Theoretical Analysis

Maximum Subarray Sum Problem

1. Problem Definition

The **Maximum Subarray Sum** problem consists of determining the largest possible sum of a contiguous subarray within a given array of integers, which may include both positive and negative values.

Input:
An array $A = [a_0, a_1, \dots, a_{n-1}]$of size n.

Output:
The maximum possible sum of any contiguous subarray of (A).

Formally, the problem can be defined as finding indices (i) and (j) such that:

Maximize S(i,j)=k=i∑jakwhere 0≤i≤j<n

2. Naive Approach (Brute Force)

Idea

The naive approach evaluates all possible contiguous subarrays, computes the sum of each subarray, and selects the maximum sum encountered.

Algorithm Description

1. Iterate over all possible starting indices (i).
2. For each (i), iterate over all possible ending indices (j).
3. Compute the sum of elements from index (i) to (j).
4. Maintain and update the maximum sum.

Time Complexity Analysis

The number of possible subarrays in an array of size (n) is:

n(n+1)/2

If the sum of each subarray is recomputed from scratch, the total number of operations is:

T(n)=i=0∑n−1j=i∑n−1k=i∑j1

This results in a **cubic order of growth**:

T(n)=O(n3)

If prefix sums are used to compute each subarray sum in constant time, the complexity improves to:

$T(n)=O(n2)$

Space Complexity

- Without prefix sums: (O(1))
- With prefix sums: (O(n))

Limitations

Despite its conceptual simplicity, the naive approach is computationally expensive and does not scale well for large input sizes due to its quadratic or cubic time complexity.

3. Optimized Approach — Kadane's Algorithm

Idea

Kadane's Algorithm is a dynamic programming technique that processes the array in a single pass.
At each position, the algorithm decides whether to:

- Extend the previously computed subarray, or
- Start a new subarray at the current element.

This decision is based on maximizing the sum of a subarray ending at the current index.

4. Algorithm Explanation

Two variables are maintained:

- currentSum: the maximum sum of a subarray ending at the current index.
- bestSum: the maximum subarray sum encountered so far.

At each index (i), the following recurrence is applied:

$currentSum=max(ai,currentSum+ai)$

$bestSum=max(bestSum,currentSum)$

5. Correctness Argument

Kadane's Algorithm is correct based on the following observations:

1. Any subarray with a negative sum will decrease the total sum of any subarray that extends it.

2. Therefore, when the accumulated sum becomes negative, it is optimal to discard the current subarray and start a new one.
3. At each step, currentSum represents the maximum possible sum of a subarray ending at the current index.

By maintaining this invariant and scanning the array once, the algorithm guarantees that the maximum subarray sum is correctly identified.

6. Time and Space Complexity Analysis

Each element of the array is processed exactly once, and all operations inside the loop are constant time.

$T(n)=c \cdot n \Rightarrow O(n)$

The algorithm uses a constant number of variables, resulting in:

- **Time Complexity:** $O(n)$
- **Space Complexity:** $O(1)$

Kadane's Algorithm is asymptotically optimal, as every element must be examined at least once.

7. Comparison of Approaches

APPROACH	TIME COMPLEXITY (ORDER)	SPACE COMPLEXITY
NAIVE (BRUTE FORCE)	$O(n^3)$	$O(1)$
NAIVE + PREFIX SUMS	$O(n^2)$	$O(n)$
KADANE'S ALGORITHM	$O(n)$	$O(1)$

8. Conclusion

The Maximum Subarray Sum problem highlights the significance of algorithmic optimization.
While the naive approach is straightforward, it suffers from poor scalability.
Kadane's Algorithm provides an efficient and elegant solution by applying dynamic programming principles, reducing the order of growth to linear time and constant space, which makes it suitable for large-scale applications.