Peer analysis report

Partner’s algorithm: Kadane’s algorithm for finding the maximum sum of subarray

Language: Java

Partner: Nurtilek Kobylandy

1. Algorithm Overview

1.1 Brief description

My partner implemented the Kadan algorithm to solve the maximum subarray problem, which consist of finding a continuous part of an array of integers with the largest sum.

The algorithm uses a dynamic programming approach and runs in a single pass over the array. It tracks two key metrics:

- currentSum: the maximum sum of the subarray ending at the current position.

- maxSum: the global maximum subarray found so far.

The implementation of my partner's includes additional logic to correctly handle cases where all elements of the array are non-positive. In this case, the largest element is returned, rather than 0 or an empty array, which is consistent with the standard definition of the problem

1.2 Theoretical Justification

The Kadan algorithm is based on a simple inductive step: if the maximum sum of the subarray ending at position (i-1) is negative, then starting a new subarray with element arr[i] will always be more advantageous than continuing old one.

Basic recurrence relation:

- CurrentSum(i) = max(arr[i], CurrentSum(i-1) + arr[i])

The global maximum sum is updated at each step:

- MaxSum = max(MaxSum, CurrentSum(i))

Since the solution doesn’t require recursion and only uses an iterative pass, it achieves the highest possible efficiency.

2. Complexity Analysis

2.1 Time Complexity

The time complexity of the algorithm is entirely determined by a single for loop, which iterates over the input array of size n exactly once. All operations within the loop are performed in constant time O(1)

Case	Θ (exact boundary)	O (upper)	Ω (lower)	Mathematical Justification
Best	Θ(n)	O(n)	Ω(n)	∑i=1,n, c =c*n The work doesn’t depend on the data. For example, all positive
Worst	Θ(n)	O(n)	Ω(n)	∑i=1,n, c =c*n The work doesn’t depend on the data. For example, alternating sings
Average	Θ(n)	O(n)	Ω(n)	Since the number of operations doesn’t depend on the distribution of data, the average complexity coincides with the worst and best cases.

Mathematical Justification
The total operating time T(n) can be expressed as:

$$T(n) = c_{\text{init}} + \sum_{i=0}^{n-1} c_{\text{loop}} + c_{\text{final}}$$

Where:
- c_init = Constant time for initializing variables and checking for empty arrays

- c_loop = Constant execution time within a cycle. For example, addition, comparison, metric updates
- c_final = Constant time for final checks and returning the result

Consequently:
T(n) = c_init + c_loop + c_final

By definition of the large Θ:

The implementation of my partner's achieves the theoretically optimal time complexity for this task. The theoretical lower bound is necessary because finding the maximum sum of a subarray requires visiting at least every element of the array once

Recursive Relations:

For the iterative Kadane algorithm, the following recursive relation can be modeled, representing the decreasing size of the subtask:

$$T(n) = T(n-1) + c \quad \textit{for } n > 0;$$

With the base case $T(0)=c'$, where c and c' are constants. Solving this equation by substitution (or by the fundamental theorem, if reformulated for recursion) confirms linear complexity:

$$T(n) = c * n + c' \quad \in \Theta(n)$$

2.2. Space Complexity

Analysis	Auxiliary Space	Justification
Space Complexity	O(1)	The algorithm uses a fixed number of variables (maxSum, currentSum, start, end, metrics, etc.), whose memory requirements do not depend on the input size n.
In-place optimization	In-place	The algorithm does not require additional memory allocation scalable relative to n (i.e., O(n)). Thus, it is an optimal in-place solution.

3. Code Review

3.1. Inefficiency Detection

Despite its optimal asymptotic complexity Θ(n), the code contains critical issues related to the constant factor c and the purity of the logic.

- Main Bottleneck:

Excessive I/O. Numerous calls to System.out.printf and System.out.println are located inside the main O(n) loop.

Consequence: Input/output (I/O) operations are 100-1000 times slower than arithmetic operations or memory access. In practice, this makes the algorithm slow, especially for small n, where I/O dominates over computation. Empirical data confirms this: the speed (ns/element) drops from ~78,000 for n=100 to ~71 for n=100,000, indicating the dominance of slow constant overhead for small n.

- Suboptimal code

Nurtilek introduced the variables hasPositive and maxElement to handle the case of all non-positive numbers.

Problem: including these checks in the O(n) loop adds unnecessary constant work on each iteration. The standard Kadan, when initialized correctly, elegently handles the case of all negative numbers

3.2. Optimization Suggestions

3.2.1. Time Complexity Improvements

Optimization 1. Eliminating I/O overhead

Action: completely remove all System.out.println calls used for tracing within the findMaxSubarray method.

Rationale: This does not change Θ(n), but critically reduces the constant factor c. Benchmarking shows that this results in up to a 47% speedup for small n. For debugging, use a professional logger that can be disabled in production.

Optimization 2: Refactoring to the Canonical Kadane Algorithm

Action: Remove the variables hasPositive and maxElement. Initialize maxSum = arr[0], currentSum = arr[0], and start the loop with i=1.

Rationale: This removes redundant O(1) operations on each iteration, makes the code cleaner, and uses a standard, proven solution that correctly handles all negative numbers without special logic.

3.2.2. Space Complexity Improvements

Conclusion: Since the complexity is already O(1), it is not possible to improve the space complexity.

3.3. Code Quality

Aspect	CommentComment
Readability/Style	Low. The code is overloaded with debugging output, which makes it difficult to focus on the main logic. Although the variables are well named, the non-standard approach with hasPositive and maxElement complicates understanding.
Supportability	Low. The non-standard implementation of the Kadan algorithm makes it less understandable for third-party developers. The code contains “patches” (e.g., if (!hasPositive)) that would be unnecessary when using the canonical form.

4. Empirical Results

4.1. Performance Plots and Validation

Theoretical Prediction: Time T(n) should grow strictly linearly ($T(n) \propto n$) for all types of input data.

n	Random (Baseline)	Random (Optimized)	T(n)/n (Baseline, ns/element)
100	7,795,400 ns	4,106,500 ns	77,954
1000	3,176,900 ns	3,257,200 ns	3,177
10000	3,365,700 ns	3,391,200 ns	337
100000	7,114,800 ns	5,332,800 ns	71

Theoretical Complexity Check

- Operation Control: The metric metrics.comparisons (counter $\sim n-1$) strictly confirms that the number of steps performed is linear with respect to n, thereby confirming $\Theta(n)$.

- Time vs n analysis: The graph of time T(n) versus n should be a straight line. However, the ratio T(n)/n (time per element) drops sharply for small n.

Conclusion: This paradoxical decline confirms that for small values of n, the execution time is dominated by constant overhead (I/O, JVM initialization) rather than $\Theta(n)$ work. As n increases, $\Theta(n)$ work (the loop itself) begins to

dominate, and the algorithm actually exhibits linear scaling (linear time growth for large n).

Comparison Analysis

- Empirical data for different distributions (Sorted, Reverse, NearlySorted) show that the execution time for a given n remains within the same order of magnitude. This reliably confirms the theoretical conclusion: the complexity of the algorithm does not depend on the input data and is always equal to Θ(n).

4.2. Optimization Impact

Assuming that “Optimized” in the CSV file means the version without I/O output in the loop, we can measure the effect of the proposed improvement.

n	Baseline time (ns)	Optimized time (ns)	Absolute Reduction	The effect
100	7,795,400	4,106,500	3,688,900	47.3% Decrease
10000	3,365,700	3,391,200	insignificant	~0%
100000	7,114,800	5,332,800	1,782,000	25.0% Decrease

Analysis: Optimization has a significant practical impact on execution time. This proves that O(1) I/O operations performed n times create a huge constant overhead, which was successfully eliminated, confirming the criticality of the recommendation to remove I/O.

5. Conclusion

5.1. Summary of Findings

The Kadan algorithm implemented by the partner is asymptotically perfect for the maximum subarray problem:

- Time Complexity: Θ(n)

- Space Complexity: O(1)

However, practical performance and code quality are suboptimal due to two factors:

- Critical Factor (Performance): Excessive use of System.out.println inside the loop, which creates significant constant overhead and distorts benchmark results.

- Quality Factor (Maintainability): Use of non-standard, redundant logic (hasPositive, maxElement) to handle edge cases, which makes the code complex and less clean than the canonical version.

5.2. Optimization Recommendations

To achieve the ideal balance between theoretical and practical effectiveness, it is recommended that:

- Remove all I/O (all System.out.println/printf calls) from the findMaxSubarray method. This will result in an immediate and significant performance gain (25–47%).

- Switch to the Canonical Form of Kadane, which does not require special variables to handle all negative numbers