Leetcode 53 – Maximum Subarray

```
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 $[a_0, \ldots, a_{n-1}]$. Let v_i denote the maximum subarray sum starting at a_i. Then, the boundary condition is given by $v_{n-1} = a_{n-1}$. For $i < n-1$, we have the recursion

$$v_i = \max\{a_i + v_{i+1}, a_i\}$$

To justify this recursion, we argue as follows: v_{i+1} is the optimal sum starting at a_{i+1}, which means it is the sum of some subarray $[a_{i+1}, \ldots, a_j]$. If we prepend a_i onto this subarray, we might expect that $[a_i, a_{i+1}, \ldots, a_j]$ is the optimal subarray starting at a_i, so that $v_i = a_i + v_{i+1}$, but this is not be true if v_{i+1} is negative. In this case, it would be better for a_i to stand alone so that $v_i = a_i$.

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

```python
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        n = len(nums)

        @cache
        def v(i):
            if i == n-1:
                return nums[n-1]

            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:

```python
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 $O(n)$ time and takes $O(n)$ memory (just as the first version), but we can reduce the memory to $O(1)$ by overwriting variables. Note that if we define $M_i = \max_{j \geq i} v_j$, then M_i satisfies the recursion $M_i = \max\{M_{i+1}, v_i\}$. 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*.