<https://leetcode.com/problems/kth-largest-element-in-a-stream/description/>

Design a class to find the kth largest element in a stream. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Implement KthLargest class:

KthLargest(int k, int[] nums) Initializes the object with the integer k and the stream of integers nums.

int add(int val) Appends the integer val to the stream and returns the element representing the kth largest element in the stream.

**Example 1:**

Input

["KthLargest", "add", "add", "add", "add", "add"]

[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]

Output

[null, 4, 5, 5, 8, 8]

Explanation

KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);

kthLargest.add(3); // return 4

kthLargest.add(5); // return 5

kthLargest.add(10); // return 5

kthLargest.add(9); // return 8

kthLargest.add(4); // return 8

**Constraints:**

1 <= k <= 104

0 <= nums.length <= 104

-104 <= nums[i] <= 104

-104 <= val <= 104

At most 104 calls will be made to add.

It is guaranteed that there will be at least k elements in the array when you search for the kth element.

**Attempt 1: 2023-11-06**

**Solution 1: Priority Queue (30 min)**

class KthLargest {

PriorityQueue<Integer> minPQ;

int size;

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

this.minPQ = new PriorityQueue<>();

this.size = k;

for(int num : nums) {

add(num);

}

}

public int add(int val) {

if(minPQ.size() < size) {

minPQ.offer(val);

} else if(minPQ.peek() < val) {

minPQ.poll();

minPQ.offer(val);

}

return minPQ.peek();

}

}

/\*\*

\* Your KthLargest object will be instantiated and called as such:

\* KthLargest obj = new KthLargest(k, nums);

\* int param\_1 = obj.add(val);

\*/

Time complexity: O((n+m)∗log(k)) where mis the number of calls to the add method

Space complexity: O(k)

**Refer to**

<https://leetcode.com/problems/kth-largest-element-in-a-stream/solutions/3553822/java-priorityqueue-7-lines-clean-code/>

# **Intuition**

Use a min heap and maintain its size at k elements. By doing this we ensure that the root node of the heap is always the kth largest element.

class KthLargest {

private PriorityQueue<Integer> heap = new PriorityQueue<>();

private int k;

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

this.k = k;

for (var n : nums) add(n);

}

public int add(int val) {

heap.offer(val);

if (heap.size() > k) heap.poll();

return heap.peek();

}

}

# **Complexity**

Time complexity: O((n+m)∗log(k)) where m is the number of calls to the add method

Space complexity: O(k)