Given an integer array nums, find the subarray with the largest sum, and return *its sum*.

**Example 1:**

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

**Output:** 6

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

**Example 2:**

**Input:** nums = [1]

**Output:** 1

**Explanation:** The subarray [1] has the largest sum 1.

**Example 3:**

**Input:** nums = [5,4,-1,7,8]

**Output:** 23

**Explanation:** The subarray [5,4,-1,7,8] has the largest sum 23.

Denote the numbers by . Let denote the maximum subarray sum starting at . Then, the boundary condition is given by . For , we have the recursion

To justify this recursion, we argue as follows: is the optimal sum starting at , which means it is the sum of some subarray . If we prepend onto this subarray, we might expect that is the optimal subarray starting at , so that , but this is not be true if is negative. In this case, it would be better for to stand alone so that .

To find the maximum among all subarrays, we return . One naïve attempt to code the above idea is the following:

class Solution:

    def maxSubArray(self, nums: List[int]) -> int:

        n = len(nums)

        @cache

        def v(i):

            if i == n-1:

                return nums[n-1]

            else:

                return nums[i] + max(v(i+1), 0)

        return max(v(i) for i in range(n))

This will pass Leetcode’s complexity and memory criteria, but it can be improved by using a for-loop in place of recursion:

class Solution:

    def maxSubArray(self, nums: List[int]) -> int:

        n = len(nums)

        v = [None] \* n

        v[n-1] = nums[n-1]

        for i in reversed(range(n-1)):

            v[i] = nums[i] + max(v[i+1], 0)

        return max(v)

This second version runs in time and takes memory (just as the first version), but we can reduce the memory to by overwriting variables. Note that if we define , then satisfies the recursion . The third and final version of the code reads

class Solution:

    def maxSubArray(self, nums: List[int]) -> int:

        n = len(nums)

        v = nums[n-1]

        M = nums[n-1]

        for i in reversed(range(n-1)):

            v = nums[i] + max(v, 0)

            M = max(M, v)

        return M

This is known as *Kadane’s algorithm*.