Analysis Report: Kadane's Algorithm Implementation

1. Algorithm Overview

Kadane's Algorithm is an efficient method for finding the maximum sum of a contiguous subarray in a one-dimensional array of numbers. The algorithm runs in linear time and uses dynamic programming.

Main Idea:

- At each position i of the array, the maximum sum of a subarray ending at i (maxEndingHere) is calculated.
- If the current element is greater than the sum including previous elements, a new subarray starts.
- A global variable maxSoFar keeps track of the maximum sum so far.
- The start and end indices of the maximum subarray are updated dynamically.

Theoretical Background:

- Dynamic programming with memory optimization (only two values are stored instead of the full DP array).
- Applicable for arrays containing negative numbers.
- Guarantees finding the optimal subarray and is fundamental for substring problems, financial analysis (maximum profit), and other computer science tasks.

Integration with Metrics:

- CountedIntArray and PerformanceTracker allow performance analysis, including the number of comparisons, assignments, array accesses, and execution time.

2. Complexity Analysis

2.1 Time Complexity

Let n be the length of the array.

For each iteration of the loop (for i = 1 to n-1), the following operations are performed:

- get(i) — O(1)
- Comparison maxEndingHere + v < v — O(1)
- Assignment maxEndingHere = ... — O(1)
- Comparison maxEndingHere > maxSoFar — O(1)
- Assignment maxSoFar = ... — O(1)

In total:

- Each iteration — O(1)
- Total iterations — n
- **Overall time complexity:** O(n)

2.2 Space Complexity

- Auxiliary variables: maxEndingHere, maxSoFar, start, end, s — O(1)
- CountedIntArray stores a reference to the array — O(n) for data storage
- **Overall space complexity:** O(n) (mainly due to the CountedIntArray wrapper)

2.3 Best, Worst, and Average Cases

- Time complexity is the same for all cases — O(n)
- Space complexity is also O(n) in all cases

Case	Time Complexity	Space Complexity
Best	Θ(n)	Θ(n)
Worst	Θ(n)	Θ(n)
Average	Θ(n)	Θ(n)
Upper Bound	O(n)	O(n)
Lower Bound	Ω(n)	Ω(n)

2.5 Comparison with Partner's Algorithm

- If a partner uses a **brute-force approach** (double loop for all subarrays):

- Time: $O(n^2)$ or $O(n^3)$ in the worst case
- Space: O(1)

- Kadane's algorithm outperforms brute-force in terms of time complexity and is suitable for large arrays.

3. Code Review

3.1 Inefficient Sections

1. Duplication of the Kadane class in the code — can complicate maintenance.
2. Calling tracker.incAssignments() twice after declaring variables start, end, s — can be made more readable.
3. CountedIntArray adds overhead for each array element access.
4. BenchmarkRunner uses Thread.sleep(10), which may distort timing measurements.

3.2 Optimization Suggestions

- Remove duplicate Kadane code.
- Inline variables or combine assignments to reduce unnecessary calls to incAssignments().
- Use the primitive array directly without the wrapper for high-performance benchmarks.
- Add a reset method in PerformanceTracker to reuse a single object for multiple runs.
- Optimize the generation of the nearly-sorted array: minimize the number of random swaps for large arrays.

3.3 Proposed Improvements for Complexity

- Time complexity O(n) is already optimal — improvements are possible only by reducing constant factors (fewer tracking method calls).
- Space complexity can be reduced to O(1) by removing CountedIntArray and using the original array.

4. Empirical Results

4.1 Performance Plots

For different array sizes (100, 1,000, 10,000, 100,000):

- Execution time grows linearly with array size — confirms theoretical O(n) complexity.
- Metrics:
    - Comparisons: ~2*(n-1)
    - Assignments: ~3*(n-1) + 2
    - Array accesses: n

4.2 Validation of Theoretical Complexity

- Plots of elapsed_ms vs array size show linear growth, matching O(n) complexity.
- Constant factors are noticeable for small arrays due to the overhead of CountedIntArray and PerformanceTracker.

4.3 Analysis of Constant Factors

- Execution time for array size 100,000 ≈ 5–10 ms (depends on JVM and hardware).
- Main overhead — calls to incAssignments(), incComparisons(), and get().
- For production, tracking can be removed to minimize constant-factor overhead.

5. Conclusion

Findings:

- The algorithm correctly finds the maximum sum subarray in all tested cases.
- The theoretical linear complexity is confirmed by empirical measurements.
- PerformanceTracker metrics provide detailed analysis of operations.
- Kadane's algorithm is significantly more efficient than brute-force approaches.

Optimization Recommendations:

1. Remove duplicate code and unnecessary tracking calls to reduce overhead.
2. For large arrays, use the primitive array without CountedIntArray.
3. Add a reset method to PerformanceTracker for multiple runs.
4. Optimize the generation of test data, especially nearly-sorted arrays.

Overall Verdict:
The project is ready for analysis and learning purposes. The code structure is modular and extensible. The algorithm is time-optimal, and tracking allows for performance study at a micro level.