# Individual Analysis Report — Kadane’s Algorithm

**Yerassyl Ibrayev** about **Albek Gusmanov’s** algorithm

## 1. Algorithm Overview

Kadane’s Algorithm finds the contiguous subarray with the maximum sum in a one-dimensional array. It uses dynamic programming by maintaining two variables:

• maxEndingHere: maximum subarray sum ending at the current position

• maxSoFar: overall maximum found so far

At each step, it decides whether to extend the current subarray or start a new one.

**Theoretical background: The algorithm exploits the optimal substructure property—each local maximum contributes to the global maximum—and requires only a single linear pass.**

## 2. Complexity Analysis

**Time Complexity:**

• Best Case (Ω(n)) – The algorithm always iterates once through all elements. Even if all numbers are positive, it still performs n comparisons.

• Average Case (Θ(n)) – For typical random input, the algorithm makes one comparison and a few constant operations per element.

• Worst Case (O(n)) – Even with alternating positive/negative values, it still performs a single pass. Thus, total time grows linearly with input size n.

**Space Complexity:**

• Only a few integer variables are used (maxSoFar, maxEndingHere, indices).

• O(1) auxiliary space.

• The implementation is in-place, requiring no additional arrays.

**Comparison with partner’s complexity: Both versions maintain O(n) time and O(1) space, confirming theoretical consistency.**

## 3. Code Review

**Inefficiencies identified:**

• Repeated tracking of arrayAccesses and comparisons slightly increases overhead in benchmarks.

• Some duplicate code between findMaximumSubarray and findMaximumSubarrayWithMetrics.

• Minor redundant increments of arrayAccesses before every conditional check.

**Optimization suggestions:**

• Combine metric tracking logic into a single reusable helper method.

• Remove duplicate code blocks to improve maintainability.

• Inline candidate1/candidate2 computations directly into comparisons for cleaner performance.

**Proposed complexity improvements:**

• Asymptotic time and space complexities cannot improve beyond O(n) and O(1).

• However, minor constant-factor optimizations (fewer counter updates and comparisons) can improve empirical speed by 5–10%.

## 4. Empirical Results

**Observed data (partner benchmarks):**

|  |  |
| --- | --- |
| **Input Size (n)** | **Avg Time (ns)** |
| 100 | 9650 |
| 500 | 31050 |
| 1000 | 44800 |

**Performance growth:**

|  |  |  |
| --- | --- | --- |
| **Input Growth** | **Expected Growth (O(n))** | **Observed Growth** |
| 100 → 500 | ×5 | ×3.2 |
| 500 → 1,000 | ×2 | ×1.44 |

Validation: Measured results grow roughly linearly, confirming O(n) behavior. Deviations arise from fixed overhead and JVM warm-up effects.

**Constant factors: The consistent per-element operations (2 comparisons + 2 array accesses) support the theoretical analysis.**

## 5. Conclusion

Kadane’s Algorithm efficiently achieves linear time and constant space complexity. The partner’s implementation is correct, structured, and metrically sound.

Suggested optimizations mainly target code clarity and minor constant-time reductions.

**Overall, empirical and theoretical analyses confirm that the algorithm performs as expected for ~large input sizes.**