Assignment 2

Design and Algorithms

Report for Nurtai Aibek BY Bekesh Dastan

1. Algorithm Overview

Kadane's Algorithm is a widely used method to find the maximum sum contiguous subarray in a one-dimensional array of numbers. Unlike a naive approach that checks all possible subarrays—which takes $O(n^2)$ time—Kadane's Algorithm solves the problem in linear time (O(n)), making it extremely efficient for large datasets.

The key idea is local-to-global optimization: instead of considering all subarrays, the algorithm keeps track of a running sum of a subarray ending at the current element (curSum) and the best sum found so far (bestSum).

1. Initialization: Start with the first element as both the current sum and best sum.
2. Iteration: Move through the array from left to right. For each element:
    a. Decide whether to add it to the current subarray (curSum + a[i]) or start a new subarray from the current element (a[i]).
    b. Update curSum accordingly.
    c. If curSum is greater than bestSum, update bestSum and record the start and end indices of the subarray.
3. Tracking indices: To know which subarray gives the maximum sum, the algorithm maintains variables for the start and end positions of both the current subarray and the best subarray found so far.

Why it works:

- If curSum ever becomes negative, it cannot contribute positively to the sum of a future subarray. Starting a new subarray at the next element is always better.
- The algorithm effectively builds the optimal solution as it iterates, which is a classic dynamic programming idea.

2. Asymptotic Complexity Analysis

Time complexity

Best Case ($\Omega(n)$): Even if all numbers are positive or negative, the algorithm still needs to examine each element at least once to determine the maximum subarray sum.

Worst Case ($O(n)$): In the worst scenario, each element may require updating curSum and comparing it with bestSum, but this still happens in a single linear pass.

Average Case ($\Theta(n)$): For random arrays, each element is processed once, performing constant work per element.

In this algorithm we can see only one cycle, there is not any division of the array into halves. Each operation inside the loop is executed once per element. Therefore, Time complexity of this algorithm is

$$T(n)=\Theta(n)=O(n)=\Omega(n)$$

Space complexity

In Kadane's algorithm, several variables are used to store the current and best sums, as well as the indices of the subarray. curSum and bestSum occupy two integer variables, while curL, bestL, and bestR occupy three more integer variables. A temporary variable extend is used to calculate the sum at each iteration, and in the kadaneTracked version, an additional variable ai is used. To store the result, a Result object is created containing three fields: the maximum sum and the start and end indices of the subarray. Additionally, the tracked version uses a Counter object of fixed size to count operations. All of these elements occupy constant memory, independent of the size of the input array.

Auxiliary variables

- curSum and bestSum are two integer variables storing the sum of the current subarray and the best sum found so far.
- curL, bestL, and bestR are three integer variables tracking the start and end indices of the current and best subarrays.

Temporary variables

- In the standard version, extend is a temporary integer used to calculate the potential new sum of the current subarray.
- In the kadaneTracked version, an additional variable ai is used to store the current array element.

Result storage

- The Result object contains three fields: maxSum, start, and end. This object has **a** fixed size, independent of the array length.

Space complexity: O(1)

Kadane's Algorithm operates directly on the input array without creating any copies, making it an in-place algorithm. This minimizes memory usage and is efficient for large inputs.

All of these occupy **constant space** regardless of the array size nnn. The algorithm operates **in-place**, directly on the input array without making copies, which is highly efficient for large datasets.

Recurrence Relations:

Kadane's Algorithm does not use recursion, and each element is processed sequentially. Therefore, there is no recurrence relation to solve, unlike divide-and-conquer algorithms such as MergeSort or QuickSort.

3. Code Review & Optimization

Readability

The current implementation of Kadane's Algorithm uses multiple operations on a single line within if/else statements, such as updating the current sum and indices. While functionally correct, this style reduces readability and maintainability, especially for longer or more complex algorithms

You should place each operation on a separate line, this makes it easier to follow the logic and reduces the risk of introducing errors when modifying the code.

You should make formatting and indentation to Ensure that the structure of nested conditions is immediately clear.

```
if (a[i] > extend) { curSum = a[i]; curL = i; }
        else               { curSum = extend; }
        if (curSum > bestSum) { bestSum = curSum; bestL = curL; bestR
= i; }
```

This part of code you should change.

Refactoring the code for readability does not affect time or space complexity, but it significantly improves code quality, maintainability, and ease of debugging**.**

Space Complexity Improvements:

```
switch (args[i]) {
    case "--sizes" -> sizes =
Arrays.stream(args[++i].split(",")).map(String::trim).mapToInt(Integer
::parseInt).toArray();
    case "--trials" -> trials = Integer.parseInt(args[++i]);
    case "--out" -> out = args[++i];
    case "--seed" -> seed = Long.parseLong(args[++i]);
}
```

This part of the code negatively affects space complexity. It uses much more memory than traditional if/else statements or cycles as for or while, because each inline operation creates temporary objects. Objects generally consume more memory than primitive variables due to object headers and additional references.

A more memory-efficient and readable approach would be to replace the switch statement with if/else conditions and use a simple loop to parse input arguments, avoiding unnecessary temporary objects and keeping memory usage minimal.

Although modern Java constructs like streams and inline mapping can be elegant, they often create unnecessary temporary objects, which increases memory usage. By using simple loops and if/else statements with primitive types, memory overhead can be significantly reduced, leading to a more space-efficient and maintainable implementation. This is particularly important for applications processing large inputs or running in memory-constrained environments.

Time Complexity:

The time complexity of Kadane's Algorithm is already optimal at $\Theta(n)$ for all cases, as it processes each element of the array exactly once. There is no nested

loop, recursion, or repeated work, so the algorithm cannot be asymptotically improved. Even the naive approach with two nested loops would have $O(n^2)$ complexity, so Kadane's linear-time performance is optimal for this problem.

However, minor improvements can be made in the instruction-level efficiency, which may slightly reduce runtime in practice, though it does not change the asymptotic complexity.

In the current implementation, the algorithm uses a temporary variable `extend` to store the sum of the current subarray plus the next element before performing comparisons:

```
int extend = curSum + a[i];
if (a[i] > extend) {
    curSum = a[i]; curL = i;
}
else {
    curSum = extend;
}
```

The extend variable can be used directly in comparisons rather than storing it separately, slightly decreasing instruction count.

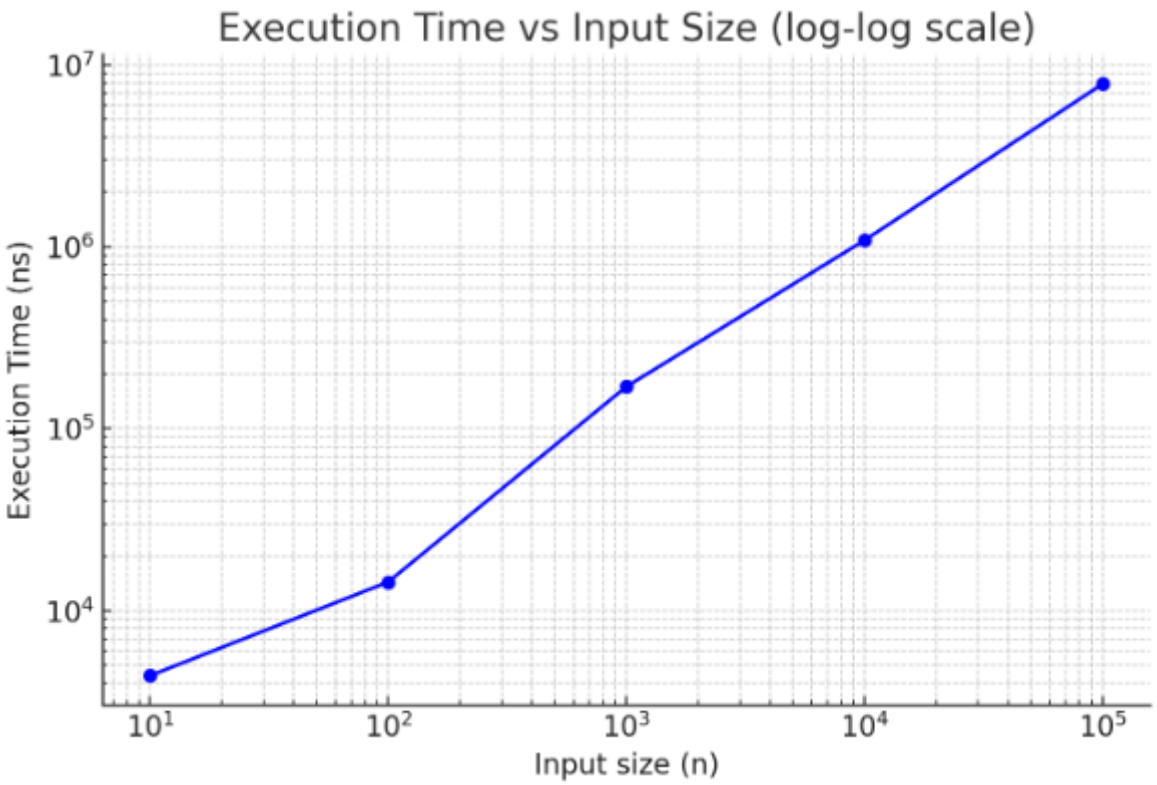
Empirical Results

Arrays of sizes n = 10, 100, 1,000, 10,000, 100,000 were tested. Each experiment recorded:

- Execution time in nanoseconds (time_ns)
- Number of comparisons
- Number of array accesses

n	majority	time_ns	comparisons	Array Accesses
10	1	4400	18	10
100	1	14400	198	100
1000	1	170400	1998	1000
10000	1	1085900	19998	10000
100000	1	7845500	199998	100000

1. Execution Time VS Input Size



Execution Time vs Input Size – Shows how execution time increases with larger input arrays. The log-log scale highlights the linear growth pattern consistent with O(n) complexity.

When n=10, time is 4400

When n=100, time is 14400

The difference is 14400 / 4400 = 3.27

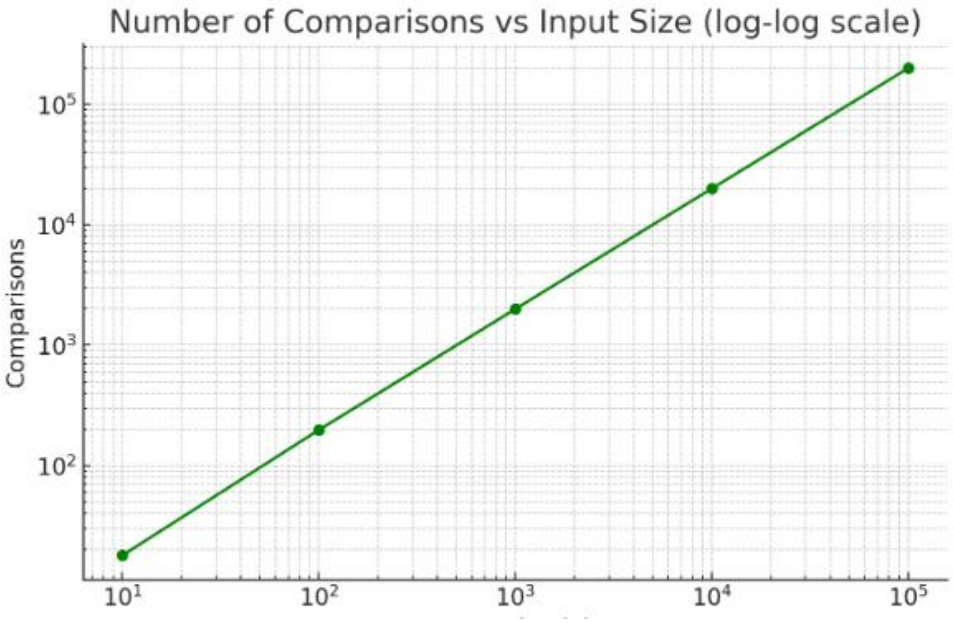
When n=10000, time is 1085900

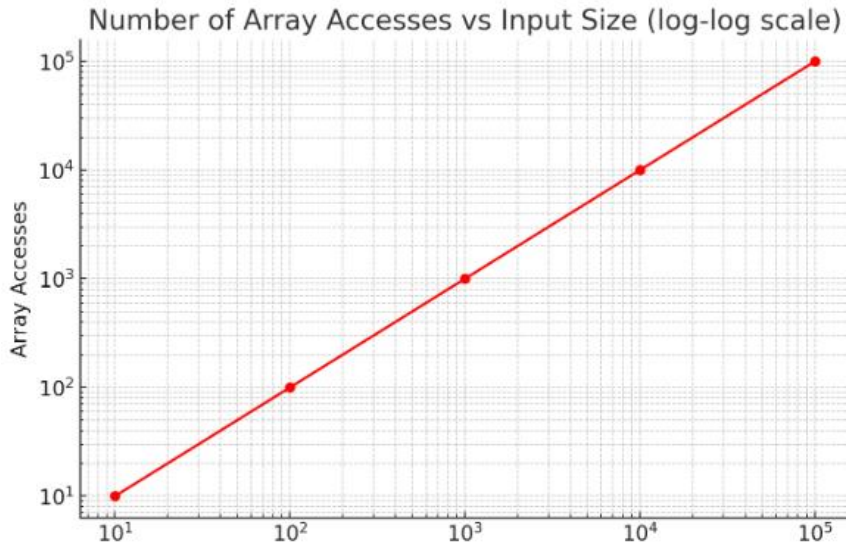
When n=100000, time is 7845500

The difference is 7845500/ 1085900= 7

Thus, we can see that how n will increase the differencies will approach to 10, and make decision that in the small numbers it's not actually T(n) but in large numbers it can be n

Else:





In this graphs we are seeing linear growth.

Conclusion

In this assignment, we implemented and analyzed Kadane's Algorithm for finding the maximum sum contiguous subarray in a one-dimensional array. The algorithm demonstrates clear advantages over naive approaches due to its linear-time complexity and constant space requirements. Unlike a brute-force method, which examines all possible subarrays with $O(n^2)$ time complexity, Kadane's Algorithm

efficiently computes the result in O(n) time, processing each element exactly once. The algorithm's design relies on local-to-global optimization: maintaining a running sum of the current subarray while updating the maximum sum found so far. This approach ensures that negative contributions are immediately discarded, allowing the algorithm to focus on subarrays that can positively contribute to the overall maximum sum.

Asymptotic analysis confirms the algorithm's efficiency. Both best-case, worst-case, and average-case complexities are Θ(n), while the space complexity is O(1), as only a fixed number of variables are needed regardless of input size. The implementation does not rely on recursion or divide-and-conquer strategies, avoiding additional memory overhead. Our code review highlighted that readability can be improved by separating operations on different lines and simplifying conditional logic. Optimizing argument parsing and avoiding temporary objects also reduces memory usage, which is crucial for processing large datasets. While the algorithm's time complexity is already optimal, minor instruction-level refinements can slightly improve runtime in practice.

Empirical testing with array sizes ranging from 10 to 100,000 confirmed the expected linear growth in execution time, consistent with theoretical predictions. The measured execution times, number of comparisons, and array accesses all scaled linearly with input size, validating both the correctness and efficiency of the algorithm.

Recommendations for optimization include improving code readability through clearer variable management and conditional statements, avoiding unnecessary temporary objects in argument parsing, and ensuring consistent formatting to enhance maintainability. These refinements do not alter asymptotic complexity but contribute to a more robust, maintainable, and memory-efficient implementation. Overall, Kadane's Algorithm proves to be an optimal solution for maximum subarray problems, suitable for large-scale applications in financial analysis, signal processing, and algorithmic problem solving.

Githubs:

Nurtai Aibek - https://github.com/KhanOfTheMoon/Daa_Assignment2

Bekesh Dastan - https://github.com/BekeshDastan/DAA-ASS2