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 <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`
- `1 <= k <= 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 $\mathcal{O}(Nk)$.

Implementation

Java | Python

```python
class Solution:
    def maxSlidingWindow(self, nums: 'List[int]', k: 'int') -> 'List[int]':
        n = len(nums)
        if n * k == 0:
            return []

        return [max(nums[i:i + k]) for i in range(n - k + 1)]
```

Complexity Analysis

- Time complexity : $\mathcal{O}(Nk)$, where `N` is number of elements in the array.
- Space complexity : $\mathcal{O}(N - k + 1)$ 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 $\mathcal{O}(N \log(k))$ time complexity for the solution.

> Could we figure out $\mathcal{O}(N)$ solution?

Let's use a *deque* (double-ended queue), the structure which pops from / pushes to either side with the same $\mathcal{O}(1)$ 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 straigthforward :

- 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

```python
from collections import deque
class Solution:
    def maxSlidingWindow(self, nums: 'List[int]', k: 'int') -> 'List[int]':
        # base cases
        n = len(nums)
        if n * k == 0:
            return []
        if k == 1:
            return nums

        def clean_deque(i):
            # remove indexes of elements not from sliding window
            if deq and deq[0] == i - k:
                deq.popleft()

            # remove from deq indexes of all elements
            # which are smaller than current element nums[i]
            while deq and nums[i] > nums[deq[-1]]:
                deq.pop()

        # init deque and output
        deq = deque()
        max_idx = 0
        for i in range(k):
            clean_deque(i)
            deq.append(i)
            # compute max in nums[:k]
```

Complexity Analysis

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

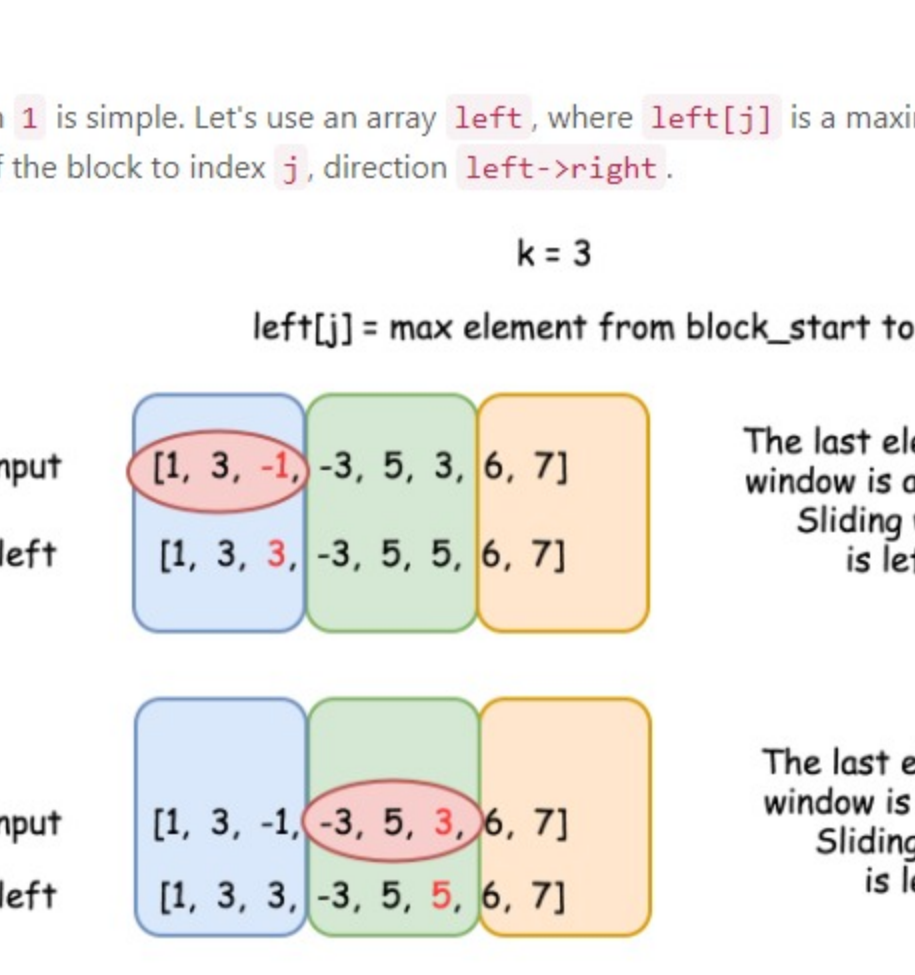
Approach 3: Dynamic programming

Intuition

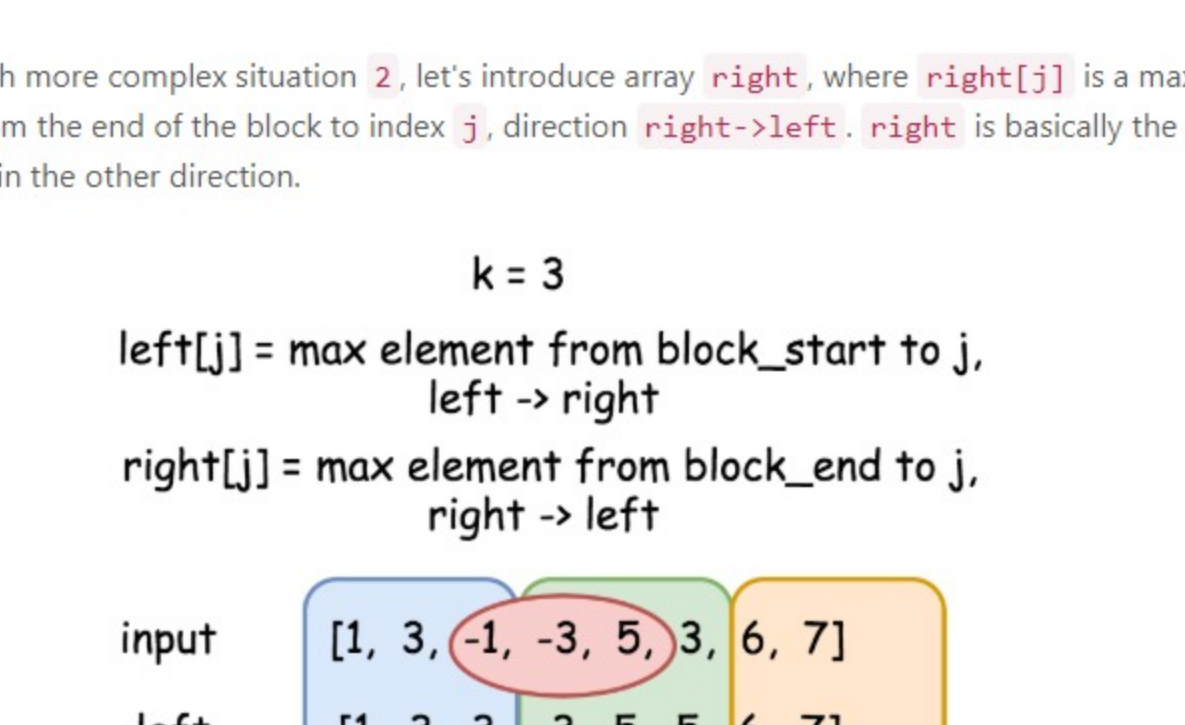
Here is another $\mathcal{O}(N)$ solution. The good thing about this solution is that you don't need any data structures but `array / list`.

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

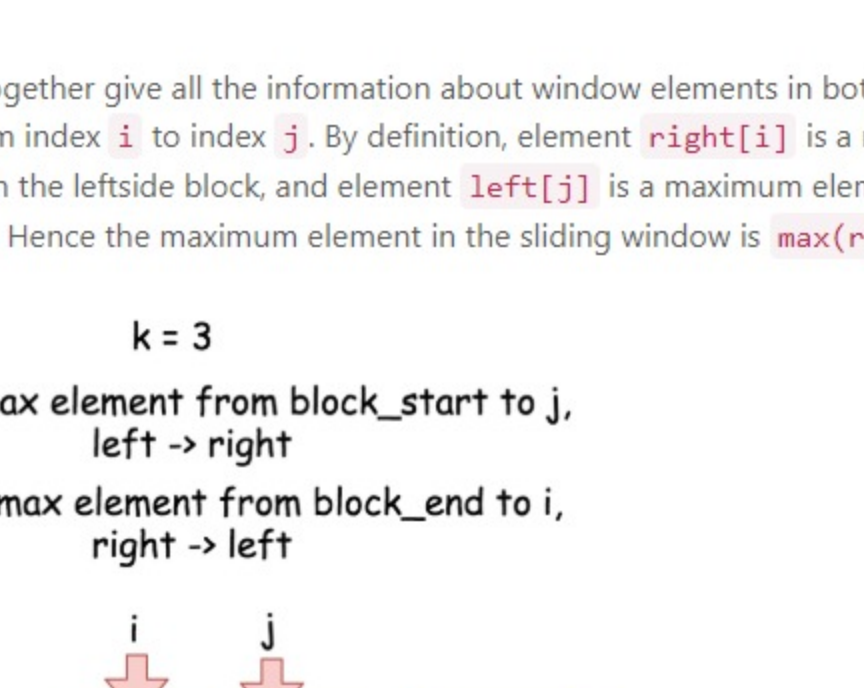
The current sliding window with the first element `i` and the last element `j` could be placed inside one block, or in two different blocks.



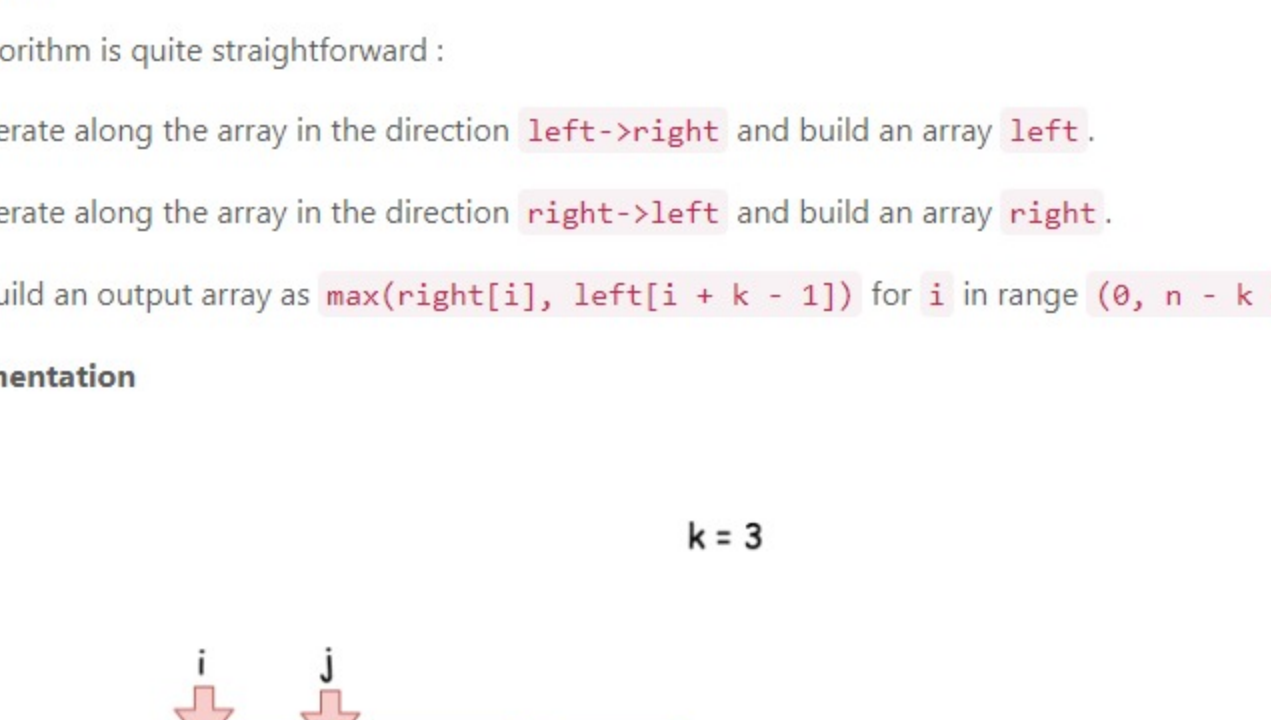
The situation `1` is simple. Let's use an array `left`, where `left[j]` is a maximum element from the beginning of the block to index `j`, direction `left->right`.



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])`.

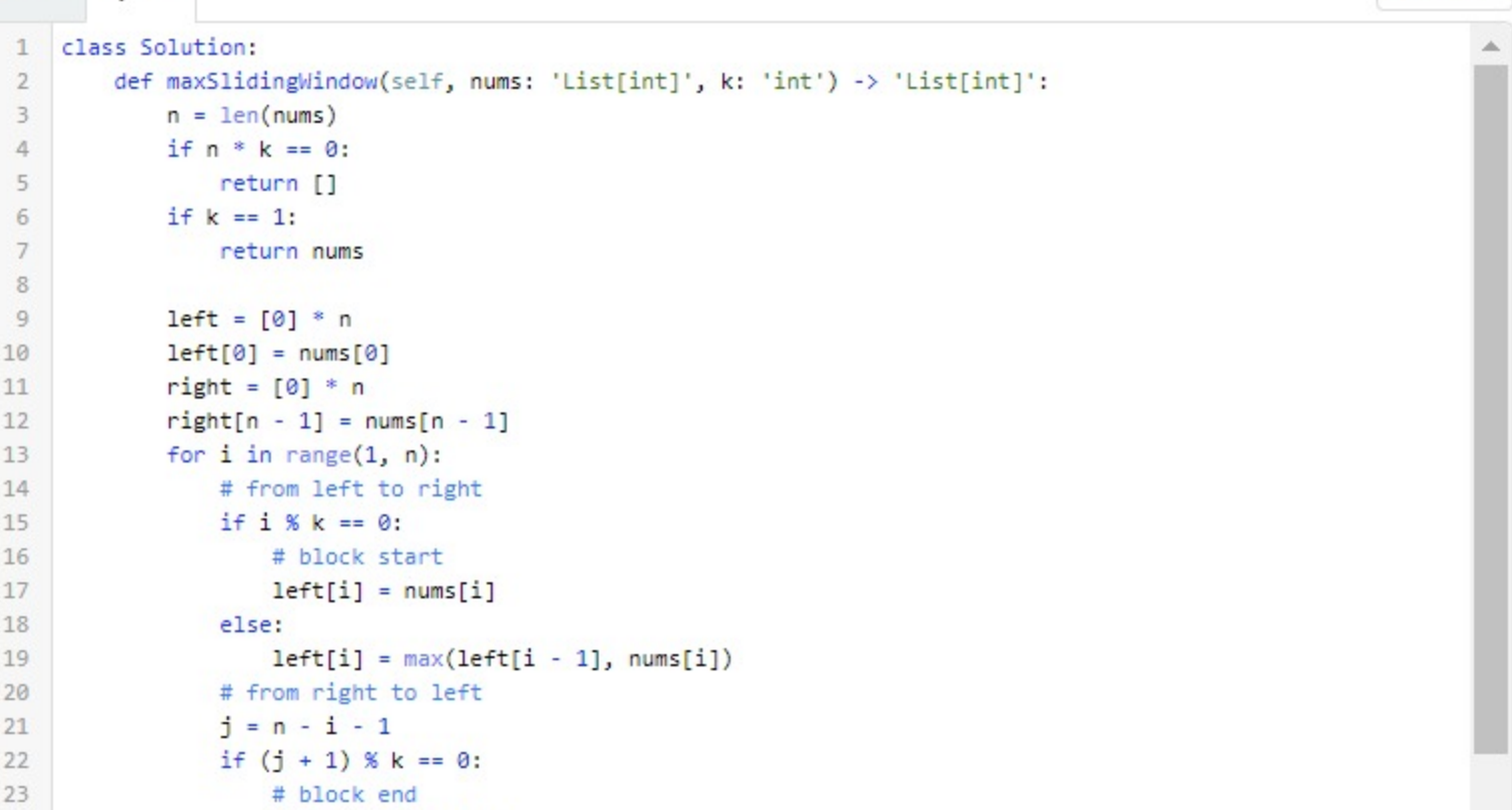


Algorithm

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

```python
class Solution:
    def maxSlidingWindow(self, nums: 'List[int]', k: 'int') -> 'List[int]':
        n = len(nums)
        if n * k == 0:
            return []
        if k == 1:
            return nums

        left = [0] * n
        left[0] = nums[0]
        right = [0] * n
        right[n - 1] = nums[n - 1]
        for i in range(1, n):
            # from left to right
            if i % k == 0:
                # block start
                left[i] = nums[i]
            else:
                left[i] = max(left[i - 1], nums[i])
            # from right to left
            j = n - i - 1
            if (j + 1) % k == 0:
                # block end
                right[j] = nums[j]
            else:
                right[j] = max(right[j + 1], nums[j])
```

Complexity Analysis

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

Rate this article: