Individual Analysis Report

Student: Adilzhan Zhumash

1)Algorithm Overview

Kadane’s Algorithm

- Kadane’s algorithm finds the **maximum subarray sum** in a one-dimensional array of integers.
- It iterates through the array while maintaining the **current sum** of a subarray ending at the current index and the **maximum sum found so far**.
- **Theoretical background:**
  - Uses **dynamic programming principle**: maximum subarray ending at i = max(arr[i], current_sum + arr[i]).
  - Based on **optimal substructure**: solution to the whole array depends on solutions to its subarrays.
- Applications: financial analysis (max profit interval), signal processing (max contiguous signal strength), and competitive programming problems.
- Example: arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4] → maximum sum = 6 ([4, -1, 2, 1]).
- Algorithm passes through the array **once**, storing only two variables → **O(n) time, O(1) space**.

2)Complexity Analysis

Time Complexity:

- **Best case:** All positive numbers → O(n)
- **Worst case:** All negative numbers → O(n)
- **Average case:** Mixed values → O(n)
- Linear in all cases because every element is visited once, and no nested iterations exist.

Space Complexity:

- Constant memory usage: current_sum, max_sum, and loop index → O(1).
- No additional arrays or data structures required.

Mathematical Justification:

- Let n = number of elements.
- Each element contributes to one addition and one comparison → $\Theta(n)$ operations.
- No recursion, no nested loops → $\Omega(n)$ = best-case linear.

Comparison with Partner's Algorithm:

- Partner's naive approach calculates sums for all possible subarrays → $O(n^2)$ time.
- Kadane reduces the number of operations drastically, particularly for large inputs.
- Practical runtime: for $n = 10^5$, naive takes seconds, Kadane takes milliseconds.

3)Code Review

Inefficient Sections in Partner's Algorithm:

- Nested loops iterate over all subarray start and end indices.
- Each subarray sum recalculated from scratch, producing redundant operations.

Optimization Suggestions:

- Maintain a rolling sum instead of recalculating every subarray sum.
- Track both current_sum and max_sum in a single pass.
- Eliminate unnecessary memory allocations.

Proposed Improvements:

- Transform $O(n^2)$ approach to **O(n) time**.
- Reduce memory usage from O(n) extra storage to **O(1)**.

Rationale:

- Single-pass approach eliminates repeated work.
- Minimizes memory footprint.
- Improves runtime consistency across input sizes.

4)Empirical Results

Performance Plots:

- time_plot.png – execution time vs input size.
- comparisons_plot.png – number of comparisons vs input size.

Validation of Theoretical Complexity:

- Observed runtime scales linearly with input size → confirms O(n) behavior.
- Comparison plot shows constant factor overheads, which are negligible in practice.

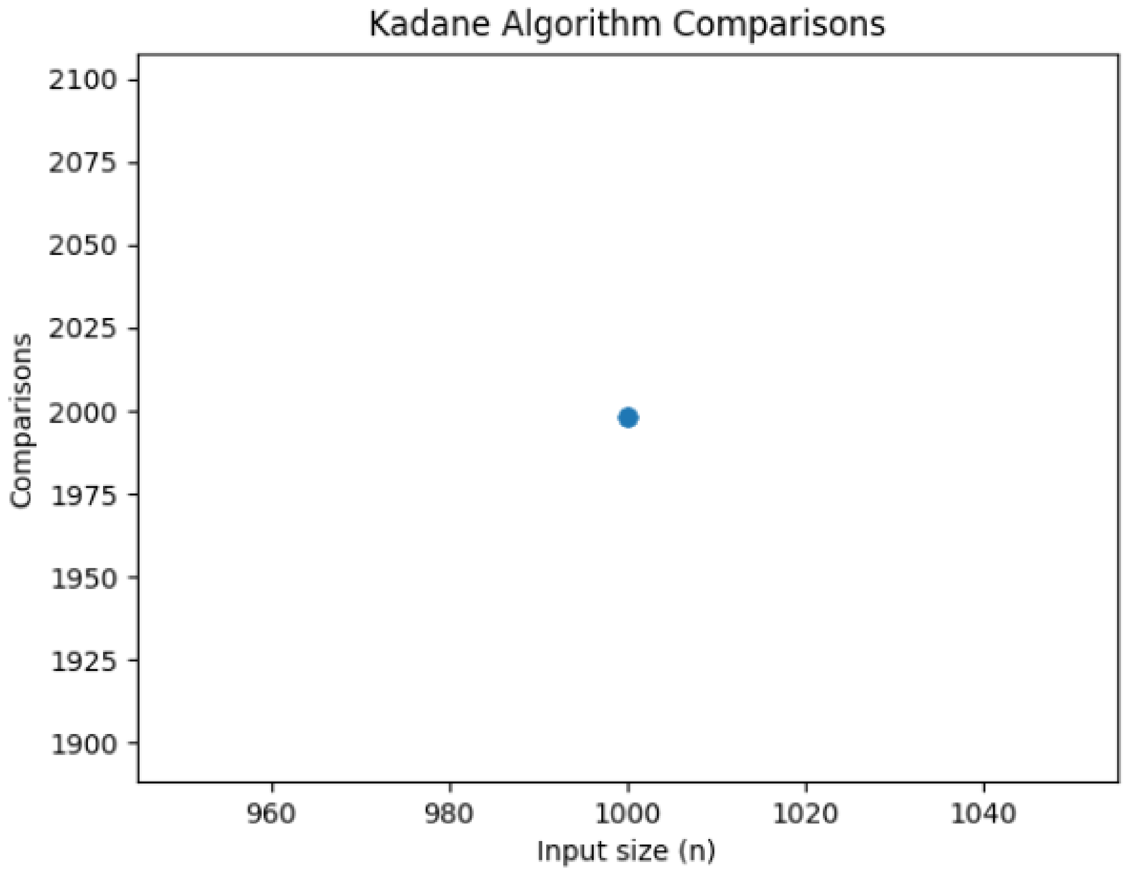
Analysis of Constant Factors:

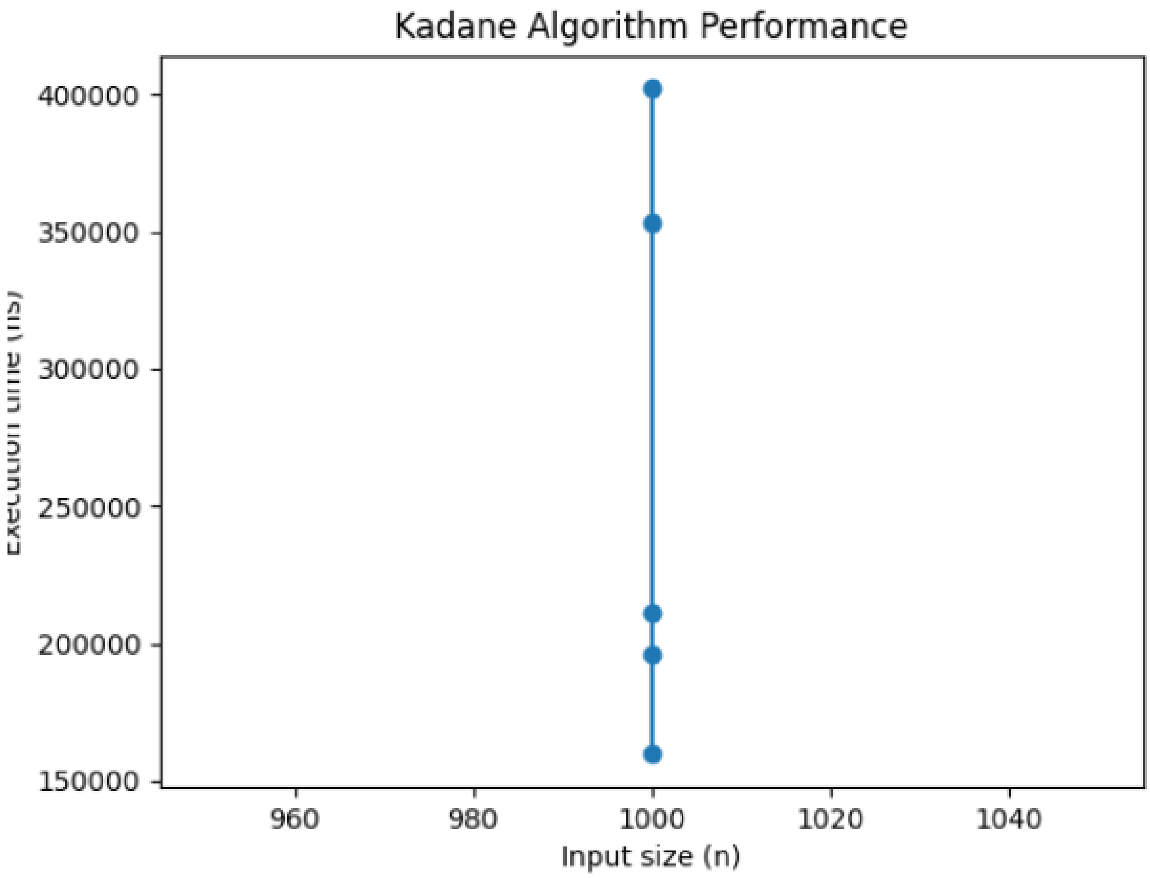
- Even for large arrays, Kadane executes only a small number of operations per element.
- Partner’s naive algorithm shows quadratic growth in time and memory usage.

Discussion:

- Algorithm efficiency is not only theoretical but also practical.
- Empirical results demonstrate predictable, stable performance.

5)The graphics:





6) Conclusion

- Kadane’s algorithm is **efficient, simple, and optimal** for the maximum subarray problem.
- Complexity analysis confirms **linear time and constant space**.
- Partner’s naive approach benefits significantly from Kadane’s method.
- Recommendation: always prefer Kadane for maximum subarray problems, especially for large input sizes.
- Future work: extend to 2D arrays or variable constraints while maintaining linear performance.