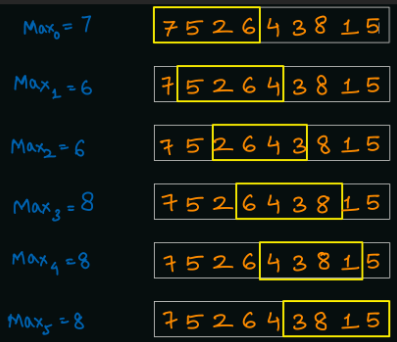
**1. Problem Discussion:**

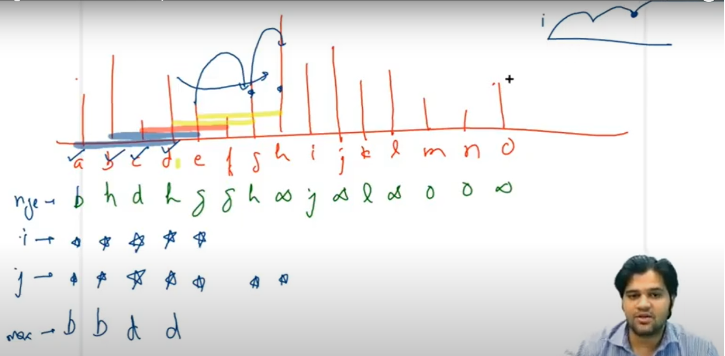
You are given an array arr of n numbers. You are also given a number k, representing the size of the window. You are required to find and print the maximum element in every window of size k. First Sliding window is a subarray of size k, starting from 0th index ending at index k-1, i.e. subarray [0, k-1]. First Sliding window is a subarray of size k, starting from 0th index ending at index k-1, i.e. subarray [0, k-1]. On each "slide", this window moves right by one position, i.e. the new window is now a subarray of size k having indices [1, k], and so on. Example: Consider an array = {7, 5, 2, 6, 4, 3, 8, 1, 5} and window size k = 4. Then the maximum of each window will be: Ans = [7, 6, 6, 8, 8, 8].

****

1• I recommend you to watch the question video, where the problem statement is explained in detail. 2•

**2. Approach :**

Brute Force - O(n \*k) Time, O(1) Space Naive approach can be to loop through all the windows possible. Then run a loop to find the maximum of the current window. Since, there can be at maximum (n-k+1) windows of size k in array of size n, thus time complexity will be O((n - k + 1) \* k) = O(n \* k). We will not take any extra space to find the maximum, as we just need to print the maximums, hence space complexity will be O(1). Note: There can be many different solutions for this problem, we can use Binary Heap, Self-Balancing Binary Search Tree (Ordered-Set), or it can also be solved using Deque Data Structure. But, we will be discussing a stack based solution in this article. Optimal Solution - O(n) Time & O(n) Space We can optimize the brute force solution by considering the fact that if we consider a given element in a window as potential maximum, then we can skip all the elements smaller than this element from the window. So, if we need to skip all the elements smaller than the current element in the window, then we will have to jump to an element which is greater than the current element and occurs after the element in the array. Hence, we first need to find the NEXT GREATER ELEMENT for all the elements of the array, so that we can jump and skip smaller elements. And I must expect from you that you know how to find of all the array elements. Hence, we will first build an array of size equal to the number of elements in array (n), and store the index of Next Greater Element (in the right direction) of all the elements. NGE to the rightIf there is no Next Greater in right for an element, then we will store n (size of array) as the NGE for that element. Now that we have the NGE array filled with indices, we will build the algorithm to find the maximum of each window:Consider the NGE of the current element at index i as NGE[i]. There may arise two cases: NGE[i] is within the current window's last index NGE[i] exceeds the current window's last index Q) So, how will we find the maximum of the current window?We will start from the first index of the current window. We will keep on jumping from i to NGE[i] until the index is within the current window. Finally, we will stop when NGE[i] is out of the current window's range, and the answer(maximum) of the current window will be the element at index i itself. Q) But, how can we say that the element at index i is the maximum of the current window?, It is because, we started from the first index of the current window, and kept on jumping to the next GREATER, upto the index for which next GREATER is out of the current window. Since we jumped from a smaller element to a greater element and stopped when there is no more greater element in the current window, we will eventually stop on the maximum element of the current window. Q) Why did we fill Next Greater as n for the elements who did not have any NGE in right?, R) Since, we need to stop at maximum element (index i) for every window, if we had not filled NGE of such elements as n, then we would have never stopped at maximum element as we would have kept on jumping to NGE of current elements which do not exist in this case. Making NGE as n will allow us to stop at this index i, as NGE[i] = n will be out of range for all windows. Q) I agree that finding the NGE of the entire array is O(n), but will starting from the first index of every window make the algorithm remain O(n) ? Answer is no, consider this corner case: If the array is sorted in increasing order, like: [1, 2, 3, 4, 5, 6, 7, 8, 9]. Then for every window, we will jump for a maximum of k times, because every element has NGE[i] = i + 1. Thus we will get the same O(n \* k) time complexity in the worst case (which is equivalent to the brute force). Q) How to modify the algorithm, so that we can get worst case time complexity as O(n) only? We need not start from the first index of the current window every time. Let's suppose we ended at index i for the previous window, then for the current window we can consider starting from this index i only. By doing so, we can skip all the elements having index less than i and are smaller than arr[i]. Hence, we are not only able to skip smaller elements in right, but also skip the elements smaller than arr[i] from the previous window. Q) So, now we have got the algorithm! But will the index i of the maximum of the previous window always be in the current window also? R) Not necessarily! (Please be patient, that's why it is a hard problem). What if the index i is less than the first index of the current window? In this case, we are not starting from an element of the current window. Hence, before starting to find the maximum of the current window, we will make the pointer as the first index only if it is less than this index. Don't panic! I think now the algorithm is complete and it can handle all the corner cases. Please refer to the solution video if you find difficulty in understanding the algorithm completely. The algorithm is explained by using the diagrams and graph plotting.

****

Pseudo Code:

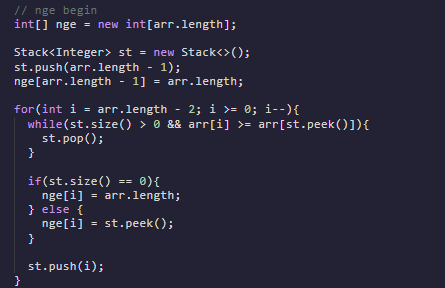
Create a NGE array of size n, with all values initialized with n. Fill this NGE array using the concept of NGE in right. (Please refer to this problem Initialize the pointer i with 0, as the first index of the current window Run a loop w, for iterating through all the windows of size k, from 0 to arr.length - k. Note: We are running this loop until arr.length - k only because if we consider subarrays with index > arr.length - k, then their size will be less than k, but we need a window of size exactly equal to k. If pointer i is less than w, (do not lie inside the current window), then make it point to w (first index of window), i.e. i = w. Update i as NGE[i] and stop when NGE[i] points to the index out of range of the current window (i.e. stop when NGE[i] >= w + k, (last index of current window)). Element arr[i] is the maximum of the current window, hence print the element. Seems simple, right? Do give it a try before reading the code.

**3. Code**

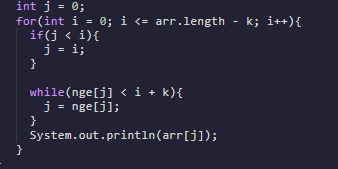
import java.io.\*; import java.util.\*; public class Main { public static void main(String[] args) throws Exception { BufferedReader br = new BufferedReader(new InputStreamReader(System.in)); int n = Integer.parseInt(br.readLine()); int[] arr = new int[n]; for (int i = 0; i < n; i++) { arr[i] = Integer.parseInt(br.readLine()); } int k = Integer.parseInt(br.readLine()); // nge begin int[] nge = new int[arr.length]; Stack< Integer> st = new Stack<>(); st.push(arr.length - 1); nge[arr.length - 1] = arr.length; for (int i = arr.length - 2; i >= 0; i--) { while (st.size() > 0 && arr[i] >= arr[st.peek()]) { st.pop(); } if (st.size() == 0) { nge[i] = arr.length; } else { nge[i] = st.peek(); } st.push(i); } // nge end int i = 0; for (int w = 0; w <= arr.length - k; w++) { if (i < w) { i = w; } while (nge[i] < w + k) { i = nge[i]; } System.out.println(arr[i]); } } }

**4. Code Explanation**

In the code below we are finding out the next greater element for every element in the array.

****

Now in every window we are finding out the maximum element with the help of nge array.

****

**5. Analysis:**

Time Complexity :

First, we are generating the NGE array of size n, which takes O(n) time and O(n) space for the NGE array and stack as auxiliary space. Since, we need not start from the first index of every window, thus skipping all smaller elements in the previous window, and we are jumping from one element to the next greater and skipping all the smaller elements in the current window, the two pointers w and i only move for at maximum n steps independently. Hence, the time complexity will be O(n + 2 \* n) = O(3 \* n) = O(n) only.

Space Complexity:

We require the NGE array, which accounts for the extra O(n). We have not studied DEQUE data structure, but this problem can also be solved using it, in similar O(n) time complexity but O(k) space complexity. Please try to come up with this solution as well, after you learn about deque. O(n) space and O(k) space solution does not bring any difference in the worst case, as k can be as maximum as n. Though, in best case, when k = 1, the deque solution will be equivalent to constant space. I hope you enjoyed solving the problem with me. We will come with another problem "Infix Evaluation" for you to solve, until then Happy Coding!