (start and end indices) of the optimal subarray.

The algorithm works through a single pass of the array. It maintains two key values at each step: the maximum sum ending at the current position and the global maximum sum seen so far. When adding an element would decrease the running sum below the element itself, the algorithm starts fresh from that element.

Core Logic:

- Track currentSum: running sum of current candidate subarray
- Track maxSum: best sum found so far
- Compare at each step: should we extend the current subarray or start a new one?
- Update position markers when a new maximum is found

This approach eliminates the need for nested loops, making it fundamentally more efficient than brute force methods that would examine all possible subarrays.

2. Complexity Analysis

2.1 Time Complexity Derivation

Best Case: Θ(n)

The algorithm processes each array element exactly once, regardless of input characteristics. Even for arrays where the maximum is found immediately (e.g., [100, 1, 2, 3]), the algorithm still traverses the entire array to verify no better subarray exists later.

Operations per iteration:

- 1 array access: array[i]
- 2 comparisons: checking if we should reset currentSum and if we found a new maximum
- Constant-time arithmetic and assignments

Total operations: $T(n) = c_1 + (n-1)(c_2) = \Theta(n)$ where c_1 represents initialization and c_2 represents constant work per loop iteration.

Worst Case: Θ(n)

No input arrangement changes the fundamental single-pass nature. Whether all elements are negative, all positive, or alternating, the algorithm still executes the same number of operations.

Average Case: Θ(n)

For random inputs, the algorithm's behavior remains consistent. The number of times we reset the subarray start position doesn't change the asymptotic complexity since each reset is O(1).

Mathematical Justification:

Let T(n) be the exact number of operations:

- $T(n) = 1$ (initial access) + $(n-1) \times k$ (loop iterations with constant work)
- $T(n) = 1 + kn - k = kn + (1-k)$

By definition of Big-Theta:

- $\exists\ c_1, c_2, n_0$ such that $c_1 \cdot n \leq T(n) \leq c_2 \cdot n$ for all $n \geq n_0$
- With $c_1 = k/2$ and $c_2 = 2k$, this holds for sufficiently large n
- Therefore $T(n) = \Theta(n)$

Since best case = worst case = Θ(n), we also have:

- Ω(n): algorithm must read all elements
- O(n): algorithm reads each element at most once

2.2 Space Complexity Derivation

Auxiliary Space: O(1)

The algorithm uses a fixed number of variables regardless of input size:

- maxSum, currentSum: 2 integers
- start, end, tempStart: 3 integers
- Loop counter i: 1 integer
- currentElement: 1 integer

Total: 7 integer variables = constant space

The MaxSubarrayResult object uses 3 integers, also constant.

Total Space: O(n)

Counting the input array itself: O(n) for the array plus O(1) for auxiliary variables = O(n) total.

2.3 Comparison Analysis

Vladislav's implementation achieves optimal complexity for this problem:

Approach	Time	Space	Notes
Kadane (Vladislav)	Θ(n)	O(1)	Optimal
Divide & Conquer	Θ(n log n)	O(log n)	Slower but educational
Brute Force	Θ(n²)	O(1)	Checks all subarrays
Dynamic Programming	Θ(n)	O(n)	Same time, more space

Kadane's algorithm is provably optimal because any correct solution must examine every array element at least once (lower bound Ω(n)), and this implementation achieves exactly that.

3. Code Review

3.1 Identified Issues

Issue 1: Redundant Variable Assignment

```
tracker.incrementArrayAccesses();
int currentElement = array[i];
```

The currentElement variable is used only once immediately after assignment. This adds an unnecessary variable declaration and assignment operation.

Rationale: Modern compilers may optimize this away, but it reduces code clarity. Direct use of array[i] would be cleaner.

Issue 2: Double Tracking Overhead

```
tracker.incrementComparisons();
if (currentElement > currentSum + currentElement) {
    // ...
}
```

Every comparison increments the tracker. For performance-critical production code, this tracking overhead (method call + increment) happens 2n times per execution.

Rationale: While valuable for benchmarking, this adds ~10-20% overhead in practice. Consider making tracking optional via a flag or interface.

Issue 3: Inefficient Comparison Logic

```
if (currentElement > currentSum + currentElement) {
    currentSum = currentElement;
    tempStart = i;
} else {
    currentSum = currentSum + currentElement;
}
```

The condition currentElement > currentSum + currentElement is algebraically equivalent to 0 > currentSum or currentSum < 0. This performs unnecessary addition in the comparison.

Optimization:

```
if (currentSum < 0) {
    currentSum = array[i];
    tempStart = i;
} else {
    currentSum += array[i];
```

}

Rationale: This eliminates redundant addition, one array access, and variable storage. The simplified condition is also more readable.

4. Empirical Results

4.1 Expected Performance Characteristics

Based on the implementation, we expect:

Time Complexity Validation:

- Linear relationship between input size and execution time
- Doubling input size should approximately double execution time
- Graph of time vs size should show straight line on linear scale

Operation Counts:

- Array accesses: exactly n (one per element)
- Comparisons: exactly 2(n-1) (two per loop iteration)
- Both should scale linearly with input size

4.2 Benchmark Configuration

The implementation uses:

- Warmup iterations: 5 (JVM optimization)
- Benchmark iterations: 10 (averaged results)
- Random input range: -1000 to 1000
- Fixed seed (42) for reproducibility
- Test sizes: 100, 1,000, 10,000, 100,000

This configuration is appropriate for measuring steady-state performance after JIT compilation.

4.3 Theoretical vs Practical Analysis

Expected Results:

For input size n:

- Array accesses = n
- Comparisons = 2(n-1)
- Execution time ≈ k·n for some constant k

Example predictions for n=100,000:

- Array accesses: 100,000
- Comparisons: 199,998
- Time: ~1-3ms (depending on hardware)

Constant Factors:

The practical performance depends on:

1. **Memory access patterns**: Sequential array access is cache-friendly, contributing to good real-world performance
2. **Branch prediction**: The two if-statements create branches that modern CPUs can predict reasonably well for random data
3. **Tracking overhead**: PerformanceTracker method calls add ~15-20% overhead vs. pure algorithm
4. **JIT compilation**: Hotspot can optimize the tight loop after warmup

Validation Method:

To verify Θ(n) complexity empirically:

1. Plot execution time vs input size on linear scale → expect straight line
2. Calculate time ratios: T(2n)/T(n) → expect ratio ≈ 2.0
3. Verify operation counts: accesses = n, comparisons = 2(n-1)

4.4 Performance Plot Analysis

Expected graph characteristics:

Expected Benchmark Data:

Input Size (n)	Array Accesses	Comparisons	Expected Time (ms)	Time Ratio T(n)/T(prev)
100	100	198	0.001-0.005	-
1,000	1,000	1,998	0.010-0.050	~10.0
10,000	10,000	19,998	0.100-0.500	~10.0
100,000	100,000	199,998	1.000-5.000	~10.0

Linear relationship confirms Θ(n) complexity

Any deviation from linearity would indicate:

- Sublinear: impossible for this problem (must read all elements)
- Superlinear: indicates unexpected overhead or inefficiency

Hardware Considerations:

Performance will vary with:

- CPU clock speed and architecture
- Cache size (L1, L2, L3)
- Memory bandwidth
- JVM version and optimization level

The constant factor k in T(n) = k·n captures these hardware-specific effects.

---

5. Conclusion

5.1 Summary of Findings

Vladislav's implementation successfully achieves optimal theoretical complexity for the maximum subarray problem. The algorithm demonstrates:

Strengths:

- Correct Θ(n) time complexity in all cases
- Minimal O(1) auxiliary space usage
- Accurate position tracking for the optimal subarray
- Comprehensive error handling for edge cases
- Well-structured code with clear separation of concerns

Areas for Improvement:

- Redundant intermediate variable currentElement
- Inefficient comparison using currentElement > currentSum + currentElement instead of currentSum < 0
- Performance tracking overhead embedded in core algorithm
- Missing early exit optimization for single-element arrays

5.2 Optimization Recommendations

Priority 1 (High Impact):

1. **Simplify comparison logic** from currentElement > currentSum + currentElement to currentSum < 0

    - Impact: Eliminates unnecessary addition per iteration
    - Estimated improvement: 5-8%
2. **Make tracking optional** through strategy pattern or flag

    - Impact: Removes method call overhead for production use
    - Estimated improvement: 15-20% when tracking disabled

Priority 2 (Code Quality): 3. Remove intermediate variable currentElement

- Impact: Cleaner code, slightly fewer operations
- Estimated improvement: 2-3%
4. **Add early exit** for single-element arrays
    - Impact: Better performance for edge cases
    - Estimated improvement: Only affects n=1 case

5.3 Final Assessment

The implementation is fundamentally sound and achieves optimal algorithmic complexity. The suggested optimizations improve constant factors without changing asymptotic behavior. For academic purposes, the current implementation effectively demonstrates Kadane's algorithm. For production use, implementing Priority 1 optimizations would be worthwhile.

Overall Grade Components:

- Correctness: Excellent
- Complexity: Optimal
- Code Quality: Good
- Testing: Comprehensive
- Documentation: Clear

The code successfully solves the problem with the best possible complexity class.