Algorithmic Analysis Report

Kadane's Algorithm

Peer Review by Serik Aitkaziev

SE-2419

1. Algorithm Overview

The Maximum Subarray Problem is a classical task in computer science and algorithm design. It asks to find a contiguous subarray within a one-dimensional array of numbers which has the largest possible sum. The problem originates from signal processing and was first formally described by Jay Kadane in 1984, hence the name Kadane's Algorithm.

Kadane's Algorithm is a dynamic programming technique used to find the contiguous subarray within a one-dimensional array that has the largest sum. The main idea is to iterate through the array while maintaining two variables:

1. currentSum – the maximum subarray sum ending at the current index;
2. maxSum – the global maximum found so far.

At each step, the algorithm decides whether to extend the previous subarray or start a new one from the current element. This implementation also tracks the start and end indices of the maximum subarray for position tracking and debugging.

Recurrence relation:
$S(i) = \max(a[i], S(i-1) + a[i]), \quad S(0) = a[0]$

Pseudocode:

```
maxSum = currentSum = array[0]
for i = 1 to n-1:
   currentSum = max(array[i], currentSum + array[i])
   maxSum = max(maxSum, currentSum)
return maxSum
```

2. Complexity Analysis

Time Complexity Derivation:
Let T(n) denote the total running time for an input array of size n.
Each iteration performs a constant number of operations: one comparison, one addition, and one assignment.
$T(n) = T(n-1) + \Theta(1) \Rightarrow T(n) = \Theta(n)$.

Case Analysis:
• Best Case ($\Theta(n)$) – all elements positive.

• Average Case (Θ(n)) – random input, one pass.
• Worst Case (Θ(n)) – all negative values, full scan.
Hence Θ(n) = O(n) = Ω(n).

Space Complexity:
The algorithm uses constant auxiliary memory (currentSum, maxSum, counters) ⇒ Θ(1).

Algorithm	Time Complexity	Space	Approach
Brute Force	$O(n^2)$	O(1)	Double loop
Divide & Conquer	O(n log n)	O(log n)	Recursive merging
Kadane's	O(n)	O(1)	Iterative DP

3. Code Review

Code Quality:
The implementation is clear, well-commented, and uses descriptive variable names (currentSum, maxSum, start, end, tempStart). Edge cases (empty arrays, all-negative arrays, single-element arrays) are handled.

Inefficiency Detection:
- No major algorithmic inefficiencies found; Kadane is asymptotically optimal.
- Micro-optimizations: reduce redundant assignments, avoid unnecessary object boxing, factor position-tracking into a helper method for clarity.

Optimization Suggestions:
- Use primitive arrays (int[]) in Java hot paths to avoid autoboxing overhead.
- Add optional JMH microbenchmarks for precise timing.
- For extremely large data sets, consider a segmented/parallel approach (divide the array, run Kadane locally, then combine results).

4. Empirical Validation

Benchmark Setup:
Measured average runtime for arrays with random integer values in range [-1000, 1000]. Input sizes: n = 100, 1,000, 10,000, 100,000. Each measurement is average of 5 runs.

Sample Results (average of 5 runs):

n	Time (ms)
100	0.03
1000	0.11
10000	0.86
100000	5.47

Validation:
The runtime scales linearly with n, confirming the theoretical O(n) complexity. The plot included shows Time vs n (log-log scale) with an approximately linear trend indicating a slope near 1.

Comparison:
Compared to divide-and-conquer approaches (O(n log n)), Kadane performs better in practice for all tested sizes.

5. Conclusion

The implementation is correct, efficient, and well-suited to the task. It achieves optimal time (O(n)) and space (O(1)) complexity. Recommended improvements are minor: add JMH benchmarking, ensure use of primitives if currently boxed types are used, and add brief helper functions for position tracking to improve readability.

Final Recommendation:
Accept the implementation as high-quality. Include the suggested microbenchmarks and documentation for completeness.

Performance Plot

Figure 1: Measured average runtime (log-log scale) for input sizes n = 100, 1,000, 10,000, 100,000.