The maximum subarray problem gives us an array of integers and asks us to return the sum of the contiguous subarray of numbers which has the largest sum.

E.g.
Input = [1, 0, 2, 3, -2, 5, -9, 4]
Output = 9 ([1, 0, 2, 3, -2, 5])

There are 2 ways to do this:

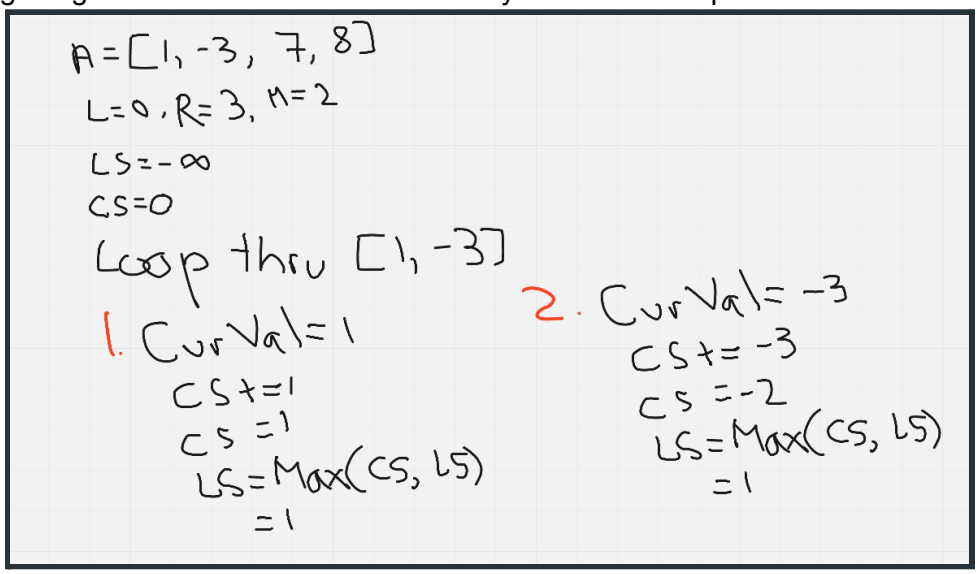
1. **Kadane's Algorithm:**
   - Time Complexity: O(n)
   - Code:

```
def maxSubArray(array):
  maxSum = float("-inf")
  curSum = 0

  for i in range(len(array)):
    curSum += array[i]
    maxSum = max(maxSum, curSum)

  return maxSum
```

2. **Divide And Conquer:**
   - Time Complexity: O(nlgn)
   - Code:

```
def maxSubArray(array):

  if(len(array) == 1):
    return array[0]

  left = 0
  right = len(array) - 1
  middle = (left + right) // 2

  return max(maxSubArray([:m+1]), maxSubArray([m+1:]),
             maxCrossingSum(array, left, right, middle))

def maxCrossingSum(array, left, right, middle):
  leftSum = float("-inf")
  curSum = 0

  for i in range(middle, left - 1, -1):
    curSum += array[i]
    leftSum = max(leftSum, curSum)

  rightSum = float("-inf")
  curSum = 0

  for i in range(middle + 1, right + 1):
    curSum += array[i]
    rightSum = max(rightSum, curSum)

  return max(leftSum, rightSum, leftSum + rightSum)
```

- Explanation of code:
  - In the function **maxSubArray**, I'm splitting the array into 2 equal (more or less) parts and returns the max sum in the left subarray, the right subarray or a subarray that straddles the division line.
  - In the function **maxCrossingSum**, I'm calculating the max sum in the left subarray, the right subarray or a subarray that straddles the division line.
  - One important thing to note is that when I'm calculating the max sum in the left subarray, I'm going from the middle to the first element. This is because, if you start from the left and go to the middle, you might end up getting a sum that's not from a subarray. See the example below.



In this example, we're adding the elements from left to right for the left subarray. Since 1 > -3, the leftSum value is 1. However, consider the return statement of **maxCrossingSum**: **return max(leftSum, rightSum, leftSum + rightSum)**. Here, **leftSum + rightSum** is not the sum of a subarray, as there is a -3 between the 1 and 7. Hence, the result would be false. Hence, for the left subarray, we have to add the elements starting from the middle and going to the left.