- Complexity Analysis of Sub_Array_Max_Sum

$$T(n) = 2T(n/2) + n \quad \text{——} \; (1)$$

Comparing Equat^n 1 with the following Equat^n

$$T(n) = aT(n/b) + n^d$$

we get

$$a = 2, \; b = 2, \; d = 1$$

$$\log_b a = \log_2 2 = 1.$$

$$1 = 1$$

$$\therefore T(n) = \Theta(n \log n).$$

The Complexity of this algorithm is O(N log N) as calculated above.

- Correctness Analysis

- The correctness analysis of an algorithm can be measured by the correct output produced by algorithm every time the Program runs.
- So for correctness analysis, the program should give correct output everytime which can be seen by the screenshots of various test case run.

Explanation of the algorithm

- We have two functions namely Sub_Array_Max_Sum and Sub_Array_Cross_Sum

- Its a recursion algorithm in which we check the left part where the maxsubarray having maximum sum is present.
- As we do this for left side, we do it for right side too, ~~or~~ it can be present across the left and right sides.

- m = l+h/2 finds the midpoint to divide the given array into left & right part.

- In the ~~ma~~ Sub_Array_Max_Sum method we check if the subarray having maxSum is present in left part by function call Sub_Array_Max_Sum(arr, l, m) and simillarly we check for the right side with function call Sub_Array_Max_Sum(arr, m+1, h).

- ~~Now~~ Similarly for checking if it exists across the array we have the function call Sub_Array_Cross_Sum

- In this method we use two for loops and compute the sum of the maxsubarray across the left and right part.