# Question

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

**Example 1:**

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

**Output:** 6

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

**Example 2:**

**Input:** nums = [1]

**Output:** 1

**Example 3:**

**Input:** nums = [0]

**Output:** 0

**Example 4:**

**Input:** nums = [-1]

**Output:** -1

**Example 5:**

**Input:** nums = [-100000]

**Output:** -100000

**Constraints:**

* 1 <= nums.length <= 3 \* 104
* -105 <= nums[i] <= 105

**Follow up:** If you have figured out the O(n) solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

# Solution

## **Solution**

#### **Approach 1: Divide and Conquer**

**Intuition**

The problem is a classical example of [divide and conquer approach](https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/), and can be solved with the algorithm similar with the merge sort.

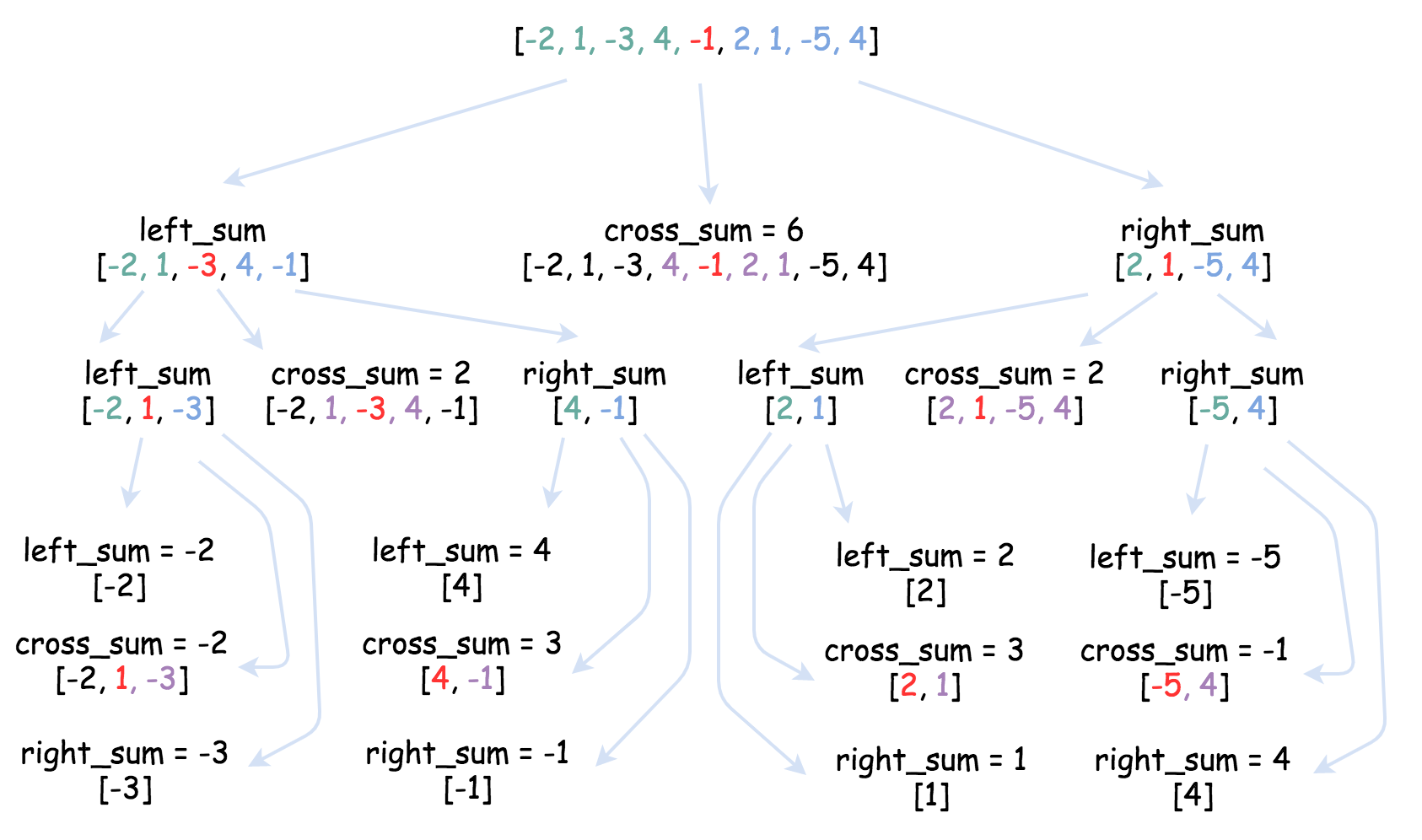
Let's follow here a solution template for the divide and conquer problems :

* Define the base case(s).
* Split the problem into subproblems and solve them recursively.
* Merge the solutions for the subproblems to obtain the solution for the original problem.

**Algorithm**

maxSubArray for array with n numbers:

* If n == 1 : return this single element.
* left\_sum = maxSubArray for the left subarray, i.e. for the first n/2 numbers (middle element at index (left + right) / 2 always belongs to the left subarray).
* right\_sum = maxSubArray for the right subarray, i.e. for the last n/2 numbers.
* cross\_sum = maximum sum of the subarray containing elements from both left and right subarrays and hence crossing the middle element at index (left + right) / 2.
* Merge the subproblems solutions, i.e. return max(left\_sum, right\_sum, cross\_sum).



**Implementation**

|  |
| --- |
| class Solution {  public int crossSum(int[] nums, int left, int right, int p) {  if (left == right) return nums[left];  int leftSubsum = Integer.MIN\_VALUE;  int currSum = 0;  for(int i = p; i > left - 1; --i) {  currSum += nums[i];  leftSubsum = Math.max(leftSubsum, currSum);  }  int rightSubsum = Integer.MIN\_VALUE;  currSum = 0;  for(int i = p + 1; i < right + 1; ++i) {  currSum += nums[i];  rightSubsum = Math.max(rightSubsum, currSum);  }  return leftSubsum + rightSubsum;  }  public int helper(int[] nums, int left, int right) {  if (left == right) return nums[left];  int p = (left + right) / 2;  int leftSum = helper(nums, left, p);  int rightSum = helper(nums, p + 1, right);  int crossSum = crossSum(nums, left, right, p);  return Math.max(Math.max(leftSum, rightSum), crossSum);  }  public int maxSubArray(int[] nums) {  return helper(nums, 0, nums.length - 1);  }  } |

|  |
| --- |
| class Solution:  def cross\_sum(self, nums, left, right, p):  if left == right:  return nums[left]  left\_subsum = float('-inf')  curr\_sum = 0  for i in range(p, left - 1, -1):  curr\_sum += nums[i]  left\_subsum = max(left\_subsum, curr\_sum)  right\_subsum = float('-inf')  curr\_sum = 0  for i in range(p + 1, right + 1):  curr\_sum += nums[i]  right\_subsum = max(right\_subsum, curr\_sum)  return left\_subsum + right\_subsum    def helper(self, nums, left, right):  if left == right:  return nums[left]    p = (left + right) // 2    left\_sum = self.helper(nums, left, p)  right\_sum = self.helper(nums, p + 1, right)  cross\_sum = self.cross\_sum(nums, left, right, p)    return max(left\_sum, right\_sum, cross\_sum)    def maxSubArray(self, nums: 'List[int]') -> 'int':  return self.helper(nums, 0, len(nums) - 1) |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N \log N)O(*N*log*N*). Let's compute the solution with the help of [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) T(N) = aT\left(\frac{b}{N}\right) + \Theta(N^d)*T*(*N*)=*aT*(*Nb*​)+Θ(*Nd*). The equation represents dividing the problem up into a*a* subproblems of size \frac{N}{b}*bN*​ in \Theta(N^d)Θ(*Nd*) time. Here one divides the problem in two subproblemes a = 2, the size of each subproblem (to compute left and right subtree) is a half of initial problem b = 2, and all this happens in a \mathcal{O}(N)O(*N*) time d = 1. That means that \log\_b(a) = dlog*b*​(*a*)=*d* and hence we're dealing with [case 2](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Application_to_common_algorithms) that means \mathcal{O}(N^{\log\_b(a)} \log N) = \mathcal{O}(N \log N)O(*N*log*b*​(*a*)log*N*)=O(*N*log*N*) time complexity.
* Space complexity : \mathcal{O}(\log N)O(log*N*) to keep the recursion stack.

#### **Approach 2: Greedy**

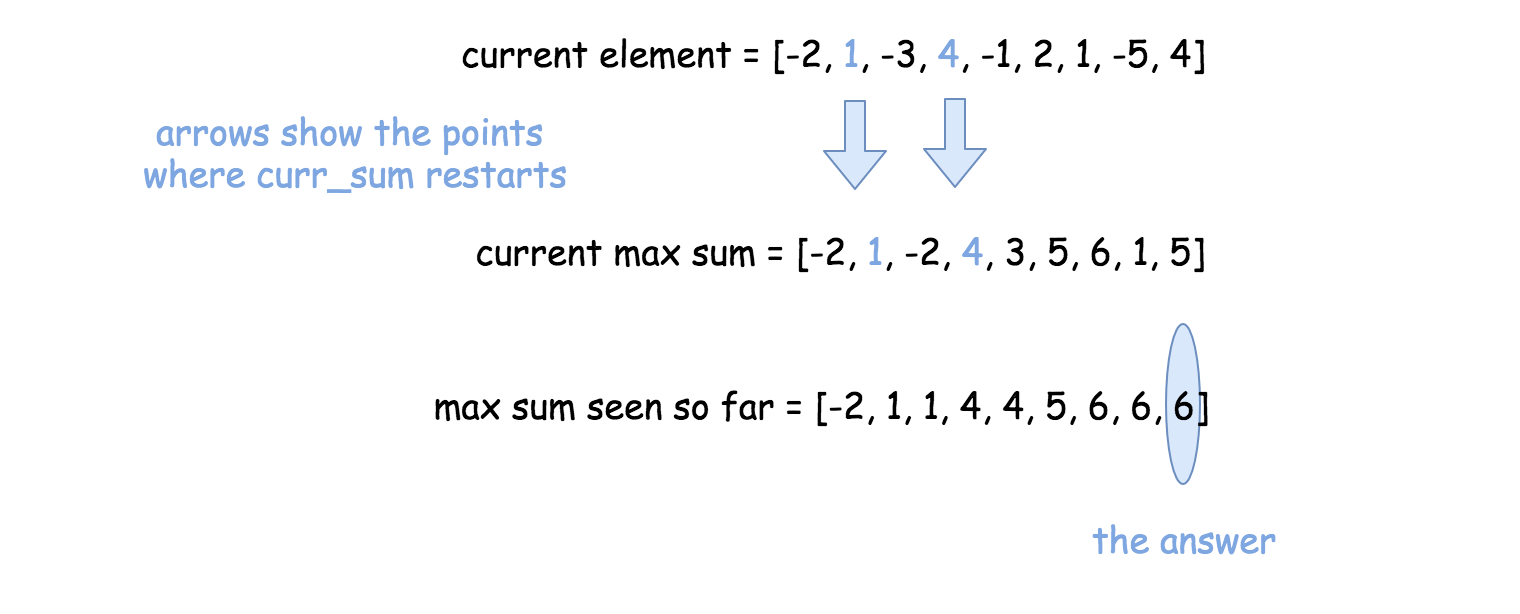
**Intuition**

The problem to find maximum (or minimum) element (or sum) with a single array as the input is a good candidate to be solved by the greedy approach in linear time. One can find the examples of linear time greedy solutions in our articles of  
[Super Washing Machines](https://leetcode.com/articles/super-washing-machines/), and [Gas Problem](https://leetcode.com/articles/gas-station/).

Pick the locally optimal move at each step, and that will lead to the globally optimal solution.

The algorithm is general and straightforward: iterate over the array and update at each step the standard set for such problems:

* current element
* current local maximum sum (at this given point)
* global maximum sum seen so far.



**Implementation**

|  |
| --- |
| class Solution {  public int maxSubArray(int[] nums) {  int n = nums.length;  int currSum = nums[0], maxSum = nums[0];  for(int i = 1; i < n; ++i) {  currSum = Math.max(nums[i], currSum + nums[i]);  maxSum = Math.max(maxSum, currSum);  }  return maxSum;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since it's one pass along the array.
* Space complexity : \mathcal{O}(1)O(1), since it's a constant space solution.

#### **Approach 3: Dynamic Programming (Kadane's algorithm)**

**Intuition**

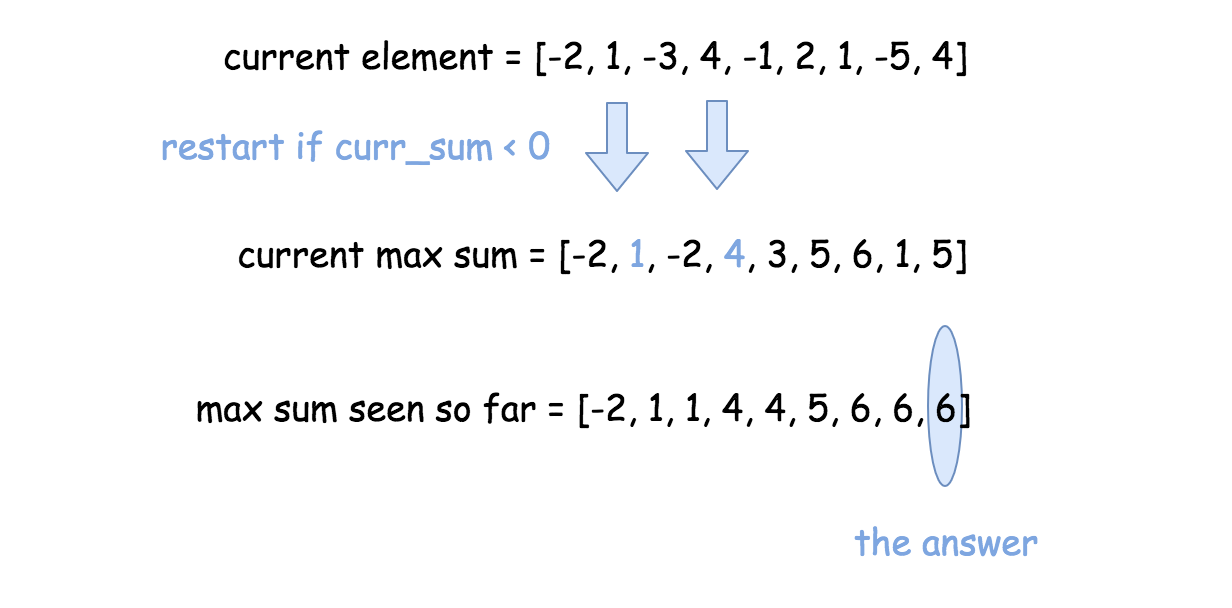
The problem to find sum or maximum or minimum in an entire array or in a fixed-size sliding window could be solved by the dynamic programming (DP) approach in linear time.

There are two standard DP approaches suitable for arrays:

* Constant space one. Move along the array and modify the array itself.
* Linear space one. First move in the direction left->right, then in the direction right->left. Combine the results. [Here is an example](https://leetcode.com/articles/sliding-window-maximum/).

Let's use here the first approach since one could modify the array to track the current local maximum sum at this given point.

Next step is to update the global maximum sum, knowing the local one.



**Implementation**

|  |
| --- |
| class Solution {  public int maxSubArray(int[] nums) {  int n = nums.length, maxSum = nums[0];  for(int i = 1; i < n; ++i) {  if (nums[i - 1] > 0) nums[i] += nums[i - 1];  maxSum = Math.max(nums[i], maxSum);  }  return maxSum;  }  } |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since it's one pass along the array.
* Space complexity : \mathcal{O}(1)O(1), since it's a constant space solution.