Maximum Subarray

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return *its sum*.

A **subarray** is a **contiguous** part of an array.

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6.
```

The naive method is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally, return the overall maximum. The time complexity of the Naive method is O(n^2).

Using **Divide and Conquer** approach, we can find the maximum subarray sum in O(nLogn) time. Following is the Divide and Conquer algorithm.

1. Divide the given array in two halves
2. Return the maximum of following three
    - Maximum subarray sum in left half (Make a recursive call)
    - Maximum subarray sum in right half (Make a recursive call)
    - Maximum subarray sum such that the subarray crosses the midpoint

```
3. static int maxCrossingSum(int arr[], int l, int m,
4.                                 int h)
5.     {
6.         // Include elements on left of mid.
7.         int sum = 0;
8.         int left_sum = Integer.MIN_VALUE;
9.         for (int i = m; i >= l; i--) {
10.                 sum = sum + arr[i];
11.                 if (sum > left_sum)
12.                     left_sum = sum;
13.             }
14.
15.             // Include elements on right of mid
16.             sum = 0;
17.             int right_sum = Integer.MIN_VALUE;
18.             for (int i = m + 1; i <= h; i++) {
19.                 sum = sum + arr[i];
20.                 if (sum > right_sum)
21.                     right_sum = sum;
22.             }
23.
```

```
            // Return sum of elements on left
            // and right of mid
            // returning only left_sum + right_sum will fail for
            // [-2, 1]
            return Math.max(left_sum + right_sum,
                            Math.max(left_sum, right_sum));
        }

        // Returns sum of maximum sum subarray
        // in aa[l..h]
        static int maxSubArraySum(int arr[], int l, int h)
        {
            // Base Case: Only one element
            if (l == h)
                return arr[l];

            // Find middle point
            int m = (l + h) / 2;

            /* Return maximum of following three
            possible cases:
            a) Maximum subarray sum in left half
            b) Maximum subarray sum in right half
            c) Maximum subarray sum such that the
            subarray crosses the midpoint */
            return Math.max(
                Math.max(maxSubArraySum(arr, l, m),
                         maxSubArraySum(arr, m + 1, h)),
                maxCrossingSum(arr, l, m, h));
        }

```

Kadane's Algorithm:

Initialize:

max_so_far = INT_MIN

max_ending_here = 0

Loop for each element of the array

(a) max_ending_here = max_ending_here + a[i]

(b) if(max_so_far < max_ending_here)

max_so_far = max_ending_here

(c) if(max_ending_here < 0)

max_ending_here = 0

return max_so_far

```
public int maxSubArray(int[] nums) {
        int maxsum=Integer.MIN_VALUE;
        int sum=0;
        int n=nums.length;
        for(int i=0;i<n;i++)
        {
           if(nums[i]>sum+nums[i])
           {
              sum=nums[i];
           }
           else
           {
              sum+=nums[i];
           }
           maxsum=Math.max(maxsum,sum);
        }
        return maxsum;
    }
```