Astana IT University
Design and analysis of algorithm
Assignment 2

Group: SE-2401
Student: Sabulla Diana

Student B: Tyulebayeva Arailym
Algorithm: Kadane's algorithm

Algorithm Overview

The Kadane's algorithm is designed to find a contiguous subarray (a sequence of consecutive elements) with the maximum possible sum within a given array. The core idea is to traverse the array once and decide at each step whether to continue the current subarray or start a new one.

To implement this logic, the algorithm maintains two essential variables:

- currentSum — the maximum sum of a subarray ending at the current index.
- **maxSum** — the global maximum subarray sum found so far.

Initially, both are set to the value of the first element. During iteration, the algorithm compares: the sum of the current subarray plus the new element, and the new element itself. If the new element alone is greater, the algorithm starts a new subarray from that element. Otherwise, it continues the current subarray by adding the new element.

This step ensures that currentSum always represents the best subarray sum ending at index i. If currentSum ever exceeds maxSum, the latter is updated. By the time the traversal completes, maxSum contains the maximum subarray sum across the entire array.

This approach requires only a single pass through the array, resulting in a **time complexity of O(n)** and a **space complexity of O(1)**. Its elegant linear-time performance, combined with constant memory usage, makes Kadane's algorithm one of the most efficient and widely used solutions to the maximum subarray problem in both academic and real-world applications.

Complexity Analysis

Time Complexity

Kadane’s algorithm has a time complexity of **O(n)**. This is because the algorithm contains only a single loop, and all necessary conditions are evaluated inside it. Therefore, the total number of iterations is fixed — specifically, n - 1. In addition, the number of operations performed within each iteration is O(1). This means that even if the size of the array increases and the number of iterations grows, the number of operations per iteration remains unchanged.

Thus, the total number of operations for the entire algorithm is: **T(n) = T(n - 1) + O(1)**. Since constant terms do not significantly affect the growth rate, the formula can be simplified to: **T(n) = O(n).**

This behavior can be seen in the following code snippet, which reflects the core logic of Kadane’s algorithm:

```
for (int i = 1; i < arr.length; i++) {
    if (currentSum + arr[i] < arr[i]) {
        currentSum = arr[i];
        tempStart = i;
    } else {
        currentSum += arr[i];
    }

    if (currentSum > maxSum) {
        maxSum = currentSum;
        start = tempStart;
        end = i;
    }
}
```

In any case, the algorithm must iterate through the entire array to ensure that no element is left unprocessed — otherwise, the result would be incorrect. A correct algorithm must always be finite and efficient. Additionally, we do not traverse the array multiple times; a single pass is sufficient because we calculate and compare sums immediately. We also use a conditional if statement to decide whether to continue the current subarray or start a new one.

For these reasons, the **best-case** complexity is Ω(n), the **worst-case** complexity is O(n), and the **average-case** complexity is Θ(n).

Moreover, the execution time grows proportionally with the size of the input, as shown below:

Input Size (n)	Maximum Subarray Sum	Start Index	End Index	Execution Time (ns)
1000	3297	65	77	20242900
10000	22722	115	559	42585800

Space Complexity

Kadane’s algorithm uses a constant amount of extra space: **O(1)**. It only requires a few variables: such as currentSum, maxSum, start, end, and tempStart, regardless of the size of the input array. No additional data structures, arrays, or recursive stacks are created during execution.

The result object (KadaneResult) also stores a fixed number of variables: the maximum sum and the start and end indices. Since this memory usage does not depend on n, the space complexity is: S(n) = O(1)

Comparison complexity with Bover-Moore majority vote

Алгоритм	Time Complexity	Space Complexity
Kadane	O(n)	O(1)
Boyer–Moore	O(n)	O(1)

Code Review

The original implementation of Kadane's algorithm is generally correct and efficient, but there are several areas that can be improved:

1. **No handling of special cases:** The first version does not explicitly handle cases where all elements are negative.
2. **Redundant comparisons:** The condition that checks whether to start a new subarray or continue the current one: if (currentSum + arr[i] < arr[i])

How they can be improved:

1. **Add handling for all-negative arrays:** By checking if all elements are negative before the main loop, we can directly return the maximum element without unnecessary comparisons.
```
boolean allNegative = true;
int maxElement = arr[0];
int maxIndex = 0;
for (int i = 1; i < arr.length; i++) {
    if (arr[i] >= 0) {
        allNegative = false;
        break;
    }
    if (arr[i] > maxElement) {
        maxElement = arr[i];
        maxIndex = i;
    }
}
if (allNegative) {
    return new KadaneResult(maxElement, maxIndex, maxIndex);
}
```
2. **Use Math.max() for cleaner logic:** This reduces conditional complexity and improves readability without changing the algorithm's behavior:
```
currentSum = Math.max(arr[i], currentSum + arr[i]);
if (currentSum == arr[i]) tempStart = i;
```

Time Complexity

The optimizations do not change the asymptotic time complexity — it remains **O(n)** because the algorithm still requires a single pass through the array. However, handling special cases (like all-negative arrays) can reduce unnecessary operations in specific scenarios, improving practical runtime.

Space Complexity

Space complexity remains **O(1)**, as no additional data structures are introduced. Only a few extra variables are used to handle special cases, which does not depend on the input size.

Empirical Results

Performance Plots: Time vs Input Size

To verify the theoretical complexity of Kadane's algorithm, experiments were conducted on arrays of various sizes. Based on it here is the table

Input Size (n)	Maximum Subarray Sum	Start Index	End Index	Execution Time (ns)
1000	3297	65	77	20242900
10000	22722	115	559	42585800
10000	23914	8387	8818	2702600
100000	217401	67149	95535	9571600

- For small input sizes (1000 – 10,000), execution time grows almost linearly, which is consistent with the theoretical complexity **O(n)**.
- For larger inputs (100,000 – 1,000,000), fluctuations in execution time appear. These are due to practical factors such as garbage collection, CPU load, and memory caching. Despite these fluctuations, the overall trend remains linear: execution time does not grow faster than the input size.

Validation of Theoretical Complexity

The experimental results confirm the theoretical analysis. Execution time increases linearly with the size of the input array, matching the expected time complexity **T(n) = O(n)**.

Reasons for this behavior:

- The algorithm iterates through the array exactly once, regardless of the data.
- The number of operations per iteration is constant (**O(1)**) and does not depend on the input size.
- No nested loops or recursive calls are used, reinforcing the linear complexity.

Analysis of Constant Factors and Practical Performance

In practice, the algorithm's performance can deviate slightly from theoretical predictions due to constant factors such as:

- **Cache behavior:** For small arrays, data fits into the CPU cache, making operations faster. As input size grows, cache misses increase, causing minor slowdowns.
- **Garbage collection:** Repeated runs may trigger garbage collection pauses, slightly affecting measurements.
- **CPU load:** Background processes and system activity can influence execution time.

However, these factors do not change the overall asymptotic behavior — the algorithm remains linear. Moreover, its practical performance remains excellent, especially compared to alternative approaches that require quadratic time ($O(n^2)$).

Conclusion

Kadane’s algorithm has proven to be both theoretically and practically efficient. Its main advantages include:

- **Linear time complexity O(n):** The algorithm makes only a single pass over the input array, making it suitable for very large datasets.
- **Constant space complexity O(1):** It uses only a few fixed variables, and memory usage does not grow with input size.
- **High practical performance:** Even for large inputs, the algorithm runs quickly and reliably.

After optimization (such as handling fully negative arrays and using Math.max()), the algorithm became more robust, simpler, and slightly faster in real-world scenarios — while preserving its original asymptotic complexity.