Technical Report: Maximum Subarray Problem using Kadane's Algorithm

Chetan Prakash
UID: 23bcs13776
Section: krg 1 b

November 7, 2025

Abstract

This report explores the Maximum Subarray Problem, a classic challenge in computer science and data analysis. The problem involves finding a contiguous subarray within a one-dimensional numeric array that has the largest possible sum. While brute-force methods can solve this in $O(n^2)$ or $O(n^3)$ time, this report focuses on **Kadane's Algorithm**, a dynamic programming approach that solves the problem in optimal $O(n)$ linear time. We provide a detailed C++ implementation that not only finds the maximum sum but also tracks the starting and ending indices of the optimal subarray. A step-by-step dry run and complexity analysis are included to validate the algorithm's efficiency.

1 Introduction

The Maximum Subarray Problem is a fundamental task in algorithm design. It was first proposed by Ulf Grenander in 1977 while analyzing 2D digital images. The problem asks: given an array of integers (containing both positive and negative numbers), which contiguous sub-segment of this array has the maximum summation?

Applications of this problem include:

- **Financial Analysis:** Finding the best time to buy and sell a stock to maximize profit (converted to a daily change array).
- **Data Mining:** Identifying genomic sequence segments with specific characteristics.
- **Computer Vision:** Finding the brightest area in a strictly rectangular region of an image.

While simple in premise, the presence of negative numbers makes the problem non-trivial. If all numbers were positive, the maximum subarray would simply be the entire array.

2 Problem Statement

Given an integer array $A[0 \dots n-1]$, we want to find indices L and R ($0 \le L \le R < n$) such that the sum S is maximized:

$$S = \sum_{i=L}^{R} A[i]$$

We require an output of:

1. The maximum sum S.
2. The starting index L.
3. The ending index R.

3 Approaches

3.1 Brute Force Approach

The naive approach is to check every possible subarray.

- We can iterate through all possible starting points i.
- For each i, iterate through all possible ending points j.
- Calculate the sum between i and j.

This results in a time complexity of $O(n^3)$ if the sum is recalculated from scratch, or $O(n^2)$ if we use prefix sums or optimized inner loops. This is inefficient for large datasets.

3.2 Kadane's Algorithm (Optimal)

Kadane's algorithm improves this to $O(n)$ by using dynamic programming. It maintains a running maximum sum ending at the current position.

The core insight is: If the maximum sum ending at index $i-1$ is negative, then including it in a subarray ending at i will only decrease the total. Therefore, if the previous running sum is negative, we should discard it and start a new subarray at index i.

We maintain two variables:

- `max_ending_here`: The maximum sum of a subarray that *must* end at the current position.
- `max_so_far`: The maximum sum found anywhere in the array so far.

4 Implementation (C++)

The following implementation finds the maximum sum and also tracks the `start` and `end` indices.

```cpp
#include <bits/stdc++.h>
using namespace std;

tuple<long long, int, int> kadane_with_indices(const vector<long long>& a)
    {
    if (a.empty()) return {0, -1, -1};

    long long max_so_far = a[0];
    long long max_ending_here = a[0];

    int start = 0;
    int end = 0;
    int s = 0;

    for (int i = 1; i < a.size(); i++) {
        if (a[i] > max_ending_here + a[i]) {
            max_ending_here = a[i];
            s = i;
        } else {
            max_ending_here += a[i];
        }

        if (max_ending_here > max_so_far) {
            max_so_far = max_ending_here;
            start = s;
            end = i;
        }
    }

    return {max_so_far, start, end};
}

int main() {
    vector<long long> arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};

    cout << "Input Array: { ";
    for(long long x : arr) cout << x << " ";
    cout << "}\n\n";

    cout << "--- Running Kadane's Algorithm ---\n";
    auto result = kadane_with_indices(arr);

    long long maxSum = get<0>(result);
    int L = get<1>(result);
    int R = get<2>(result);

    cout << "Maximum Contiguous Sum: " << maxSum << "\n";
    if (L != -1) {
        cout << "Start Index: " << L << "\n";
        cout << "End Index: " << R << "\n";
        cout << "Subarray Elements: [ ";
        for (int i = L; i <= R; ++i) {
            cout << arr[i] << (i == R ? " " : ", ");
        }
        cout << "]\n";
    }

    return 0;
}
```

4.1 Output

Running the above code produces the following output:

```
Input Array: { -2 1 -3 4 -1 2 1 -5 4 }

--- Running Kadane's Algorithm ---
Maximum Contiguous Sum: 6
Start Index: 3
End Index: 6
Subarray Elements: [ 4, -1, 2, 1 ]
```

5 Algorithm Dry Run

Let us trace the algorithm step-by-step with the example array: $A = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$
Initial State: `max_so_far = -2`, `max_ending_here = -2`, `start = 0`, `end = 0`, `s = 0`

Table 1: Step-by-step trace of Kadane's algorithm.

Iteration (i)	Value A[i]	Decision (Start new vs Extend)	max_ending_here	max_so_far Update
1	1	$1 > (-2 + 1) \rightarrow$ **Start New** ($s = 1$)	1	Yes (-2 $\rightarrow$ 1)
2	-3	$-3 < (1 - 3) \rightarrow$ Extend	-2	No
3	4	$4 > (-2 + 4) \rightarrow$ **Start New** ($s = 3$)	4	Yes (1 $\rightarrow$ 4)
4	-1	$-1 < (4 - 1) \rightarrow$ Extend	3	No
5	2	$2 < (3 + 2) \rightarrow$ Extend	5	Yes (4 $\rightarrow$ 5)
6	1	$1 < (5 + 1) \rightarrow$ Extend	6	Yes (5 $\rightarrow$ 6)
7	-5	$-5 < (6 - 5) \rightarrow$ Extend	1	No
8	4	$4 < (1 + 4) \rightarrow$ Extend	5	No

Final Result: The maximum sum is **6**, found in the subarray from index 3 to 6: $[4, -1, 2, 1]$.

6 Complexity Analysis

6.1 Time Complexity: $O(n)$

The algorithm uses a single loop that iterates through the array exactly once. Inside the loop, we only perform constant-time arithmetic operations and comparisons. Thus, the time complexity is linear with respect to the input size n.

6.2 Space Complexity: $O(1)$

We only use a fixed number of integer variables (`max_so_far`, `max_ending_here`, `start`, `end`, `s`, `i`) regardless of the input array size. We do not use any auxiliary data structures like extra arrays or hash maps.

7 Conclusion

Kadane's algorithm is the most efficient standard approach for solving the Maximum Subarray Problem. Its capability to handle negative numbers and its linear time complexity make it superior to naive brute-force approaches. The implementation provided successfully identifies not just the sum, but the exact location of the optimal subarray, making it suitable for practical applications in financial and data analysis.