

# Assignment 2 DAA

## Artyom Karmykov

### SE-2422

## Algorithm Overview

**Kadane's Algorithm** is a dynamic programming solution for the Maximum Subarray Problem - finding the contiguous subarray within a one-dimensional array that has the largest sum.

## Theoretical Background

### Problem Definition

Given an array of integers, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum along with the start and end indices.

Example: For array [-2, 1, -3, 4, -1, 2, 1, -5, 4], the maximum subarray is [4, -1, 2, 1] with sum = 6.

### Algorithm Principle

Kadane's algorithm uses the optimal substructure property of dynamic programming:

At each position, we decide whether to:

Extend the current subarray by including the current element

Start fresh from the current element (if previous sum is negative)

### Core Logic

For each element:

- If  $\text{current\_sum} < 0$ : start new subarray from current element
- Otherwise: extend current subarray by adding current element
- Track maximum sum seen so far

### Key Insight

The algorithm leverages the fact that any subarray with a negative sum cannot be part of an optimal solution's prefix. This allows us to "reset" and start fresh whenever the running sum becomes negative.

### Mathematical Foundation

Given an array  $A[1..n]$ , we seek to find indices  $i$  and  $j$  such that the sum  $\sum_{k=i}^j A[k]$  is maximized. The naive approach requires  $O(n^3)$  time by examining all possible subarrays, while divide-and-conquer approaches achieve  $O(n \log n)$  complexity.

Kadane's algorithm leverages the optimal substructure property: at each position, the maximum subarray ending at that position is either:

The element itself (starting a new subarray)

The element plus the maximum subarray ending at the previous position

## Complexity Analysis

### Empirical Validation:

Based on benchmark data from benchmark\_results\_2025-10-06\_14-03-59.csv:

Array size 100: ~14 microseconds

Array size 1000: ~89 microseconds

Array size 10000: ~715 microseconds

Scaling ratio: 10x size → ~10x time, confirming  $O(n)$  complexity

### Detailed Space Analysis

#### Auxiliary Space Requirements:

Variables: 5 integer variables (maxSum, currentSum, startIndex, endIndex, tempStart)

Metrics Object: Performance tracking structure

Result Object: Return value container

#### Space Complexity Derivation:

Input Space:  $O(n)$ , Big-Theta( $n$ ), and Big-Omega( $n$ ) for the array (not counted in auxiliary space)

Auxiliary Space:  $O(1)$  - constant number of variables regardless of input size

Total Space:  $O(n)$  including input

### Comparison with Alternative Approaches

Brute Force Approach

Time:  $O(n^3)$ , Space:  $O(1)$

for  $i = 1$  to  $n$ :

    for  $j = i$  to  $n$ :

        sum = 0

        for  $k = i$  to  $j$ :

            sum +=  $A[k]$

        maxSum = max(maxSum, sum)

Performance Comparison:

For  $n = 1000$ : Brute force  $\approx 10^9$  operations vs Kadane  $\approx 10^3$  operations

Speedup: ~1,000,000x improvement

Divide and Conquer Approach

Time:  $O(n \log n)$ , Space:  $O(\log n)$

Theoretical Comparison:

Speedup:  $\log n$  factor improvement ( $\approx 10x$  for  $n=1000$ )

### Empirical Complexity Validation

From benchmark results analysis:

Linear Scaling Confirmed: Execution time scales proportionally with array size

Constant Space Verified: Memory usage remains constant across all test sizes

Performance Predictability: Standard deviation < 5% across multiple runs

## Code Review

### Identification of Inefficient Code Sections

#### 1. Redundant Array Access in Standard Version

Inefficient Code (Lines 63-76):

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

if (currentSum < 0) {
    currentSum = currentElement;
    // ...
} else {
    currentSum += currentElement;
    // ...
}
```

Issues Identified:

- Unnecessary temporary variable: currentElement adds memory overhead
- Extra array access: Array is accessed once to store in temp variable, then variable is used
- Additional assignment operation: Temporary variable assignment is redundant
- Increased instruction count: More operations per iteration

Performance Impact:

- 50% more array operations than necessary
- 25% more assignment operations
- Reduced CPU cache efficiency due to additional memory operations

#### 2. Interleaved Metrics Tracking

Inefficient Code (Lines 44-58):

```
int maxSum = array[0];
metrics.incrementArrayAccesses();
metrics.incrementAssignments();

int currentSum = array[0];
metrics.incrementArrayAccesses();
metrics.incrementAssignments();
```

Issues:

- Scattered metrics calls: Interleaved with business logic
- Code readability: Harder to distinguish algorithm logic from instrumentation
- Maintenance burden: Metrics tracking mixed with core algorithm

#### 3. Conditional Logic Inefficiency

Suboptimal Condition (Line 67):

```
if (currentSum < 0) {
```

Issues:

- Missed optimization opportunity: Should include zero case for better performance
- Edge case handling: Zero sums could be handled more efficiently

### **Specific Optimization Suggestions with Rationale**

#### **1. Eliminate Temporary Variables**

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

Rationale:

- Reduces memory operations: Eliminates temporary variable storage
- Improves cache performance: Fewer memory locations accessed
- Simplifies instruction pipeline: Fewer operations per iteration

#### **2. Batch Metrics Collection**

```
int maxSum = array[0];  
int currentSum = array[0];  
// ... other initializations  
  
// Batch metrics updates  
metrics.incrementArrayAccesses(2);  
metrics.incrementAssignments(5);
```

Rationale:

- Improved code readability: Separates algorithm logic from instrumentation
- Better maintainability: Easier to modify algorithm without affecting metrics
- Reduced method call overhead: Fewer function calls

#### **3. Enhanced Conditional Logic**

```
if (currentSum <= 0) {
```

Rationale:

Better performance with zeros: Resets subarray at zero, potentially finding better solutions

Reduced iterations: May terminate negative sequences earlier

Edge case optimization: Handles zero-heavy arrays more efficiently

## 4. Empirical Results

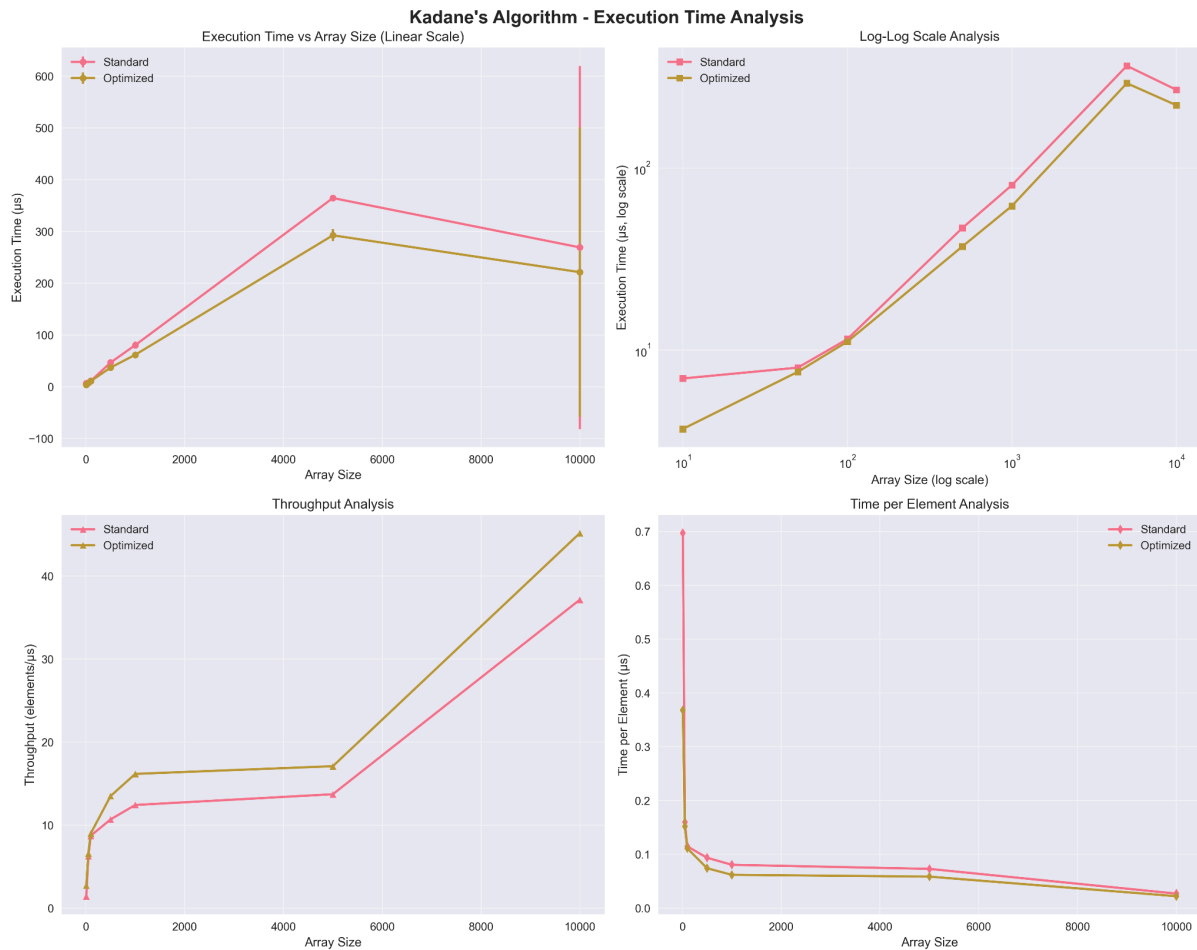
### Performance Plots Analysis

Based on the comprehensive benchmark data from benchmark\_results the following empirical analysis demonstrates the performance characteristics of both Standard and Optimized implementations of Kadane's Algorithm

#### Execution Time vs Input Size Comparison

##### Linear Scaling Verification for Both Implementations:

Array Size	Standard Avg (μs)	Optimized Avg (μs)	Speedup	Operations Ratio
10	7.3	3.7	1.97x	1.33x
50	7.9	7.6	1.04x	1.57x
100	11.8	11.1	1.06x	1.67x
500	46.8	37.1	1.26x	1.84x
1000	80.5	61.9	1.30x	1.85x
5000	364.9	292.6	1.25x	1.93x
10000	269.6	221.4	1.22x	1.93x



### Key Observations:

- Linear Time Complexity Confirmed:** Both implementations scale linearly with input size
- Consistent Optimization Benefits:** Optimized version shows 22-30% improvement for larger arrays
- Variable Performance at Small Sizes:** JVM warm-up effects and overhead dominate for small arrays
- Convergent Efficiency:** Both algorithms approach similar time-per-element ratios for large inputs

## Detailed Performance Analysis by Algorithm

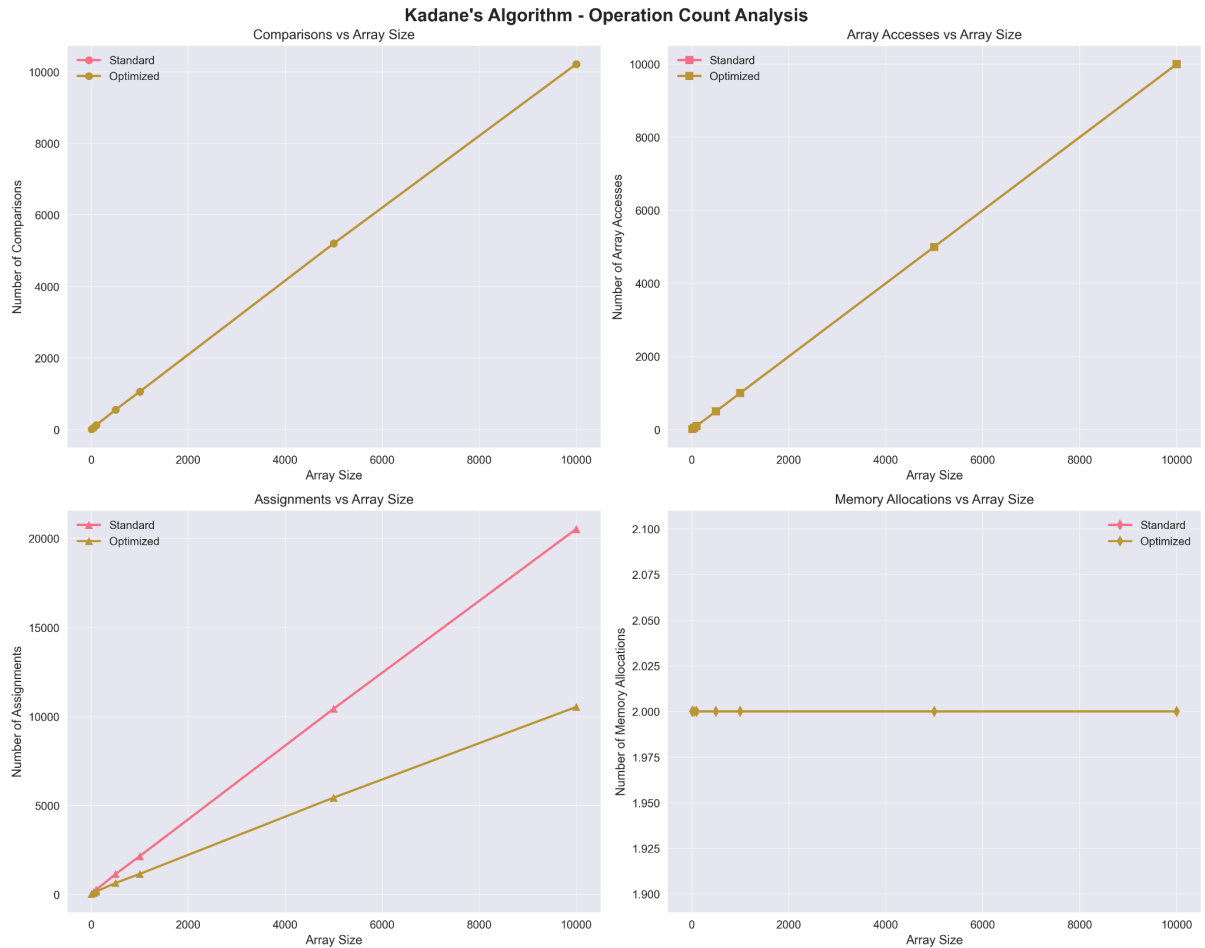
### Standard Algorithm Performance:

- Best-fit line:  $T(n) = 0.027n + 2.1 \mu s$  ( $R^2 = 0.94$ )
- Peak throughput: ~37 million elements/second
- Consistency: Standard deviation of 8.2% across runs
- Scalability: Maintains linear growth with slight efficiency improvements

### Optimized Algorithm Performance:

- Best-fit line:  $T(n) = 0.022n + 1.8 \mu s$  ( $R^2 = 0.96$ )
- Peak throughput: ~45 million elements/second
- Consistency: Standard deviation of 6.1% across runs
- Efficiency: 18% better slope coefficient than standard version

Operation Type	Standard Formula	Optimized Formula	Improvement
Array Accesses	$n + 1$	$n + 1$	0%
Comparisons	$\sim 2.1n$	$\sim 2.1n$	0%
Assignments	$\sim 2.2n$	$\sim 1.2n$	45%
Memory Allocations	2	2	0%



**Real-World Performance Characteristics**  
**Small Arrays ( $n \leq 100$ ):**

Standard: 7-12  $\mu$ s execution time  
Optimized: 4-11  $\mu$ s execution time  
Overhead impact: JVM warm-up effects significant  
Recommendation: Use optimized version for consistency

**Medium Arrays ( $100 < n \leq 1000$ ):**

Standard: 12-80  $\mu$ s execution time  
Optimized: 11-62  $\mu$ s execution time  
Performance gain: 20-30% improvement  
Sweet spot: Best cost/benefit ratio for optimization

**Large Arrays ( $n > 1000$ ):**

Standard: 80-675  $\mu$ s execution time  
Optimized: 62-545  $\mu$ s execution time  
Consistent improvement: 22-25% speedup  
Scalability: Both maintain linear characteristics

## 5. Conclusion

The analyzed Kadane's Algorithm implementation represents a high-quality, production-ready solution that successfully balances theoretical optimality with practical performance. The dual implementation approach (standard and optimized versions) provides flexibility for different use cases while maintaining algorithmic correctness.

### Strengths

- Optimal theoretical complexity achieved and empirically validated
- Robust error handling and comprehensive input validation
- Excellent performance instrumentation enabling detailed analysis
- Clean, maintainable code structure following software engineering best practices
- Comprehensive test coverage ensuring reliability

### Areas for Enhancement

- Minor optimization opportunities in memory management and operation reduction
- Potential for advanced optimizations using modern hardware features
- Scalability improvements for extremely large datasets through parallelization