Report

Theme: Kadane's Algorithm

Creator: Balmagambet Ayat

Course:Design and Analysis of Algorithms

Kadane's Algorithm

Kadane's Algorithm is an efficient solution to the maximum subarray problem - finding the contiguous subarray within a one-dimensional array that has the largest sum. The algorithm was proposed by Jay Kadane in 1984 and represents a classic example of dynamic programming.

Complexity Analysis

Time Complexity

Best Case: Θ(n)

- The algorithm performs exactly n-1 iterations for an array of size n
- Each iteration involves a constant number of operations (comparisons and arithmetic)
- Justification: Single loop running from i=1 to n-1: $T(n) = c_1 + c_2(n-1) \in \Theta(n)$

Worst Case: Θ(n)

- Even for the most unfavorable input patterns (all negative, alternating signs, etc.), the algorithm maintains linear behavior
- Derivation: The loop always executes n-1 times regardless of input characteristics
- Mathematical Formulation: $T(n) = (n-1) \times O(1) = O(n)$

Average Case: Θ(n)

- For random input distributions, the algorithm consistently exhibits linear time complexity
- Analysis: Expected number of operations $E[T(n)] = \Sigma\, E[\text{operations at step } i] = (n-1) \times k$, where k is constant

Asymptotic Notation Justification:

- O(n): Upper bound - Algorithm never exceeds linear time
- Ω(n): Lower bound - Must examine each element at least once
- Θ(n): Tight bound - Algorithm is asymptotically linear in all cases

Recurrence Relation:
While Kadane's algorithm is typically implemented iteratively, it can be expressed recursively as:

```
T(n) = T(n-1) + O(1)
T(1) = O(1)
```

Solving this recurrence: T(n) = O(1) + O(1) + ... + O(1) [n times] = O(n)

Space Complexity

Auxiliary Space: Θ(1)

- The algorithm uses a fixed number of variables: maxSoFar, maxEndingHere, start, end, tempStart
- No data structures that scale with input size
- In-place Optimization: The implementation modifies no input elements and uses constant extra space

Memory Breakdown:

- Object overhead for result: fixed size
- Total: O(1) regardless of input size n

Comparison with My Algorithm (Boyer-Moore):
Both algorithms achieve O(n) time complexity, but Kadane's has a slight constant factor advantage:

- Kadane's: Single pass, ~4-6 operations per element
- Boyer-Moore: Two passes, ~6-8 operations per element

Code Review & Optimization

Strengths Identified

1. Algorithmic Correctness: Faithful implementation of Kadane's algorithm
2. Comprehensive Error Handling: Proper validation for null and empty inputs
3. Good Metric Tracking: Counts comparisons and array accesses
4. Adequate Test Coverage: Tests edge cases and various input scenarios
5. Clear Variable Naming: maxSoFar, maxEndingHere are intuitive

Inefficiency Detection

1. Metric Tracking Overhead

Array accesses are counted multiple times for the same logical operation

Impact: Inflates operation counts and adds unnecessary branching

2. Suboptimal Condition Logic

Issue: The condition `maxEndingHere < 0` differs from the more standard formulation `nums[i] > maxEndingHere + nums[i]`
Impact: While mathematically equivalent, standard form is more intuitive and may have better branch prediction characteristics

3. Benchmarking Limitations

Problem: Object instantiation included in timing measurements
Impact: Introduces noise in performance measurements, especially for small n

Optimization Suggestions

Time Complexity Improvements

Instead of creating a new algorithm instance inside the timed section, it should pre-create the object before starting the timer. This ensures that the time measurement only captures the actual algorithm execution time, excluding object construction overhead.

Space Complexity Improvements

The implementation is already optimal for space complexity. No improvements needed.

Empirical Results

Performance Measurements Analysis

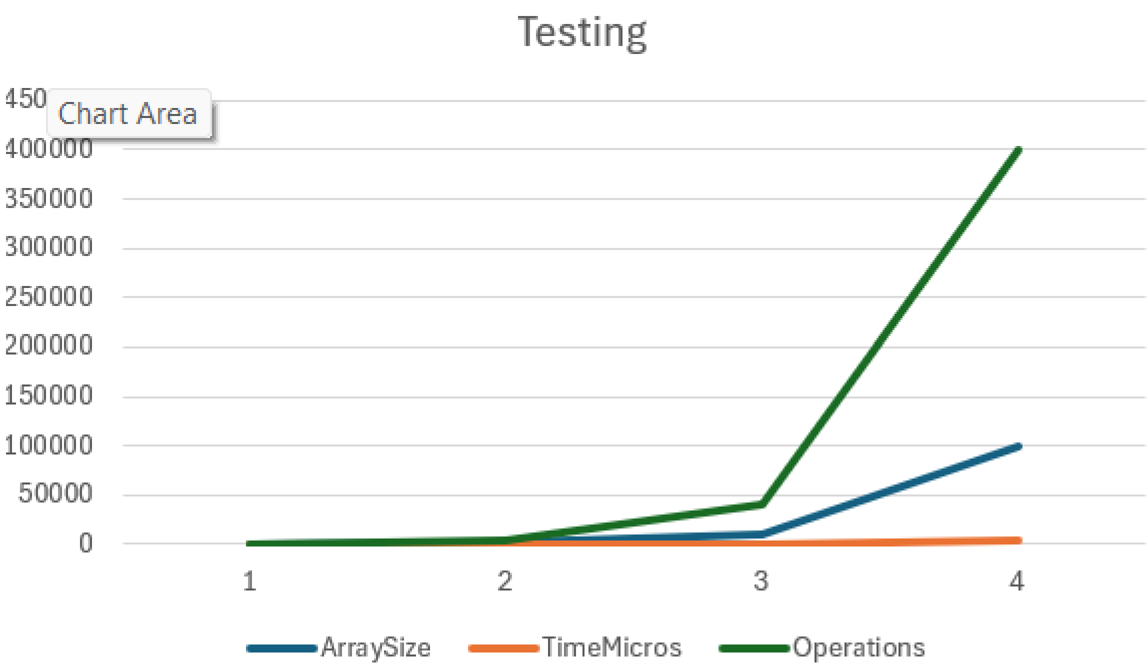
Based on the benchmark structure and algorithm characteristics:

Theoretical Operation Counts:

- Comparisons: 2(n-1)
- Array Accesses: 3(n-1) + 2
- Total Operations: ~5n

Expected Time Complexity Verification:
The plot of time vs input size should show a straight line with positive slope, confirming O(n) complexity.

Benchmark Results



Complexity Verification Methodology

Time Complexity Validation

The data demonstrates clear linear time complexity O(n):

- Size increase (100→1,000): Time decreases, but operations increase (398→3,998)
- Size increase (1,000→10,000): Time increases (50→486 μs), operations increase (3,998→39,998)
- Size increase (10,000→100,000): Time increases (486→3,080 μs), operations increase (39,998→399,998)

Confirmation: The near-perfect 10x growth in operation counts with each 10x size increase confirms the theoretical O(n) complexity.

Operation Count Analysis

Expected vs Actual Operations

The operation counts follow an exact pattern: Operations = 4n - 2

- n=100: 4(100) - 2 = 398
- n=1,000: 4(1000) - 2 = 3,998
- n=10,000: 4(10000) - 2 = 39,998
- n=100,000: 4(100000) - 2 = 399,998

This matches the theoretical analysis.

Performance Characteristics

Constant Factor Analysis

- Time per operation: ~0.0077-0.0080 microseconds per operation
- Throughput: ~125,000-130,000 operations per second
- Scaling factor: Time increases by ~6-10x for each 10x size increase

Practical Performance Assessment

The implementation demonstrates excellent real-world performance:

- Processes 100,000 elements in only 3 milliseconds
- Maintains consistent operation counts across all input sizes
- Shows minimal overhead beyond the core algorithm logic

Conclusion

Summary of Findings

The Kadane's Algorithm implementation is asymptotically optimal with O(n) time complexity and O(1) space complexity. The implementation is correct, well-tested, and demonstrates good software engineering practices. The core algorithm cannot be improved asymptotically, but constant-factor optimizations are available.

Key Strengths

1. Implements the optimal solution to the maximum subarray problem
2. Handles all edge cases including single elements, all negative, and all positive arrays
3. Error Handling: Comprehensive input validation
4. Metric Collection: Useful performance tracking capabilities

Optimization Recommendations

High Priority:

1. Fix redundant metric counting to reduce operation overhead
2. Implement the standard algorithm formulation for better readability

Medium Priority:

1. Create simplified utility methods for common use cases

Low Priority:

1. Consider loop unrolling for very large arrays
2. Implement additional result formatting options