53. Maximum Subarray

June 4, 2019 | 174.7K views

Average Rating: 3.94 (88 votes)

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

Example:

```
Input: [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

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

Approach 1: Divide and Conquer

Intuition

The problem is a classical example of divide and conquer approach, and can be solved with the algorithm similar with the merge sort.

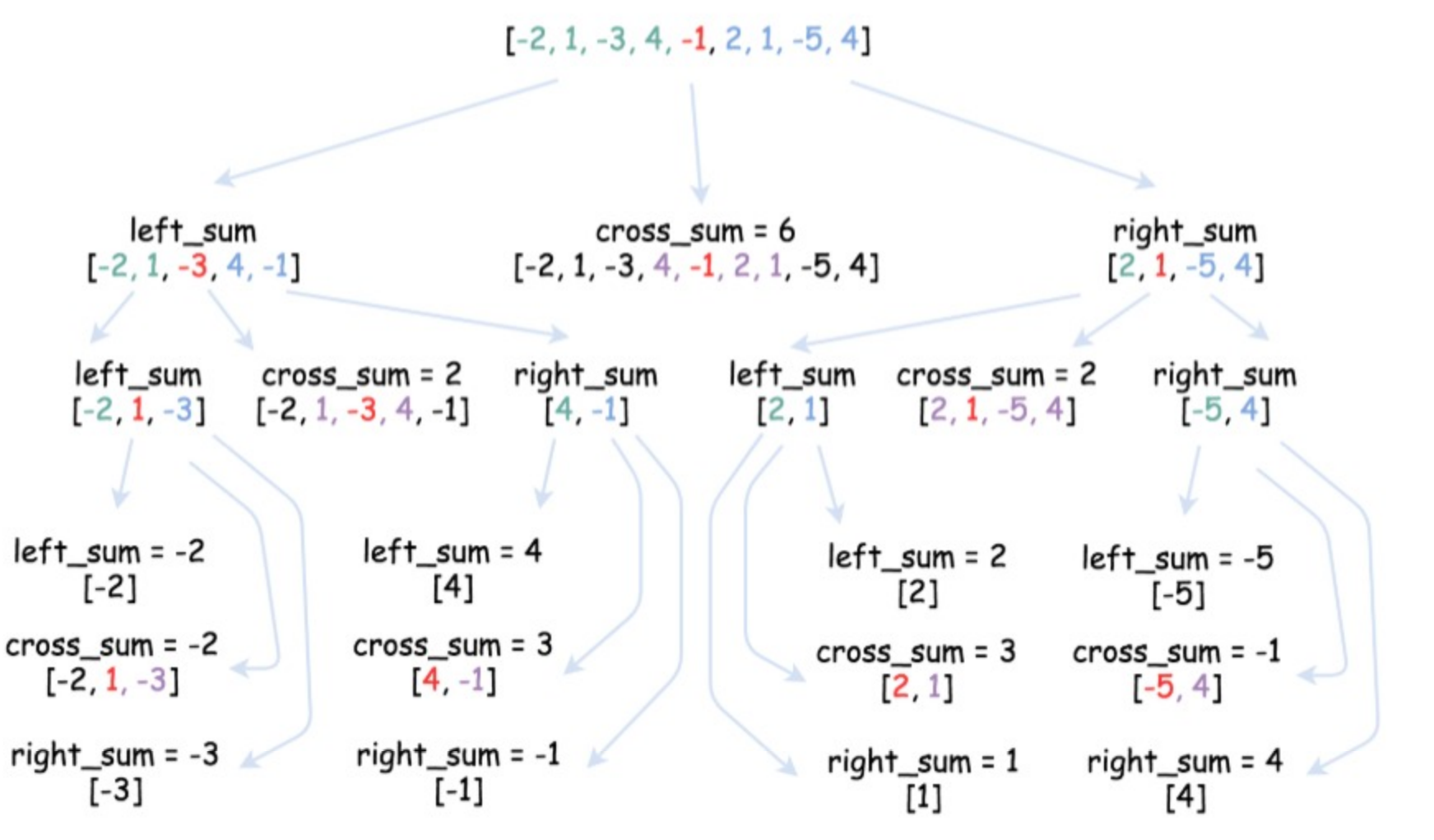
Let's follow here a solution template for the divide and conquer problems :

- 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

Java Python Copy

```python
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)
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N \log N)$. Let's compute the solution with the help of master theorem $T(N) = aT\left(\frac{b}{N}\right) + \Theta(N^d)$. The equation represents dividing the problem up into a subproblems of size $\frac{N}{b}$ in $\Theta(N^d)$ 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 $\mathcal{O}(N)$ time `d = 1`. That means that $\log_b(a) = d$ and hence we're dealing with case 2 that means $\mathcal{O}(N^{\log_b(a)} \log N) = \mathcal{O}(N \log N)$ time complexity.
- Space complexity : $\mathcal{O}(\log N)$ to keep the recursion stack.

Approach 2: Greedy

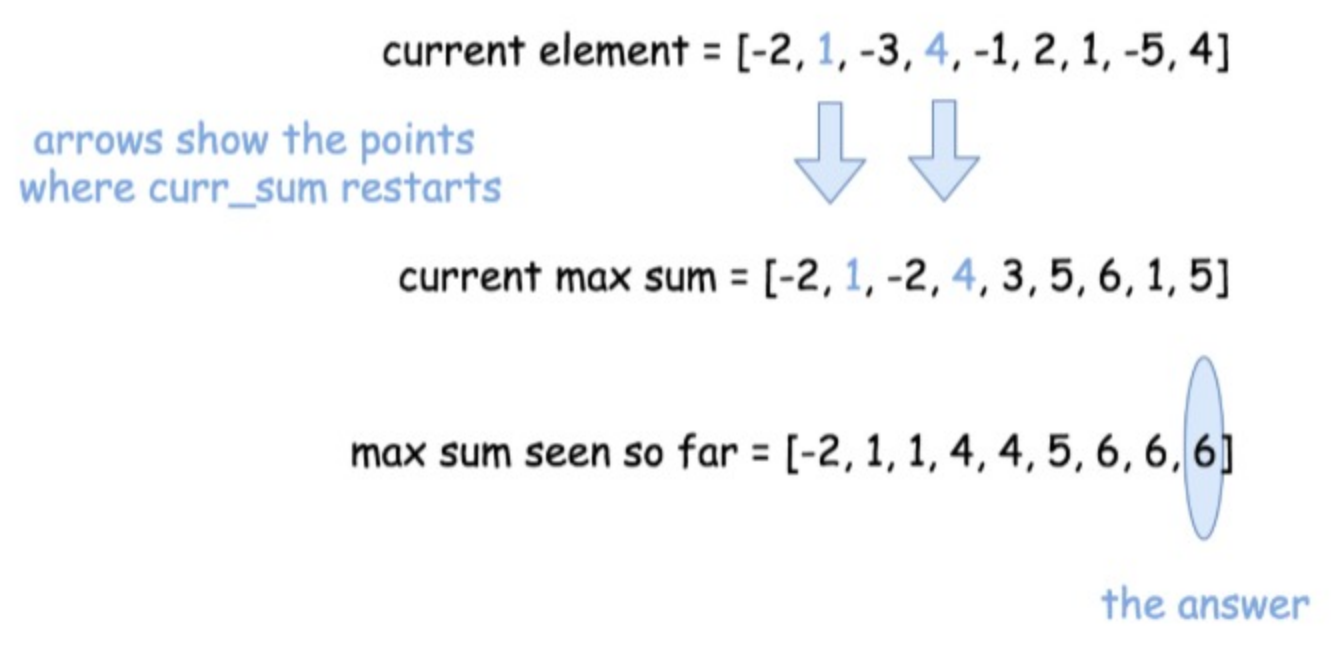
Intuition

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, and Gas Problem.

> 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

Java Python Copy

```python
class Solution:
    def maxSubArray(self, nums: 'List[int]') -> 'int':
        n = len(nums)
        curr_sum = max_sum = nums[0]

        for i in range(1, n):
            curr_sum = max(nums[i], curr_sum + nums[i])
            max_sum = max(max_sum, curr_sum)

        return max_sum
```

Complexity Analysis

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

Approach 3: Dynamic Programming (Kadane's algorithm)

Intuition

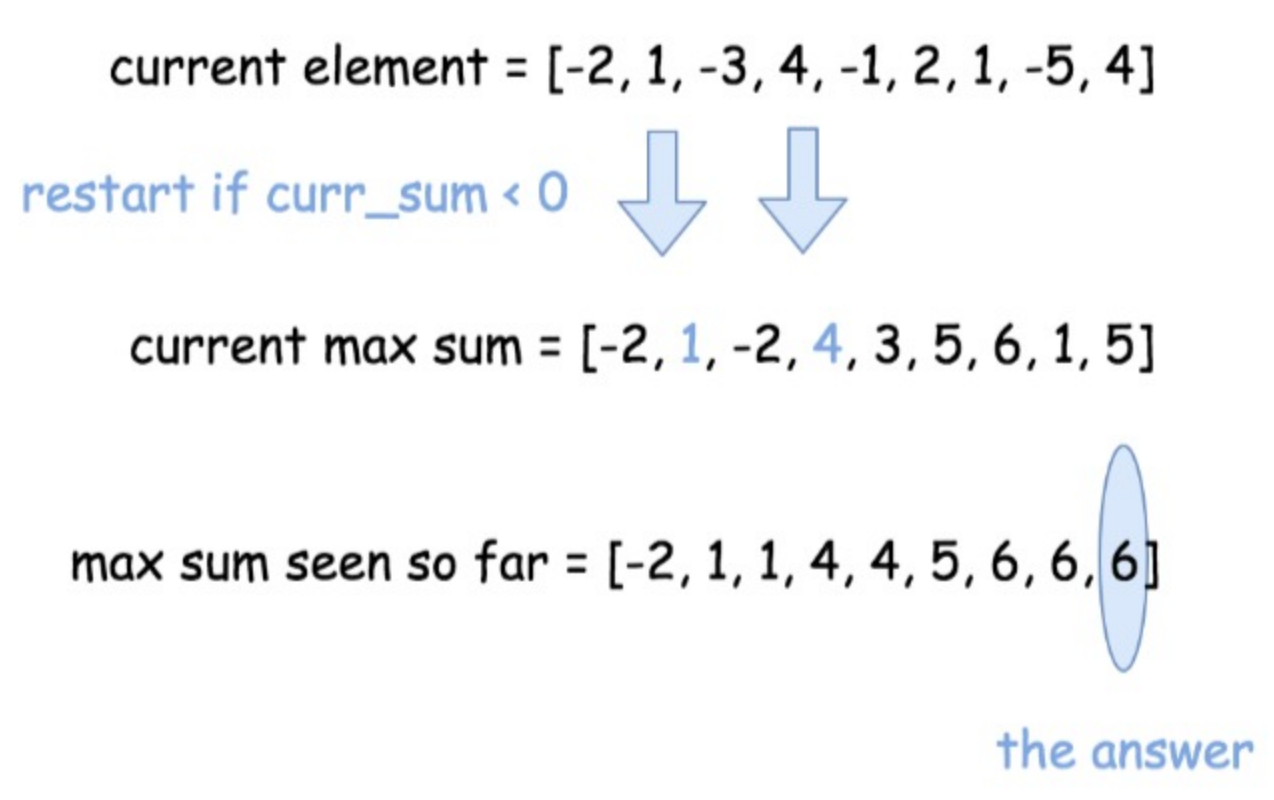
The problem to find sum or maximum or minimum in an entire array or in a fixed-size sliding window could be solved by the dynamic programming (DP) approach in linear time.

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.

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

Java Python Copy

```python
class Solution:
    def maxSubArray(self, nums: 'List[int]') -> 'int':
        n = len(nums)
        max_sum = nums[0]
        for i in range(1, n):
            if nums[i - 1] > 0:
                nums[i] += nums[i - 1]
            max_sum = max(nums[i], max_sum)

        return max_sum
```

Complexity Analysis

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

Rate this article:

Previous Next

Comments: 67

Sort By

Type comment here... (Markdown is supported)

Preview Post

ghost204nit ★ 823 October 9, 2019 5:34 AM
How in the world this question is marked 'easy'? :/
752 Share Reply
SHOW 12 REPLIES

yc2523 ★ 122 August 22, 2019 11:40 PM
I can't see the difference between approach 2 and 3
110 Share Reply
SHOW 5 REPLIES

alex970448359 ★ 75 August 24, 2019 2:35 PM
The question is easy. But these approaches are not ... lol
75 Share Reply

totsubo ★ 730 June 20, 2019 12:49 PM
The diagram is incorrect.
42 Share Reply
SHOW 2 REPLIES

alexishe ★ 234 August 18, 2019 11:45 PM
the greedy method looks very similar to the dynamic programming one
36 Share Reply

amanzholovm ★ 40 February 10, 2020 7:53 AM
I don't find it reasonable to ask for a divide and conquer solution after a person finds a better solution.
35 Share Reply

trilllzz ★ 27 August 29, 2019 2:33 AM Report
Alternative way to think about it (similar to 2):

Use a sliding window and if the number at the right index equals or is greater to the current accumulation of the window, that window is irrelevant to the problem so move the left index to the right index and accumulate again.

Read More
15 Share Reply
SHOW 2 REPLIES

btbam91 ★ 59 July 4, 2019 9:10 AM
I don't like Approach 3 because the input array is being modified.
13 Share Reply
SHOW 4 REPLIES

starlord ★ 50 August 3, 2019 1:47 PM Report
the approach 3 is not sufficient to approve the algorithm is correct. Assume [a, b, c, d, e] as input array, [a, b, c] added to 0, then it will check [d, e], but it didn't approve [b, c, d, e] could also greater than [d, e]
4 Share Reply
SHOW 1 REPLY

yhecoder ★ 3 10 hours ago
After spending two hours, I can only solve this in O(N^2) time -- and my Google onsight is in two days. Lord have mercy && miracle.
2 Share Reply