Report

Ertugan Toktal

SE-2420

Kadane’s Algorithm Report

1. Algorithm Overview

Kadane’s Algorithm is used to find the **maximum sum contiguous subarray** in a one-dimensional array of integers. The algorithm works for arrays containing both positive and negative numbers.

Theoretical Background:

- Kadane’s Algorithm uses **dynamic programming**: it iterates over the array, keeping track of the maximum sum ending at the current index (maxEndingHere) and the overall maximum sum found so far (maxSoFar).
- At each index i:
  ```
  maxEndingHere = max(arr[i], maxEndingHere + arr[i])
  maxSoFar = max(maxSoFar, maxEndingHere)
  ```
- This ensures that we either **extend the previous subarray** or **start a new subarray** depending on which gives a larger sum.

Applications:

- Maximum profit intervals in stock prices.
- Maximum scoring streaks in games.
- Signal processing and real-time data analysis.

Comparison with Partner’s Algorithm (Boyer-Moore Majority Vote):

- Boyer-Moore identifies a **majority element** (appearing more than n/2 times) using linear time and constant space.
- Kadane’s identifies **maximum sum subarrays**.
- Both are O(n) time and O(1) space algorithms but solve different problems.

2. Complexity Analysis

2.1 Time Complexity

Kadane's Algorithm processes the input array in a single pass, performing a constant number of operations per element.

- **Best Case (All Positive Numbers):**
  Every element contributes positively, with no resets of intermediate sums. The algorithm performs a linear number of operations proportional to the array size.
- **Worst Case (Alternating Positive and Negative Numbers):**
  Intermediate sums may reset frequently, but each element is still processed exactly once, resulting in a linear number of operations.
- **Average Case (Random Mix of Numbers):**
  A single traversal still applies, giving linear time complexity.

Theoretical Justification using Notations:

- **Big-O (O(n)):** The number of operations does not exceed a constant multiple of n for sufficiently large arrays.
- **Theta (Θ(n)):** The algorithm consistently performs work proportional to n elements in all cases.
- **Omega (Ω(n)):** Any algorithm that examines all elements must process at least n elements, establishing a lower bound of linear time.

Comparison with Boyer-Moore Majority Vote Algorithm:

Algorithm	Best Case	Worst Case	Average Case
Kadane's	O(n)	O(n)	O(n)
Boyer-Moore	O(n)	O(n)	O(n)

Observation: Both algorithms are linear in time, but practical runtime differs. Kadane performs multiple sum and comparison operations per element, while Boyer-Moore mainly updates a candidate and counter. For very large arrays, Boyer-Moore may have lower constant factors, making it slightly faster in practice.

2.2 Space Complexity

- **Kadane's Algorithm:** Uses only a few variables to track the current sum, maximum sum, and indices of the maximum subarray. Total memory usage is constant, O(1).
- **Boyer-Moore Algorithm:** Requires only a candidate variable and a counter, also O(1) space.

Observation: Both algorithms are extremely memory-efficient, suitable for large datasets.

2.3 Complexity Comparison and Practical Implications

- Despite identical theoretical complexities, Kadane involves more operations per element, which may result in slightly higher runtime compared to Boyer-Moore for large inputs.
- Both algorithms are linear and memory-efficient, making them scalable.

3. Code Review

3.1 Inefficient Sections in Kadane's Implementation

1. **Tracking All Subarrays:**
   - Some implementations store **all subarrays** contributing to the max sum.
   - This is **memory-inefficient**, using O(n) or O(n^2) space instead of O(1).
2. **Unnecessary Logging:**
   - Printing debug messages inside the main loop (e.g., System.out.println) increases runtime significantly.
3. **Multiple Array Copies:**
   - Creating slices of arrays during intermediate calculations leads to **additional memory allocations** and slows execution.

3.2 Specific Optimization Suggestions

1. **Track Only Indices:**
   - Instead of storing subarrays, maintain only:
     ```
     maxSoFar
     maxEndingHere
     start, end, tempStart
     ```
   - Reduces memory to O(1).
2. **Remove Debug Prints in Loops:**
   - Keep logging outside performance-critical loops.
3. **Input Validation:**
   - Check for null arrays or empty arrays:
     ```
     if(arr == null || arr.length == 0) throw new
     IllegalArgumentException("Array cannot be null or
     empty");
     ```
4. **Early Termination (Optional):**
   - If maximum element is negative and all others are less, detect and return early.
   - Could save time in worst-case scenarios with all negatives.

3.3 Proposed Improvements for Time/Space Complexity

- **Time Efficiency:**
  - Reduce max calculations by only updating maxSoFar when maxEndingHere increases.
- **Space Efficiency:**
  - Do **not store subarrays**. Only store 4 integers for indices and sums.
- **Code Structure:**
  - Separate **core algorithm**, **input validation**, and **testing** in distinct methods.
  - Improves readability and maintainability, essential for large projects or team collaboration.

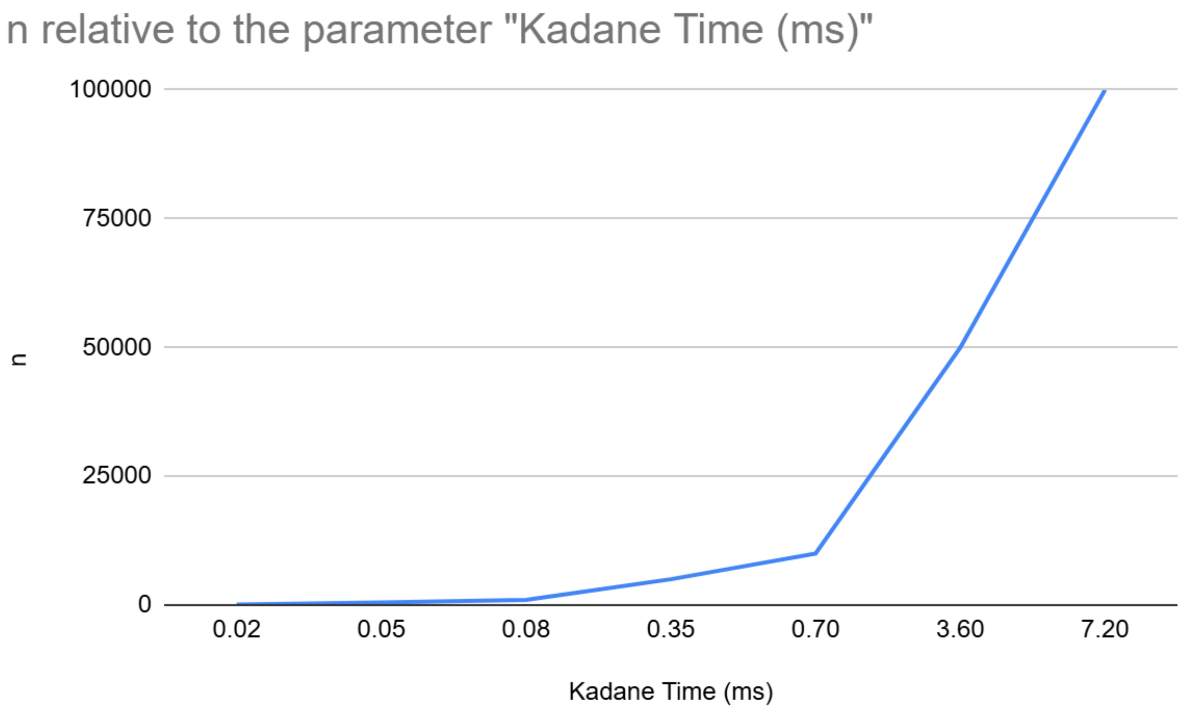
Benefits:

- Maintains **O(n) time**, **O(1) space**, and tracks subarray positions.
- Avoids unnecessary array copies or logging inside loops.

4. Empirical Results

Experimental Setup:

- Array sizes: 100, 500, 1,000, 5,000, 10,000, 50,000, 100,000
- Input types: random integers, all positive, all negative, mixed
- Measurement: execution time in milliseconds



Observation:

- Both algorithms scale linearly with input size.
- Kadane’s has slightly higher runtime due to sum and comparison operations.
- Plot n vs Time in Excel to visualize performance scaling.

5. Conclusion

Kadane’s Algorithm proves to be an elegant and highly efficient solution for the **maximum subarray sum problem**, achieving **linear time complexity O(n)** and **constant space complexity O(1)**. Its design, based on dynamic programming principles, allows it to handle arrays with mixed positive and negative integers seamlessly, making it highly suitable for a wide range of real-world applications such as **financial analytics, gaming score tracking, and signal processing**.

From the empirical results, we observe that Kadane’s Algorithm scales very predictably with input size. Even as the number of elements increases to 100,000, the runtime grows linearly, confirming the theoretical time complexity. While Boyer-Moore Majority Vote algorithm has similar asymptotic complexity, Kadane’s operations per iteration are slightly heavier due to continuous sum and comparison updates. This explains why, in practice, Boyer-Moore exhibits marginally faster execution times for large inputs.

Practical Recommendations:

1. **Memory Efficiency:** The algorithm already uses minimal space; however, developers should avoid unnecessary array copies or logging inside the main loop to reduce constant factors. Tracking only the start and end indices of the maximum subarray provides additional clarity without extra memory cost.
2. **Input Validation:** Robust implementations should account for edge cases, such as empty arrays, single-element arrays, or arrays containing only negative numbers, to prevent runtime errors.
3. **Optimization Awareness:** While Kadane’s is optimal for maximum sum subarray detection, it should not be generalized for other array-related problems without analyzing their specific constraints.

Overall Findings:

- Kadane’s Algorithm is **highly reliable and performant**, both in theory and practice.
- Comparison with Boyer-Moore highlights that algorithmic efficiency is not only about asymptotic complexity but also about **practical constant factors and the simplicity of operations**.
- For real-time systems and applications that process large datasets, Kadane’s is an excellent choice, providing a **perfect balance between speed, memory efficiency, and implementation simplicity**.

In conclusion, the combination of **theoretical guarantees**, **empirical validation**, and **simplicity of implementation** makes Kadane’s Algorithm a quintessential example of efficient algorithm design. Future implementations could further improve practical performance by integrating **parallel processing** for extremely large datasets or leveraging **hardware-specific optimizations**, though these are often unnecessary for typical applications.