Maximum Subarray Sum (Kadane's Algorithm)

Problem Definition:

Given an array of integers (which may include both positive and negative numbers), the goal is to identify the contiguous subarray that has the maximum sum and return that sum.

Example:

Consider the array: arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4].

- The subarray [4, -1, 2, 1] has the maximum sum of 6.

Explanation:

- The **contiguous subarray** means the elements in the subarray are consecutive elements in the array.
- **Sum** is the sum of all elements in the subarray.
- The task is to find such a subarray with the largest sum among all possible subarrays.

Brute Force Approach:

One way to solve this is to consider every possible subarray, calculate the sum, and track the maximum sum found. This approach, though correct, is inefficient, with a time complexity of O(n^3).

Optimized Approach - Kadane's Algorithm:

A more efficient approach is to use Kadane's Algorithm, which solves the problem with a time complexity of O(n). The algorithm works by iterating through the array, maintaining the current maximum subarray sum ending at each position, and updating the global maximum sum encountered.

Kadane's Algorithm Recap:

1. **Initialize** max_ending_here and max_so_far with the first element of the array.
2. **Iterate** through the array from the second element:
   - Update max_ending_here to be the maximum of the current element or the sum of max_ending_here and the current element.
   - Update max_so_far to be the maximum of max_so_far and max_ending_here.
3. **Return** max_so_far as the maximum sum of the contiguous subarray.

Practical Uses:

- The Maximum Subarray Sum problem is a classical problem in computer science, often used in algorithm and data structure courses to teach dynamic programming.
- It's also useful in various real-world applications, such as financial modeling (e.g., finding the period of maximum profit/loss) or in any situation where the optimization of sequential data is needed.

Example Walkthrough:

Let's walk through an example array: arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

1. **Initial Values**:
   - max_ending_here = -2
   - max_so_far = -2
2. **Iteration 1 (i = 1, x = 1)**:
   - max_ending_here = max(1, -2 + 1) = 1
   - max_so_far = max(-2, 1) = 1
3. **Iteration 2 (i = 2, x = -3)**:
   - max_ending_here = max(-3, 1 + -3) = -2
   - max_so_far = max(1, -2) = 1
4. **Iteration 3 (i = 3, x = 4)**:
   - max_ending_here = max(4, -2 + 4) = 4
   - max_so_far = max(1, 4) = 4
5. **Iteration 4 (i = 4, x = -1)**:
   - max_ending_here = max(-1, 4 + -1) = 3
   - max_so_far = max(4, 3) = 4
6. **Iteration 5 (i = 5, x = 2)**:
   - max_ending_here = max(2, 3 + 2) = 5
   - max_so_far = max(4, 5) = 5
7. **Iteration 6 (i = 6, x = 1)**:
   - max_ending_here = max(1, 5 + 1) = 6
   - max_so_far = max(5, 6) = 6
8. **Iteration 7 (i = 7, x = -5)**:
   - max_ending_here = max(-5, 6 + -5) = 1
   - max_so_far = max(6, 1) = 6
9. **Iteration 8 (i = 8, x = 4)**:
   - max_ending_here = max(4, 1 + 4) = 5
   - max_so_far = max(6, 5) = 6

Result:

The maximum sum of a contiguous subarray in the array [-2, 1, -3, 4, -1, 2, 1, -5, 4] is 6, which corresponds to the subarray [4, -1, 2, 1].

Brute Force Approach:

The problem is to find the maximum sum of a contiguous subarray in an array of integers.

```python
def max_subarray_sum_brute_force(arr):
    n = len(arr)
    max_sum = float('-inf')

    for i in range(n):
        for j in range(i, n):
            current_sum = 0
            for k in range(i, j+1):
                current_sum += arr[k]
            max_sum = max(max_sum, current_sum)

    return max_sum

# Example usage:
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum_brute_force(arr))  # Output: 6
```

Time Complexity: O(n^3)

Space Complexity: O(1)

Optimized Approach (Kadane's Algorithm):

We can optimize the above brute force approach by using Kadane's Algorithm, which scans the array in a single pass.

```python
def max_subarray_sum_kadane(arr):
    max_ending_here = max_so_far = arr[0]

    for x in arr[1:]:
        max_ending_here = max(x, max_ending_here + x)
        max_so_far = max(max_so_far, max_ending_here)

    return max_so_far

# Example usage:
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print(max_subarray_sum_kadane(arr))  # Output: 6
```

Time Complexity: O(n)

Space Complexity: O(1)