

239. Sliding Window Maximum

March 5, 2019 | 66.7K views

★★★★★
Average Rating: 4.56 (78 votes)

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position. Return the max sliding window.

Follow up:

Could you solve it in linear time?

Example:

Input: *nums* = [1,3,-1,-3,5,3,6,7], and *k* = 3
Output: [3,3,5,5,6,7]
Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- $1 \leq k \leq \text{nums.length}$

Solution

Approach 1: Use a hammer

Intuition

The straightforward solution is to iterate over all sliding windows and find a maximum for each window. There are $N - k + 1$ sliding windows and there are *k* elements in each window, that results in a quite bad time complexity $O(Nk)$.

Implementation

```
Java Python Copy
1 class Solution:
2     def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
3         n = len(nums)
4         if n * k == 0:
5             return []
6
7         return [max(nums[i:i + k]) for i in range(n - k + 1)]
```

Complexity Analysis

- Time complexity: $O(Nk)$, where *N* is number of elements in the array.
- Space complexity: $O(N - k + 1)$ for an output array.

Approach 2: Deque

Intuition

How one could improve the time complexity? The first idea is to use a *heap*, since in a maximum heap *heap[0]* is always the largest element. Though to add an element in a heap of size *k* costs $\log(k)$, that means $O(N \log(k))$ time complexity for the solution.

Could we figure out $O(N)$ solution?

Let's use a *deque* (double-ended queue), the structure which pops from / pushes to either side with the same $O(1)$ performance.

It's more handy to store in the deque indexes instead of elements since both are used during an array parsing.

Algorithm

The algorithm is quite straightforward :

- Process the first *k* elements separately to initiate the deque.
- Iterate over the array. At each step :
 - Clean the deque :
 - Keep only the indexes of elements from the current sliding window.
 - Remove indexes of all elements smaller than the current one, since they will not be the maximum ones.
 - Append the current element to the deque.
 - Append *deque[0]* to the output.
- Return the output array.

Implementation

```
Java Python Copy
1 from collections import deque
2 class Solution:
3     def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
4         # base cases
5         n = len(nums)
6         if n * k == 0:
7             return []
8         if k == 1:
9             return nums
10
11         def clean_deque(i):
12             # remove indexes of elements not from sliding window
13             if deque and deque[0] == i - k:
14                 deque.popleft()
15
16             # remove from deque indexes of all elements
17             # which are smaller than current element nums[i]
18             while deque and nums[i] > nums[deque[-1]]:
19                 deque.pop()
20
21         # init deque and output
22         deque = deque()
23         max_idx = 0
24         for i in range(k):
25             clean_deque(i)
26             deque.append(i)
27             # compute max in nums[:k]
```

Complexity Analysis

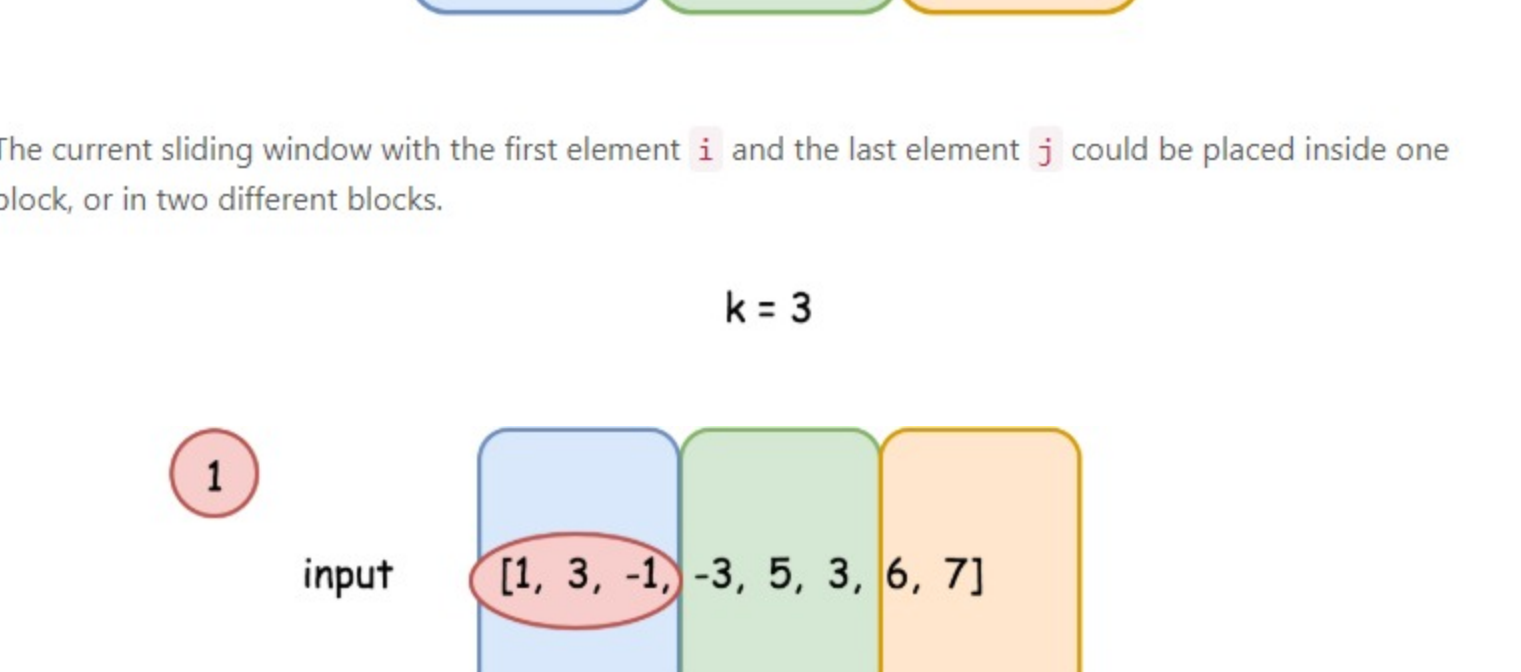
- Time complexity: $O(N)$, since each element is processed exactly twice - it's index added and then removed from the deque.
- Space complexity: $O(N)$, since $O(N - k + 1)$ is used for an output array and $O(k)$ for a deque.

Approach 3: Dynamic programming

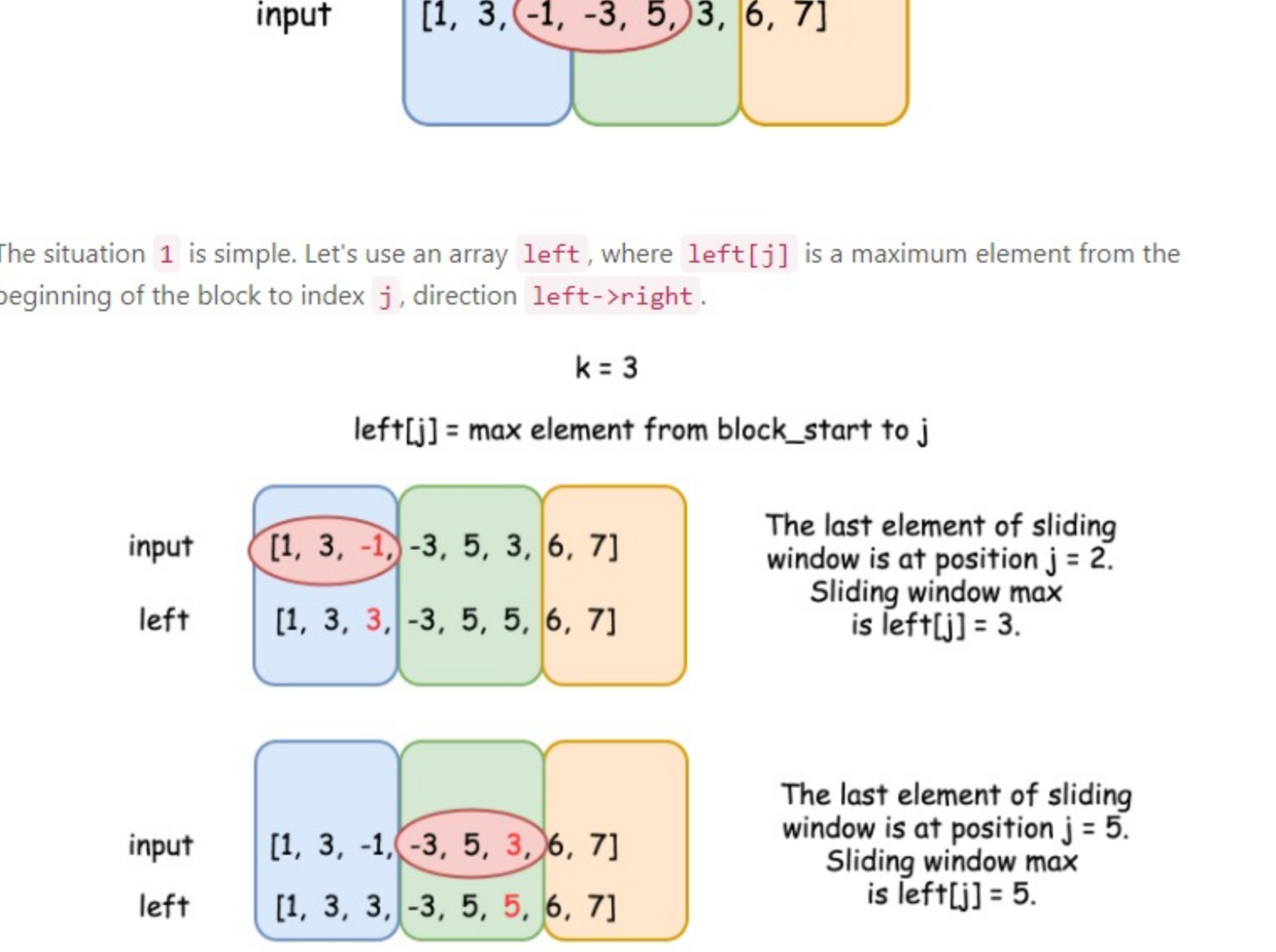
Intuition

Here is another $O(N)$ solution. The good thing about this solution is that you don't need any data structures but *array / list*.

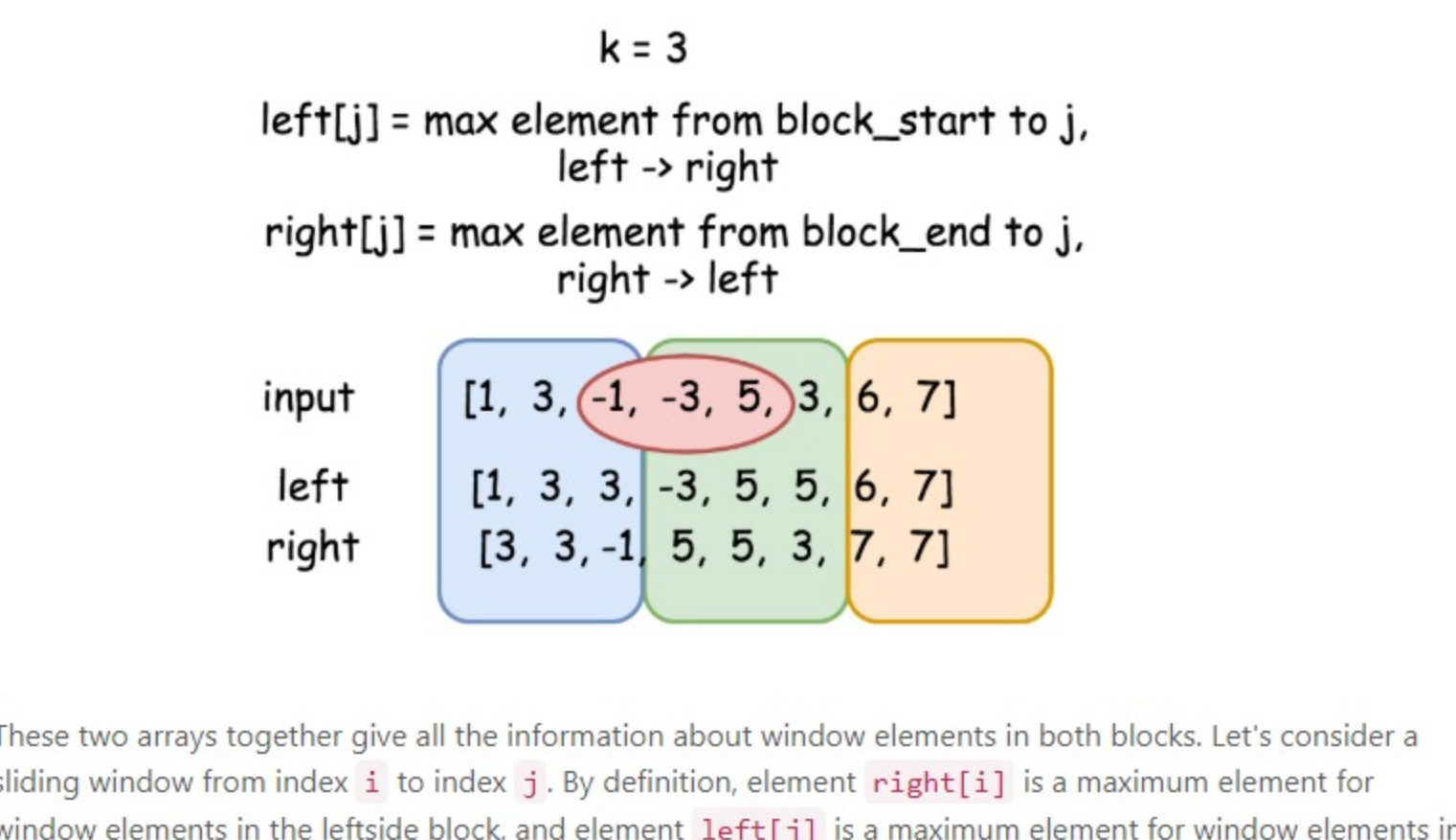
The idea is to split an input array into blocks of *k* elements. The last block could contain less elements if *n % k != 0*.



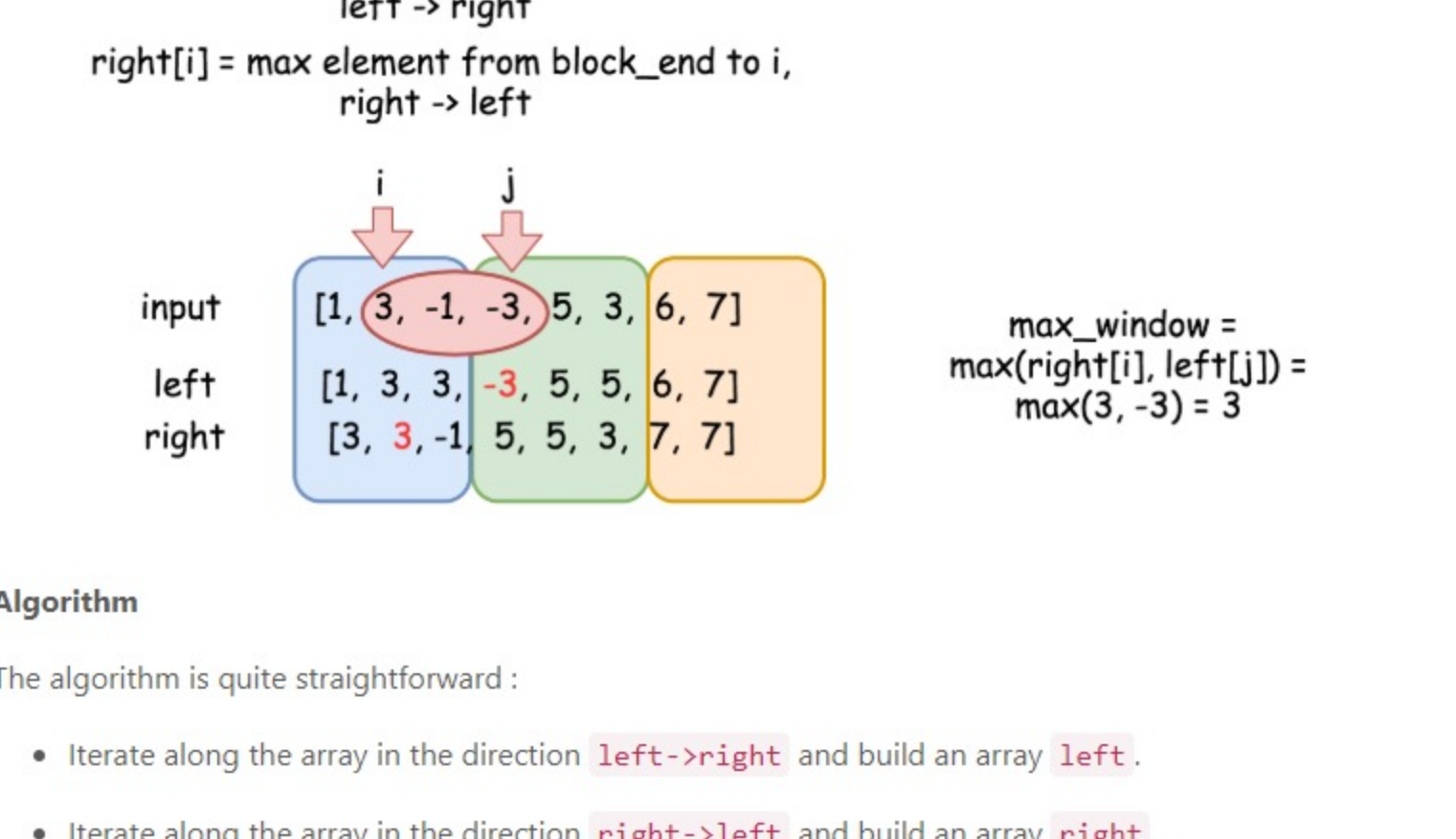
The current sliding window with the first element *i* and the last element *j* could be placed inside one block, or in two different blocks.



To work with more complex situation 2, let's introduce array *right*, where *right[j]* is a maximum element from the end of the block to index *j*, direction *right->left*. *right* is basically the same as *left*, but in the other direction.



These two arrays together give all the information about window elements in both blocks. Let's consider a sliding window from index *i* to index *j*. By definition, element *right[i]* is a maximum element for window elements in the leftside block, and element *left[j]* is a maximum element for window elements in the rightside block. Hence the maximum element in the sliding window is *max(right[i], left[j])*.



Algorithm

The algorithm is quite straightforward :

- Iterate along the array in the direction *left->right* and build an array *left*.
- Iterate along the array in the direction *right->left* and build an array *right*.
- Build an output array as *max(right[i], left[i + k - 1])* for *i* in range *(0, n - k + 1)*.

Implementation

```
Java Python Copy
1 class Solution:
2     def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
3         n = len(nums)
4         if n * k == 0:
5             return []
6         if k == 1:
7             return nums
8
9         left = [0] * n
10        left[0] = nums[0]
11        right = [0] * n
12        right[n - 1] = nums[n - 1]
13        for i in range(1, n):
14            # from left to right
15            if i % k == 0:
16                # block start
17                left[i] = nums[i]
18            else:
19                left[i] = max(left[i - 1], nums[i])
20            # from right to left
21            j = n - i - 1
22            if (j + 1) % k == 0:
23                # block end
24                right[j] = nums[j]
25            else:
26                right[j] = max(right[j + 1], nums[j])
27
```

Complexity Analysis

- Time complexity: $O(N)$, since all we do is 3 passes along the array of length *N*.
- Space complexity: $O(N)$ to keep *left* and *right* arrays of length *N*, and output array of length *N - k + 1*.

Rate this article: ★★★★★

Previous Next Copy

Comments: 47

Sort By ▾

Type comment here... (Markdown is supported)

Preview

Post

moonlight16 ★ 366 @ July 1, 2019 5:41 AM

Wow, when I read this sentence it totally confuse me!!!

How one could improve the time complexity? The first idea is to use a heap, since in a maximum heap heap[0] is always the largest element. Though to add an element in a heap of size k costs $\log(k)\log(k)$, that means $\ln(\ln(\ln(N))) \cdot \log(k)\log(k) \cdot \ln(N)\log(k)$ time complexity for the solution.

60 ▴ ▾ | Share | Reply

SHOW 9 REPLIES

eefiasfira ★ 595 @ July 8, 2019 8:23 AM

Great explanation. Please note that it is **misleading** to include the output array in the space complexity - this makes solution 2 and 3 seem identical when in fact solution 2 is superior.

Solution 1 takes $O(nk)$ time and $O(1)$ space

Solution 2 takes $O(n)$ time and $O(k)$ space

Solution 3 takes $O(n)$ time and $O(n)$ space

26 ▴ ▾ | Share | Reply

SHOW 7 REPLIES

ricace ★ 51 @ April 17, 2019 7:31 AM

I think the thought behind solution 2 is monotonous stack/queue, and here Deque is to poll from both head and tail

15 ▴ ▾ | Share | Reply

SHOW 1 REPLY

genehacker ★ 14 @ June 15, 2019 9:08 PM

For solution 2, we don't have to explicitly have a different way to keep track of max element, from 1..k and k..n separately. We can combine into same logic.

```
# init deque and output
deque = deque()
```

12 ▴ ▾ | Share | Reply

Read More

sankalp ★ 52 @ August 13, 2019 1:46 AM

Third solution is brilliant brother.

11 ▴ ▾ | Share | Reply

jianchao-5 ★ 14478 @ March 9, 2019 9:58 AM

The third solution is really elegant. Rewrite in C++.

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
```

8 ▴ ▾ | Share | Reply

Read More

SHOW 2 REPLIES

willye ★ 878 @ April 8, 2020 6:42 PM

Another way of explaining the deque method (approach 2): **You want to ensure the deque window only has decreasing elements. That way, the leftmost element is always the largest.**

7 ▴ ▾ | Share | Reply

lenchen1112 ★ 1005 @ February 25, 2020 11:21 AM

I don't think the approach 3 meet the description of this question. We are told that **You can only see the k numbers in the window**, which means we should treat it as a data stream instead of a static data list. Therefore we shouldn't do any preprocess on it.

7 ▴ ▾ | Share | Reply

winterchocolate ★ 7 @ July 28, 2019 7:55 PM

Shouldn't the heap implementation be $O(nk)$ not $O(n\log k)$ because removing a non maximal element is an $O(k)$ operation? Only the adding of an element is $O(\log k)$ in this implementation. Would appreciate it if someone could clarify this.

7 ▴ ▾ | Share | Reply

Report

SHOW 3 REPLIES

wong_le ★ 15 @ November 22, 2019 3:30 PM

How is approach 2 $O(n)$?

5 ▴ ▾ | Share | Reply

SHOW 1 REPLY

1

2

3

4

5