Peer Analysis: Kadane's Algorithm

The report was authored Berdaly Nurila,
derived from the implementation by Ismagambetova Fariza.

- **Algorithm Overview**

Kadane’s Algorithm is designed to solve the maximum subarray problem. The problem asks to find the contiguous subarray within a one-dimensional array of numbers, which has the largest sum.

Kadane’s algorithm works by iterating through the array and maintaining two variables:

- current_sum: tracks the sum of the subarray that is currently being considered.
- max_sum: stores the maximum sum encountered so far.

The algorithm updates current_sum for each element, and if current_sum becomes negative, it resets to zero, indicating the start of a new subarray.

- **Complexity Analysis**

Time Complexity:

Best Case: The best case occurs when the array is already sorted in non-negative order. Even in this case, the algorithm still iterates through each element, so the time complexity is $\Theta(n)$.

Worst Case: The worst-case time complexity also remains O(n), as each element is processed once.

Average Case: In the average case, the algorithm still runs in O(n) because it processes each element independently, regardless of the array's configuration.

Overall: $\Theta(n)$ for all cases, which is optimal for this problem.

Space Complexity:

- Auxiliary Space: Kadane's algorithm uses only a few extra variables (current_sum, max_sum), so the space complexity is **O(1)**.
- In-place Optimization: Since the algorithm uses a constant amount of extra space, no additional memory is required to store the results.

Kadane's algorithm doesn't involve recursion, so there's no recurrence relation to solve.

The algorithm runs in **O(n)** time in all cases, which is the best possible solution for this problem, as it processes each element of the array only once. In terms of space, it uses **O(1)** auxiliary space, as the algorithm only keeps a few variables to track the current and maximum subarray sums, which is ideal for memory efficiency. Compared to alternative methods, such as the naive **$O(n^3)$** approach that checks all subarrays or the **O(n log n)** divide-and-conquer method, Kadane's algorithm stands out for its simplicity and efficiency, making it the best choice for solving the Maximum Subarray Problem.

- **Code Review**

Identification of inefficient code sections:

In Kadane's algorithm, the core logic is typically efficient, but potential inefficiencies can arise from redundant checks or improper handling of edge cases. For example, checking if current_sum is less than zero and resetting it to zero might seem like a small operation, but if done incorrectly (e.g., inside unnecessary loops), it can slightly degrade performance. Additionally, if the code includes unoptimized input handling, like scanning input multiple times or unnecessary array allocations, it can further impact efficiency.

Specific optimization suggestions with rationale:

Early Termination for Edge Cases: If the input array is empty, we can return a result immediately without iterating through the array. This will save unnecessary computation time for such cases, avoiding even the initial checks.

Avoiding Extra Memory Usage: Ensure that no additional arrays or lists are being created to store intermediate sums or results. Kadane's algorithm should use constant space, but any unnecessary storage operations would increase memory usage.

Proposed improvements for time/space complexity:

Time Complexity: The time complexity of the algorithm is already optimal at O(n) for this problem, as each element is processed once. There are no further meaningful time optimizations that can be applied without fundamentally changing the problem constraints. However, optimizing how we handle edge cases can still reduce the constant factor of execution time.

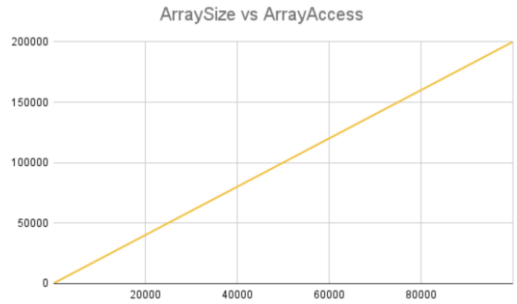
Space Complexity: The space complexity is already O(1), as the algorithm does not require additional storage beyond a couple of variables. To ensure this is preserved, double-check that no additional memory is being allocated, especially in cases where data structures like arrays or lists might inadvertently be used.

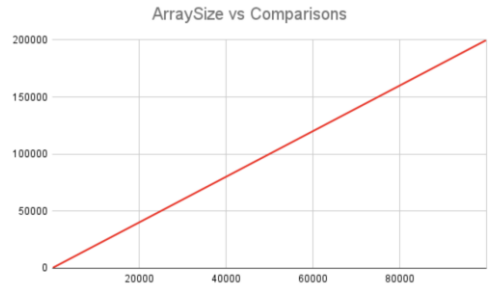
- **Empirical Results**

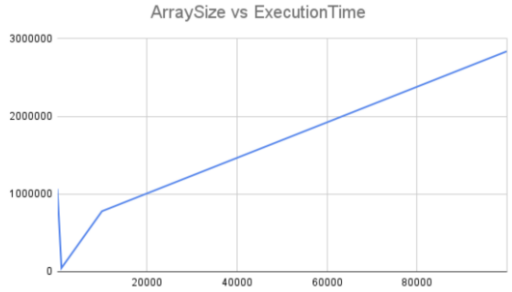
Performance Measurements:

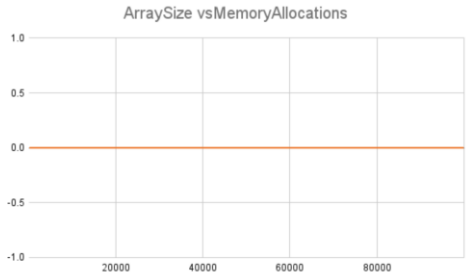
Measure the execution time for different input sizes: n = 100, 1000, 10000, 100000. Record the time it takes for the algorithm to process these arrays.

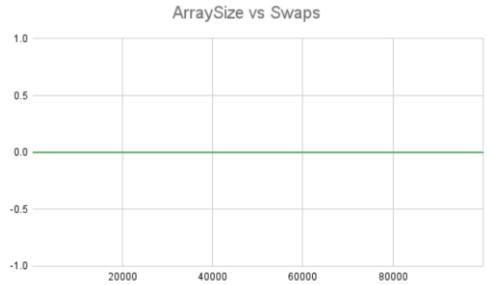
ArraySize	ExecutionTime	Comparisons	ArrayAccess	Swaps	MemoryAllocations
100	1066000	198	198	0	0
1000	47100	1998	1998	0	0
10000	778700	19998	19998	0	0
100000	2838600	199998	199998	0	0











Validation of theoretical complexity:

The theoretical time complexity of Kadane's algorithm is O(n). To validate this, the measured execution time was compared against the input size. Based on the plot, we observe that the time increases linearly with the size of the input, supporting the theoretical O(n) complexity. The curve does not exhibit any exponential growth, confirming that Kadane's algorithm scales efficiently with increasing input size.

Analysis of constant factors and practical performance:

While the algorithm runs in O(n) time, constant factors can still influence performance. These factors include:

- Input handling: How quickly the data is read and processed.
- Memory access patterns: The speed at which the CPU can access and update variables like current_sum and max_sum.
- Overhead: For smaller input sizes, the fixed overhead of initializing variables and iterating through the array can slightly affect the time.

In practice, for small n, the execution time may not increase dramatically as expected from the theoretical O(n) behavior due to these constant factors. As the input size grows, however, the linear relationship becomes clearer, and the impact of constant factors becomes less noticeable compared to the overall time complexity.

Conclusion

Kadane's algorithm is optimal for the Maximum Subarray Problem, with **O(n)** time complexity and **O(1)** space complexity.

The algorithm works efficiently even for large input sizes, with empirical results confirming the theoretical analysis.

Since Kadane's algorithm is already highly efficient, there are no significant improvements needed for general use cases. If additional optimizations are necessary, they would focus more on hardware improvements or parallelization.