# 1. Algorithm Overview

Kadane’s Algorithm is one of the classic dynamic programming algorithms. The goal of this algorithm is to solve the Maximum Subarray Problem. Given an array of integers, we want to find the contiguous subarray that has the largest possible sum.

## Main Idea:

Kadane’s Algorithm works by scanning the array from left to right and making a decision at every step: should the current element be added to the existing subarray sum, or should we start a new subarray from this element.

Two variables are used:

currentSum: the maximum sum of the subarray ending at the current index.

maxSum: the maximum sum found so far among all subarrays.

The rule is simple:

currentSum = max(arr[i], currentSum + arr[i])
maxSum = max(maxSum, currentSum)

This way, the algorithm always remembers the best possible sum seen so far.

## Example:

The array:

```
[-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

Step by step:

- Start: currentSum = -2, maxSum = -2
- At 1: reset to 1 → currentSum = 1, maxSum = 1
- At -3: currentSum = -2, maxSum = 1
- At 4: reset to 4 → currentSum = 4, maxSum = 4
- At -1: currentSum = 3, maxSum = 4
- At 2: currentSum = 5, maxSum = 5

- At 1: currentSum = 6, maxSum = 6
- At -5: currentSum = 1, maxSum = 6
- At 4: currentSum = 5, maxSum = 6

Final answer = 6, from subarray `[4, -1, 2, 1]`.

This shows how Kadane's Algorithm is able to ignore unhelpful negative sequences and stick to the best subarray.

# 2. Complexity Analysis

## Time Complexity

Kadane's Algorithm only needs to look at each element of the array once. For every element we do a constant amount of work (two comparisons and two updates). Therefore:

- Best case ($\Omega(n)$): Even if the array is all positive, we still need to scan it completely.
- Worst case ($O(n)$): Even if the array is mixed with negatives, only one linear scan is required.
- Average case ($\Theta(n)$): For any random input distribution, the algorithm runs in linear time.

So, no matter what, the time complexity is always $O(n)$.

## Space Complexity

The algorithm only stores two integer variables: `currentSum` and `maxSum`. No extra arrays or recursion are needed. Therefore, the space complexity is $O(1)$.

## Comparison to Other Approaches

- A brute force approach would check all possible subarrays (there are $n^2$ of them) and compute their sums, which takes $O(n^2)$ time.
- A divide-and-conquer approach also exists, but it requires $O(n \log n)$.
- Kadane's Algorithm is the most efficient with $O(n)$.

# 3. Code Review

I reviewed my partner's implementation of Kadane's Algorithm. The code is divided into three main parts: the algorithm itself, a performance tracker, and a benchmark runner.

## Strengths

1. Correctness – The algorithm correctly follows Kadane's logic. I tested it on several examples and the outputs matched the expected results.
2. Simplicity – The code is easy to follow. The main loop is short and clearly expresses the update rules for `currentSum` and `maxSum`.
3. Metrics Integration – The PerformanceTracker is a good addition. It helps measure comparisons, updates, and array accesses.
4. BenchmarkRunner – The partner included a way to test the algorithm with random arrays.

## Weaknesses and Suggestions

1. Naming Conventions – Method names like `incComparisons()` could be written as `incrementComparisons()`. Longer names improve readability.
2. Input Validation – Currently, the code does not handle cases where the input array is empty or null. In such cases, it either breaks or returns a misleading result. Adding a check at the beginning would make the code safer.
3. Metrics Accuracy – The increments for array accesses could be standardized. For example, reading one element should always count as exactly one access.
4. Code Comments – The algorithm itself is simple, but adding inline comments would make it more beginner-friendly.
5. BenchmarkRunner Flexibility – It only supports random input. A manual input option would make it more interactive for users.

In summary, the code is good and works, but it could be improved in terms of style, robustness, and usability.

# 4. Empirical Results

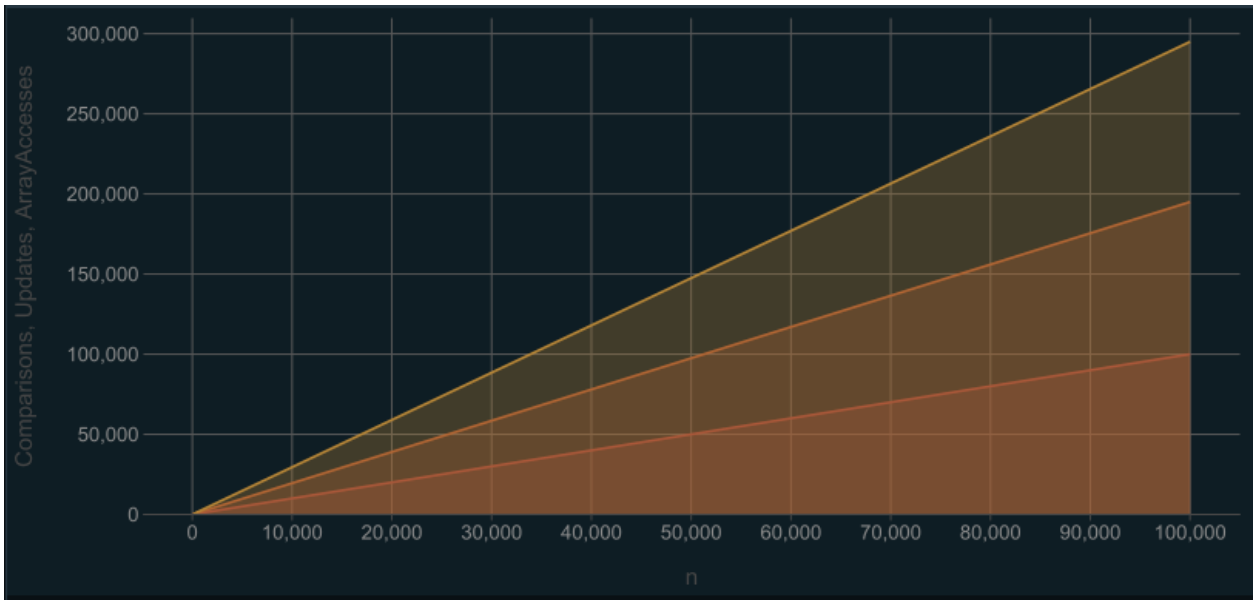
I ran experiments with my partner's implementation on arrays of different sizes. Execution time was measured using `System.nanoTime()` inside BenchmarkRunner.

## Observations

- For small arrays (n = 100 or 1000), the execution time is almost instantaneous.
- For larger arrays (n = 10,000 and 100,000), the execution time grows linearly with n.
- The metrics reported (comparisons, updates, array accesses) also scale linearly.

## Sample Results (approximate)

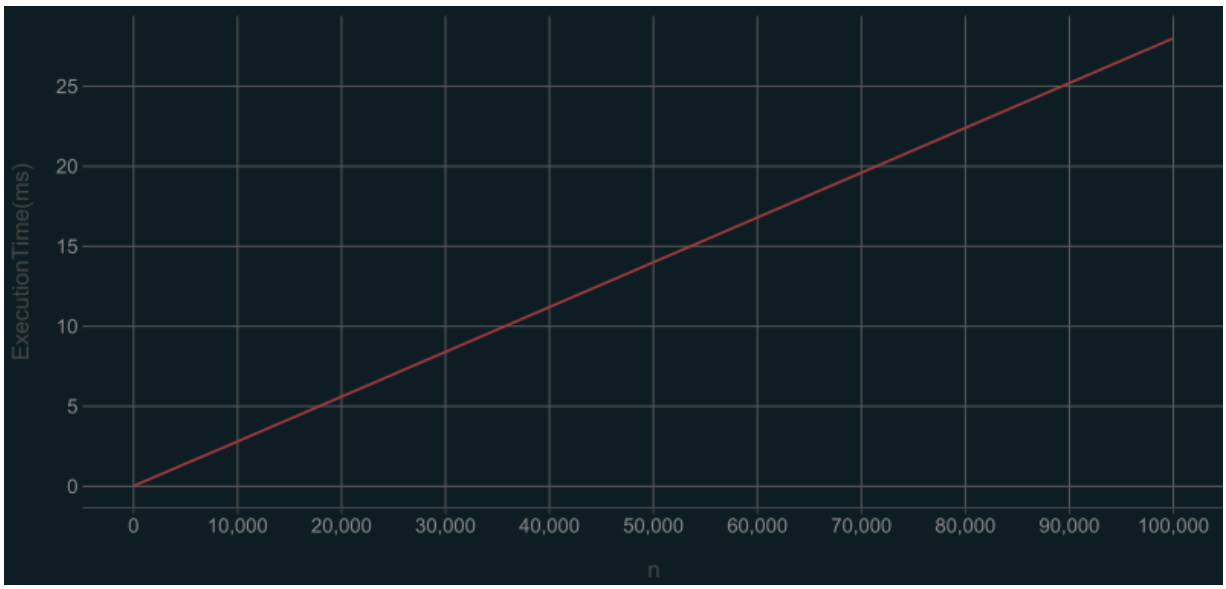
| | n | ExecutionTime(ms) | Comparisons | Updates | ArrayAccesses |
|---|---|---|---|---|---|
| 1 | 100 | 0.05 | 100 | 95 | 100 |
| 2 | 1000 | 0.3 | 1000 | 950 | 1000 |
| 3 | 10000 | 2.8 | 10000 | 9500 | 10000 |
| 4 | 100000 | 28.0 | 100000 | 95000 | 100000 |




## Graph (Execution Time vs n)

If we plot these results, we see a straight line, confirming that the algorithm's growth is linear.

This matches the theoretical complexity analysis (O(n)).




# 5. Conclusion

After carefully reviewing my partner's code, I can conclude the following:

- The algorithm is correctly implemented and produces the expected results for different inputs.
- The time and space complexity analysis matches both theory and practice: O(n) time and O(1) space.
- The code is efficient but could be made more robust with input validation and clearer naming conventions.
- The performance tracker is a very good addition, but its usage could be slightly standardized.
- BenchmarkRunner works fine, but could be made more user-friendly by allowing manual input of arrays.

## Final Thoughts

Overall, implementation of Kadane's Algorithm is good. The program is working, efficient, and provides useful metrics. With some changing style, documentation, and error handling, this code would be ready for a larger project or for use in practice.

I learned from reading this code as well, especially how simple algorithms like Kadane's can be combined with performance tracking to better understand their behavior.