<https://leetcode.com/problems/sliding-window-maximum/>

You are given an array of integers 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*.

**Example 1:**

Input: nums = [1,3,-1,-3,5,3,6,7], 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

**Example 2:**

Input: nums = [1], k = 1

Output: [1]

**Constraints:**

1 <= nums.length <= 10^5

-10^4 <= nums[i] <= 10^4

1 <= k <= nums.length

**Attempt 1: 2022-09-10**

**Solution 1 (5min, but Time Complexity: O(nlogk) consider as Time Limit Exceeded)**

class Solution {

    public int[] maxSlidingWindow(int[] nums, int k) {

        int len = nums.length;

        int[] result = new int[len - k + 1];

        PriorityQueue<Integer> maxPQ = new PriorityQueue<Integer>((a, b) -> b.compareTo(a));

        for(int i = 0; i < len; i++) {

            maxPQ.add(nums[i]);

            if(i >= k - 1) {

                result[i - k + 1] = maxPQ.peek();

                maxPQ.remove(nums[i - k + 1]);

            }

        }

        return result;

    }

}

Space Complexity: O(n)

Time Complexity: O(nlogk)

Not a linear solution, instead, it is of O(nlogk) complexity, since add, pop and  remove operation

of PriorityQueue cost O(logk) time. What we need to do is just maintain a heap, that heap top gets

the maximal value of the k elements.

**Solution 2 (360min, too long since not familiar with Monotonic Deque and have difficulty to come up with store index only)**

class Solution {

    public int[] maxSlidingWindow(int[] nums, int k) {

        int len = nums.length;

        // Use deque to store index (store index also store value by the way)

        // Since deque can add or remove element from either end, we can

        // leverage this attribute to simulate the natural scanning order(from

        // left to right) to add or remove nums' indexes from left to right,

        // which means always try to remove old indexes from deque's left end(

        // front end) by using removeFirst() / peekFirst() methods, and always

        // try to add new indexes on deque's right end(rear end) by using

        // addLast() / peekLast() methods.

        // Java Deque:

        // https://jenkov.com/tutorials/java-collections/deque.html

        Deque<Integer> deque = new LinkedList<Integer>();

        int[] result = new int[len - k + 1];

        for(int i = 0; i < len; i++) {

            // Remove index(represent corresponding number) out of range k

            // from deque's left end(front end)

            if(!deque.isEmpty() && deque.peekFirst() == i - k) {

                deque.removeFirst();

            }

            // Add new index(represent corresponding number) onto deque's

            // right end(rear end), but since it requires O(n) time complexity,

            // we could not implement additional sort algorithem or use auto

            // sort data structure like Priority Queue which is O(nlogn), finally

            // comes to Montonic Deque which guarantee O(n)

            // Before add a new index onto deque's rear end, we have to compare new

            // element's value(nums[i]) against all previous rear end elements'

            // values(nums[deque.peekLast()]), if previous rear end elements'

            // values less than new element value, we have to looply remove them

            // till find a larger or equal element to maintain a strictly decreasing

            // order of elements' values on monotonic deque from left to right(front

            // to rear)

            // In short, the elements(indexes represented) stored on deque must

            // monotonically decrease from left to right, e.g 1st > 2nd > 3rd...

            // and we can easily find the maximum element value at the left end(front

            // end) of deque by using nums[deque.peekFirst()]

            while(!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {

                deque.removeLast();

            }

            deque.addLast(i);

            if(i >= k - 1) {

                result[i - k + 1] = nums[deque.peekFirst()];

            }

        }

        return result;

    }

}

Space Complexity: O(n)

Time Complexity: O(n)

**Refer to**

**[Python] Decreasing deque, short, explained**

<https://leetcode.com/problems/sliding-window-maximum/solutions/951683/python-decreasing-deque-short-explained/>

There are a big variety of different algorithms for this problem. The most difficult, but most efficient uses idea of decreasing deque: on each moment of time we will keep only decreasing numbers in it. Let us consider the following example: nums = [1,3,-1,-3,5,3,6,7], k = 3. Let us process numbers one by one: (I will print numbers, however we will keep indexes in our stack):

We put 1 into emtpy deque: [1].

New element is bigger, than previous, so we remove previous element and put new one: [3].

Next element is smaller than previous, put it to the end of deque: [3, -1].

Similar to previous step: [3, -1, -3].

Now, let us look at the first element 3, it has index 1 in our data, what does it mean? It was to far ago, and we need to delete it: so we popleft it. So, now we have [-1, -3]. Then we check that new element is bigger than the top of our deque, so we remove two elements and have [5] in the end.

New element is smaller than previous, just add it to the end: [5, 3].

New element is bigger, remove elements from end, until we can put it: [6].

New element is bigger, remove elements from end, until we can put it: [7].

So, once again we have the following rules:

Elements in deque are always in decreasing order.

They are always elements from last sliding window of k elements.

It follows from here, that biggest element in current sliding window will be the 0-th element in it.

**Complexity**: time complexity is O(n), because we iterate over our elements and for each element it can be put inside and outside of our deque only once. Space complexity is O(k), the maximum size of our deque.

class Solution:

def maxSlidingWindow(self, nums, k):

deq, n, ans = deque([0]), len(nums), []

for i in range (n):

while deq and deq[0] <= i - k:

deq.popleft()

while deq and nums[i] >= nums[deq[-1]] :

deq.pop()

deq.append(i)

ans.append(nums[deq[0]])

return ans[k-1:]

**Refer to**

[L1425.Constrained Subsequence Sum (Ref.L239,L739,L53)](note://WEBa39329fbc7421070539b51823f6f484d)