

[◀ Previous \(/articles/super-washing-machines/\)](/articles/super-washing-machines/)   [Next ▶ \(/articles/inorder-successor-in-a-bst-ii/\)](/articles/inorder-successor-in-a-bst-ii/)

# 53. Maximum Subarray (/problems/maximum-subarray/)

June 4, 2019 | 154.5K views

Average Rating: 4.08 (79 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 (<https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/>), and can be solved with the algorithm similar with the merge sort.

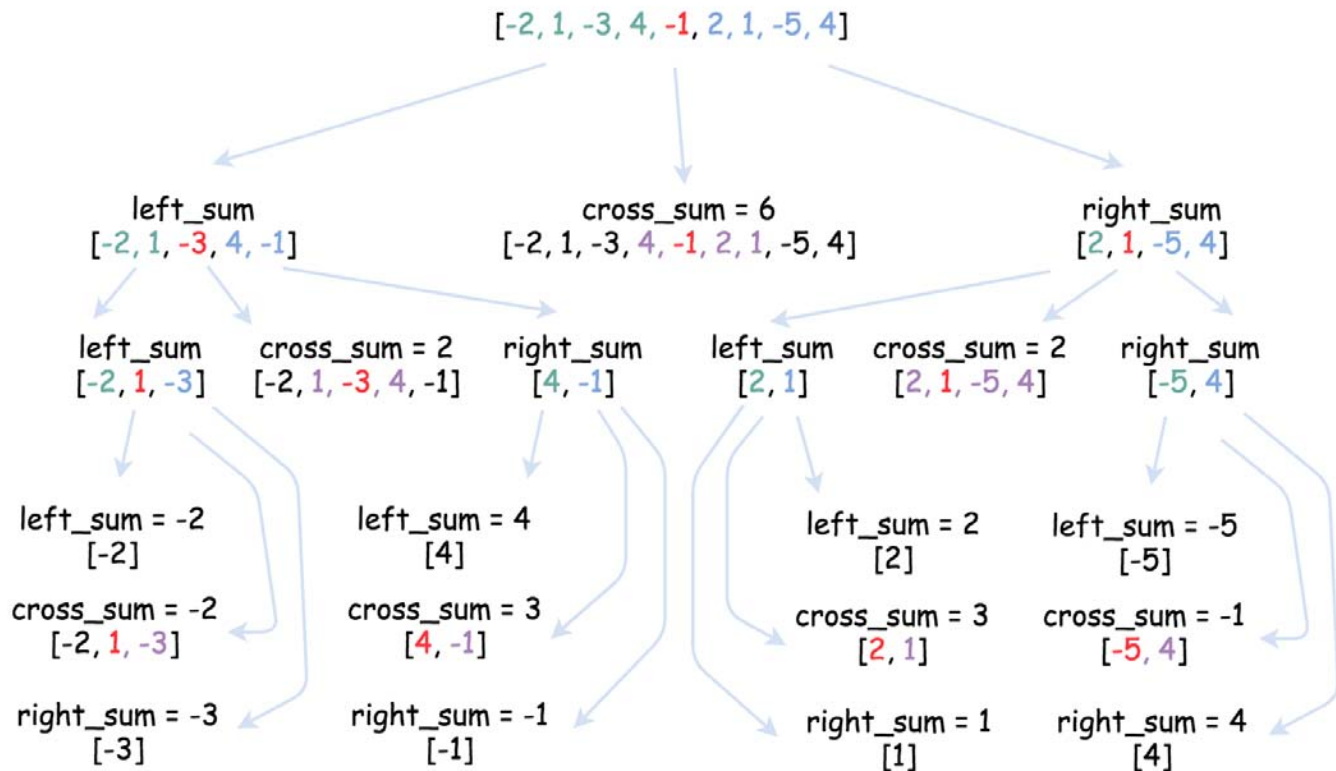
Let's follow here a solution template for the divide and conquer problems : [Articles > 53. Maximum Subarray](#) ▼

- 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

Articles > 53. Maximum Subarray  Copy

```

1 class Solution:
2     def cross_sum(self, nums, left, right, p):
3         if left == right:
4             return nums[left]
5
6         left_subsum = float('-inf')
7         curr_sum = 0
8         for i in range(p, left - 1, -1):
9             curr_sum += nums[i]
10            left_subsum = max(left_subsum, curr_sum)
11
12            right_subsum = float('-inf')
13            curr_sum = 0
14            for i in range(p + 1, right + 1):
15                curr_sum += nums[i]
16                right_subsum = max(right_subsum, curr_sum)
17
18            return left_subsum + right_subsum
19
20     def helper(self, nums, left, right):
21         if left == right:
22             return nums[left]
23
24         p = (left + right) // 2
25
26         left_sum = self.helper(nums, left, p)
27         right_sum = self.helper(nums, p + 1, right)
28         cross_sum = self.cross_sum(nums, left, right, p)
29
30         return max(left_sum, right_sum, cross_sum)
31
32     def maxSubArray(self, nums: 'List[int]') -> 'int':
33         return self.helper(nums, 0, len(nums) - 1)

```

## Complexity Analysis

- Time complexity :  $\mathcal{O}(N \log N)$ . Let's compute the solution with the help of master theorem ([https://en.wikipedia.org/wiki/Master\\_theorem\\_\(analysis\\_of\\_algorithms\)](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)))  $T(N) = aT\left(\frac{b}{N}\right) + \mathcal{O}(N^d)$ . The equation represents dividing the problem up into  $a$  subproblems of size  $\frac{N}{b}$  in  $\mathcal{O}(N^d)$  time. Here one divides the problem in two subproblems  $a = 2$ , 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 ([https://en.wikipedia.org/wiki/Master\\_theorem\\_\(analysis\\_of\\_algorithms\)#Application\\_to\\_common\\_algorithms](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#Application_to_common_algorithms)) that means  $\mathcal{O}(N^{\log_b(a)} \log N) = \mathcal{O}(N \log N)$  time complexity.

- Space complexity :  $\mathcal{O}(\log N)$  to keep the recursion stack.  Articles > 53. Maximum Subarray ▼

## Approach 2: Greedy

### Intuition

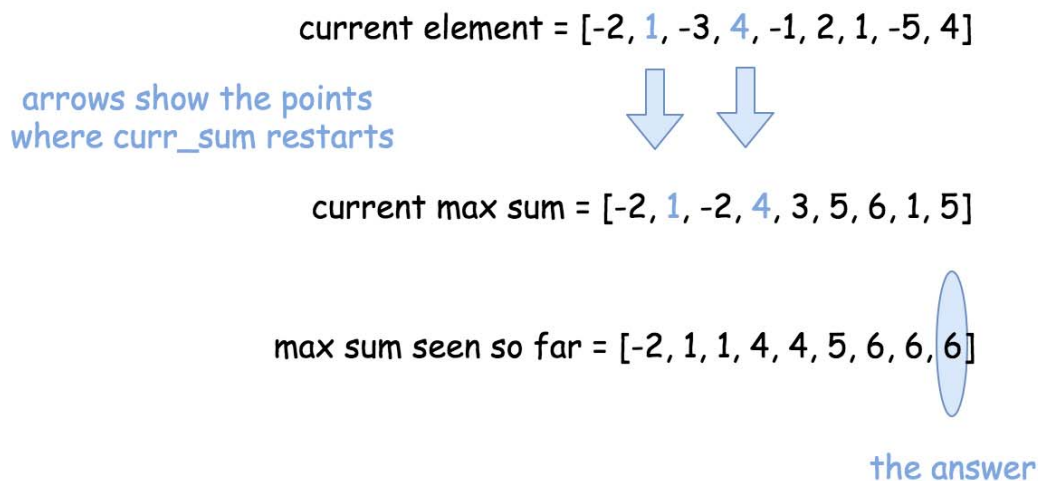
The problem to find maximum (or minimum) element (or sum) with a single array as the input is a good candidate to be solved by the greedy approach in linear time. One can find the examples of linear time greedy solutions in our articles of

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

Java

Python

Articles &gt; 53. Maximum Subarray



```
1 class Solution:
2     def maxSubArray(self, nums: 'List[int]') -> 'int':
3         n = len(nums)
4         curr_sum = max_sum = nums[0]
5
6         for i in range(1, n):
7             curr_sum = max(nums[i], curr_sum + nums[i])
8             max_sum = max(max_sum, curr_sum)
9
10        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:

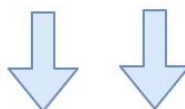
- Constant space one. Move along the array and modify the array itself.
- Linear space one. First move in the direction `left->right`, then in the direction `right->left`. Combine the results. Here is an example (<https://leetcode.com/articles/sliding-window-maximum/>).

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.

current element = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

restart if `curr_sum < 0`



current max sum = [-2, 1, -2, 4, 3, 5, 6, 1, 5]

Articles > 53. Maximum Subarray ▾

max sum seen so far = [-2, 1, 1, 4, 4, 5, 6, 6, 6]

the answer

## Implementation

Java

Python

Copy

```
1 class Solution:
2     def maxSubArray(self, nums: 'List[int]') -> 'int':
3         n = len(nums)
4         max_sum = nums[0]
5         for i in range(1, n):
6             if nums[i - 1] > 0:
7                 nums[i] += nums[i - 1]
8             max_sum = max(nums[i], max_sum)
9
10        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 (/articles/super-washing-machines/)

Next (/articles/inorder-successor-in-a-bst-ii/)

Comments: 60

Sort By ▾



Type comment here... (Markdown is supported)

Preview

Post



ghost204nit (/ghost204nit) ★ 668 🕒 October 8, 2019 8:04 PM

Articles > 53. Maximum Subarray ▾

How in the world this question is marked 'easy'? :/

(/ghost204nit)

604 ^ v | Share | Reply

SHOW 10 REPLIES



yc2523 (/yc2523) ★ 108 🕒 August 22, 2019 2:10 PM

I can't see the difference between approach 2 and 3

(/yc2523)

96 ^ v | Share | Reply

SHOW 3 REPLIES



alex970448359 (/alex970448359) ★ 66 🕒 August 24, 2019 5:05 AM

The question is easy. But these approaches are not ... lol

(/alex970448359)

66 ^ v | Share | Reply



totsubo (/totsubo) ★ 639 🕒 June 20, 2019 3:19 AM

The diagram is incorrect.

(/totsubo)

39 ^ v | Share | Reply

SHOW 2 REPLIES



alexishe (/alexishe) ★ 207 🕒 August 18, 2019 2:15 PM

the greedy method looks very similar to the dynamic programming one

(/alexishe)

32 ^ v | Share | Reply



amanzholovm (/amanzholovm) ★ 19 🕒 February 9, 2020 9:23 PM

I don't find it reasonable to ask for a divide and conquer solution after a person finds a better solution.

(/amanzholovm)

17 ^ v | Share | Reply



trilllzz (/trilllzz) ★ 26 🕒 August 28, 2019 5:03 PM

Alternative way to think about it (similar to 2):

(/trilllzz)

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

14 ^ v | Share | Reply

SHOW 2 REPLIES



(/btbam91)

btbam91 (/btbam91) ★ 54 🕒 July 3, 2019 11:40 PM

[Articles](#) > [53. Maximum Subarray](#) ▼

I don't like Approach 3 because the input array is being modified.

12 ▲ ▼ | [Share](#) | [Reply](#)

SHOW 3 REPLIES



(/starlord)

starlord (/starlord) ★ 50 🕒 August 3, 2019 4:17 AM

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



(/melodyjoyou)

MelodyJoyOu (/melodyjoyou) ★ 2 🕒 February 1, 2020 4:51 PM

I'm new to programming and I have a question for Approach 1- Divide and Conquer:  
Are all the possibilities of subarrays covered in this way? What if the largest sum is the sum by the elements with index 2 to 4?

2 ▲ ▼ | [Share](#) | [Reply](#)

SHOW 3 REPLIES

[<](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [>](#)