Name : Sneha Singh
Reg. No. 231081067
Branch : SY BTech Information Technology

Objective:

Implement and analyze the Find-Max-Crossing-Subarray algorithm, a key part of the Divide-and-Conquer approach to solving the Maximum Subarray Problem. This assignment will deepen your understanding of algorithmic design and efficiency in solving array-based problems.

The **Maximum Subarray Problem** is about finding the contiguous subarray within a one-dimensional array of numbers that has the largest sum. The **Divide and Conquer** approach is an efficient method that reduces the problem into smaller subproblems and then combines their results. Let's break down the approach step by step:

Steps of the Divide and Conquer Approach:

1. **Divide** the given array into two halves.
   - The midpoint of the array is used to separate it into a left half and a right half.
2. **Conquer** each half:
   - Recursively find the maximum subarray sum in the left half.
   - Recursively find the maximum subarray sum in the right half.
   - This process continues, dividing the array into smaller and smaller subarrays until the base case (a subarray with just one element) is reached.
3. **Combine** the results:
   - The overall maximum subarray may be in one of three places: a. It might lie **entirely within the left half**. b. It might lie **entirely within the right half**. c. It might **cross the midpoint**, meaning it includes elements from both halves of the array.
4. The recursive calls in steps 2.a and 2.b handle the first two possibilities (left and right subarrays). The trickier part is handling the third case: the subarray that **crosses the midpoint**.

Finding the Maximum Crossing Subarray:

To handle the crossing subarray efficiently, we don't need to examine every possible combination. Instead, we can find this subarray in **linear time** by considering two parts:

1. **Maximum sum on the left side of the midpoint:**
   - Start from the midpoint and move leftwards, keeping track of the maximum sum of any subarray that ends at the midpoint.
2. **Maximum sum on the right side of the midpoint:**

    - Start from the element just after the midpoint and move rightwards, keeping track of the maximum sum of any subarray that starts at .

Finally, combine the two sums (left and right) to get the maximum subarray that crosses the midpoint.

Detailed Steps for Finding the Maximum Crossing Subarray:

- **Step 1:** Start from the midpoint and move leftwards, calculating the sum of elements as you go. Track the largest sum encountered, which will give you the maximum subarray sum ending at the midpoint.
- **Step 2:** Similarly, start from the element right after the midpoint and move rightwards. Keep a running sum, and track the maximum sum encountered, which will give you the maximum subarray sum starting from .
- **Step 3:** Combine the two results (left sum + right sum). This sum represents the maximum subarray that crosses the midpoint.
-

TESTING : .

Example:

Consider the array: .

- Divide it into two halves: and .
- Recursively find the maximum subarrays in each half.
- Then, calculate the maximum subarray that crosses the midpoint by finding the maximum sum from the midpoint to the left and from the midpoint to the right.
- Finally, compare the left half's result, the right half's result, and the crossing subarray sum to get the overall maximum.

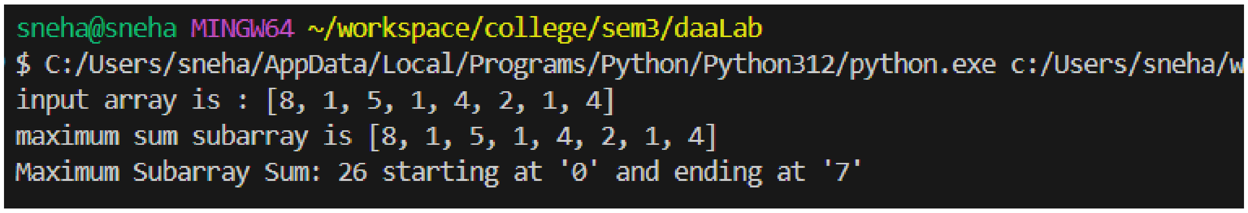
Example TestCases:

Testcase 1 : An array with all positive numbers.

```
arr = [8, 1, 5, 1, 4, 2, 1, 4]
```

Expected Output:
Sum : 26
StartIdx = 0, EndIdx = 7

Actual output :

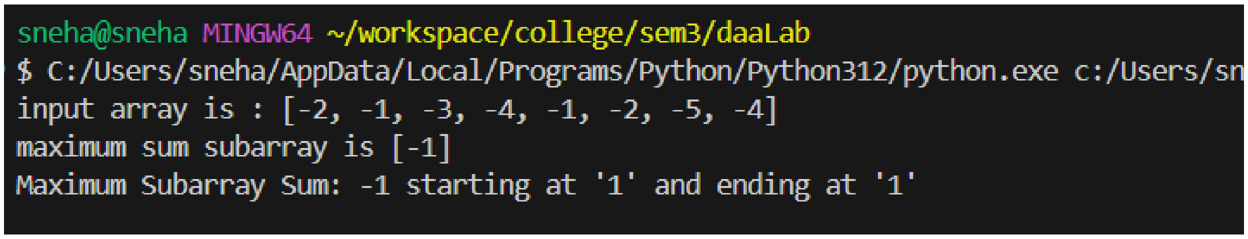

```
sneha@sneha MINGW64 ~/workspace/college/sem3/daaLab
$ C:/Users/sneha/AppData/Local/Programs/Python/Python312/python.exe c:/Users/sneha/w
input array is : [8, 1, 5, 1, 4, 2, 1, 4]
maximum sum subarray is [8, 1, 5, 1, 4, 2, 1, 4]
Maximum Subarray Sum: 26 starting at '0' and ending at '7'
```

Testcase 2 : An array with all negative numbers.

```
arr = [-2, -1, -3, -4, -1, -2,  -5, -4]
```

Expected Output:
max Sum : -1
StartIdx = 1, EndIdx = 1

Actual output :

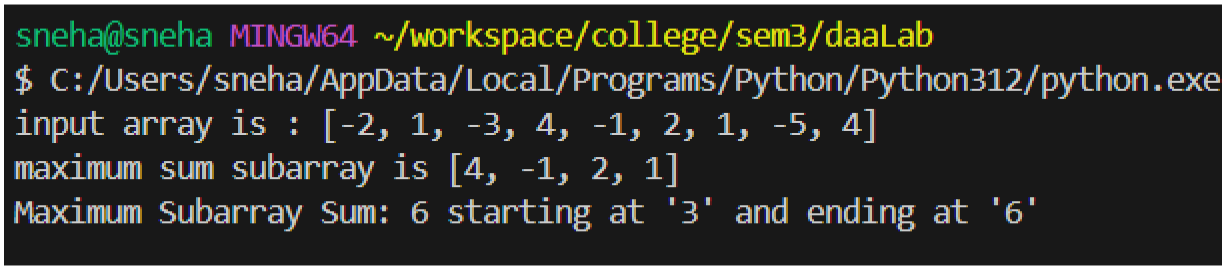

```
sneha@sneha MINGW64 ~/workspace/college/sem3/daaLab
$ C:/Users/sneha/AppData/Local/Programs/Python/Python312/python.exe c:/Users/sn
input array is : [-2, -1, -3, -4, -1, -2, -5, -4]
maximum sum subarray is [-1]
Maximum Subarray Sum: -1 starting at '1' and ending at '1'
```

Testcase 3 : An array with a mix of positive and negative numbers.

```
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
```

Expected Output:
max Sum : 6
StartIdx = 3, EndIdx = 6

Actual output :

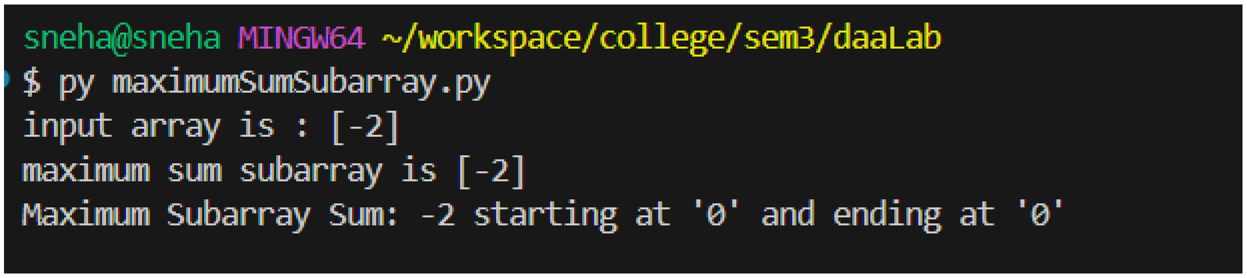

```
sneha@sneha MINGW64 ~/workspace/college/sem3/daaLab
$ C:/Users/sneha/AppData/Local/Programs/Python/Python312/python.exe
input array is : [-2, 1, -3, 4, -1, 2, 1, -5, 4]
maximum sum subarray is [4, -1, 2, 1]
Maximum Subarray Sum: 6 starting at '3' and ending at '6'
```

TestCase 4 : An array with a single element.

```
arr = [-2]
```

Expected Output:
max Sum : -2
StartIdx = 0,   EndIdx = 0

Actual output :



```
sneha@sneha MINGW64 ~/workspace/college/sem3/daaLab
$ py maximumSumSubarray.py
input array is : [-2]
maximum sum subarray is [-2]
Maximum Subarray Sum: -2 starting at '0' and ending at '0'
```

Analysis of time and space complexity.

1. Time Complexity

The time complexity of the Divide and Conquer algorithm for finding the maximum subarray sum can be understood through its process:

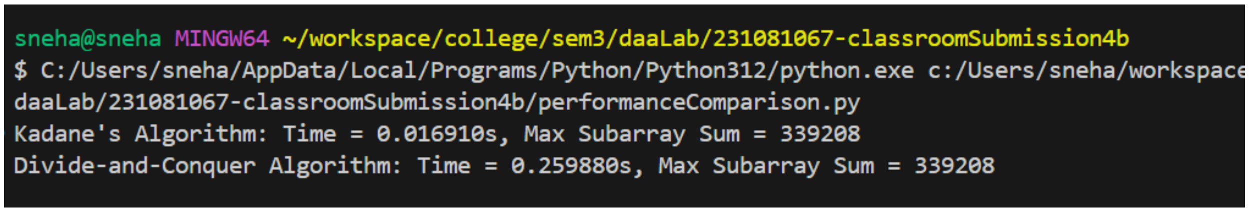
1. Divide the Array:
   - The algorithm recursively divides the input array into two halves until it reaches subarrays of size one. Each division step takes constant time.
2. Conquer the Subarrays:
   - For each pair of subarrays (left and right), the algorithm makes two recursive calls to find the maximum subarray sum in both halves. Therefore, the number of recursive calls grows exponentially, resembling a binary tree structure.
3. Combine the Results:
   - After calculating the maximum subarray sums for the left and right halves, the algorithm computes the maximum subarray sum that crosses the midpoint. This involves two linear scans: one to find the maximum sum in the left half and another to find the maximum sum in the right half, resulting in linear time complexity, O(n), for this step.
4. Overall Complexity:
   - As a result, the overall time complexity can be expressed as:
     - The time taken to divide the array: O(logn) for the recursive divisions.
     - The time taken to combine the results: O(n) for the crossing sum calculations.
   - Hence, the total time complexity of the algorithm is: O(nlogn)

2. Space Complexity

The space complexity of the algorithm is primarily determined by the following factors:

1. Recursive Call Stack:
   - Since the algorithm uses recursion, each recursive call adds a new layer to the call stack. The maximum depth of the recursion occurs when the array is divided down to single elements, which leads to a logarithmic growth in the number of calls. Therefore, the space used by the call stack is O(logn).
2. Auxiliary Space:
   - The algorithm itself does not use any additional data structures that grow with the size of the input. The space utilized for variables (e.g., sums and indices) within each function call remains constant, contributing O(1) to the space complexity.
3. Total Space Complexity:
   - Considering both the recursive call stack and the constant space used for variables, the total space complexity is: O(logn)

<u>Performance Comparison :</u>



```
sneha@sneha MINGW64 ~/workspace/college/sem3/daaLab/231081067-classroomSubmission4b
$ C:/Users/sneha/AppData/Local/Programs/Python/Python312/python.exe c:/Users/sneha/workspace
daaLab/231081067-classroomSubmission4b/performanceComparison.py
Kadane's Algorithm: Time = 0.016910s, Max Subarray Sum = 339208
Divide-and-Conquer Algorithm: Time = 0.259880s, Max Subarray Sum = 339208
```

The input array is an array of random numbers of size 10^5 .

<u>Comparison of the Divide-and-Conquer approach with Kadane's algorithm.</u>

Aspect	**Divide-and-Conquer**	**Kadane's Algorithm**

Approach	Recursive, splits array	Iterative, single pass
Time Complexity	O(nlogn)	O(n)
Space Complexity	O(logn) due to recursion call stack	O(1)
Ease of Implementation	Moderate to complex	Simple