Top K Frequent Elements

Given an integer array nums and an integer k, return the k most frequent elements.
You may return the answer in any order.

Ex1 :

Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

Ex2 :

Input: nums = [1], k = 1
Output: [1]

Constraints:

1 <= nums.length <= 105
k is in the range [1, the number of unique elements in the array].
It is guaranteed that the answer is unique.

Follow up: Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

Ex1 :

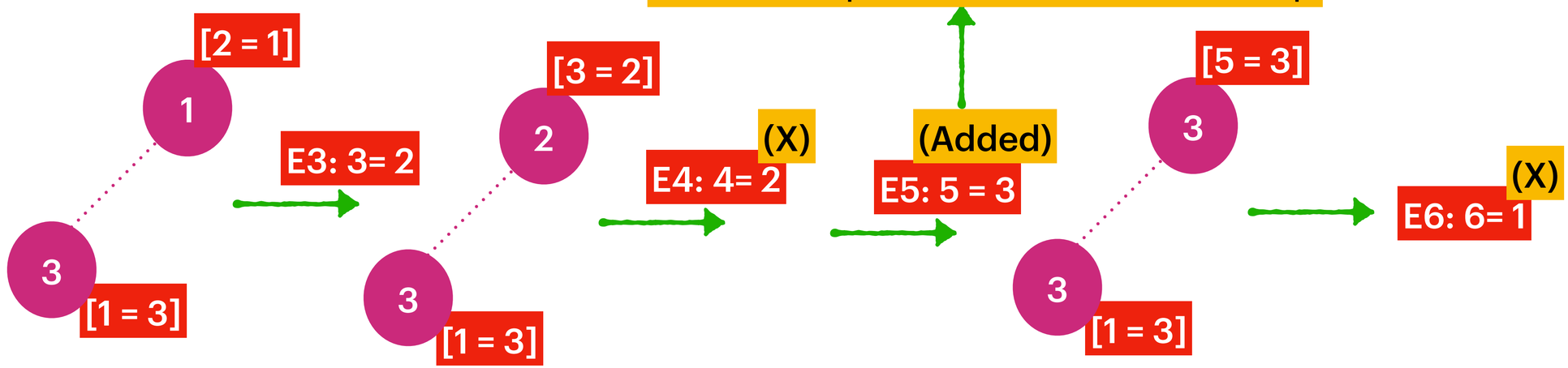
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

Ex2 :

Input: nums = [1,1,1,2,3,3,4,5,6,4,5,5], k = 2
Output: [1,5]

[1,1,1,2,3,3,4,5,6,4,5,5], k = 2

counterMap :
E1: 1 = 3
E2: 2 = 1
E3: 3= 2
E4: 4 = 2
E5: 5 = 3
E6: 6 = 1



Heap has [5=3] , [1=3] these elements return keys = {5,1}

Constructing Map + Add the elements int to minHeap [Sort by Counter]
O(n) + O(nlogk) = O(nlogk)

Kth Largest Element in a Stream

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:

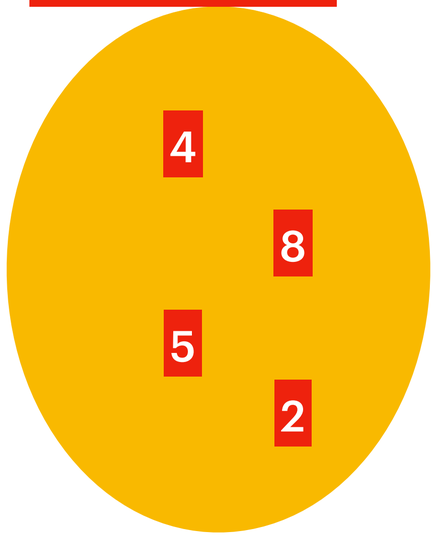
1 <= k <= 104
0 <= nums.length <= 104
-104 <= nums[i] <= 104
-104 <= val <= 104
At most 104 calls will be made to add.
It is guaranteed that there will be at least k elements in the array
when you search for the kth element.

Kth Largest Element in a Stream

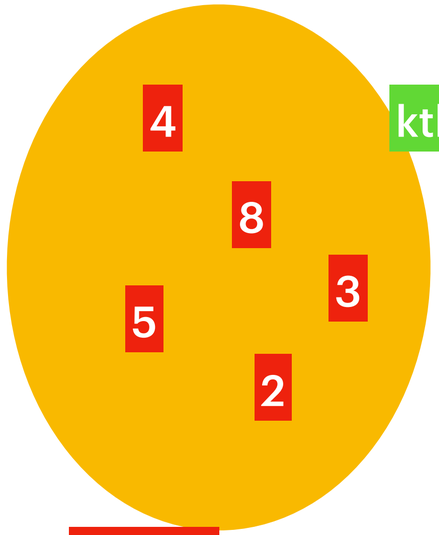
[[k=3, construct[4, 5, 8, 2]], add[3], add[5], add[10], add[9], add[4]]

Output : [null, 4, 5, 5, 8, 8]

Initial Stream :

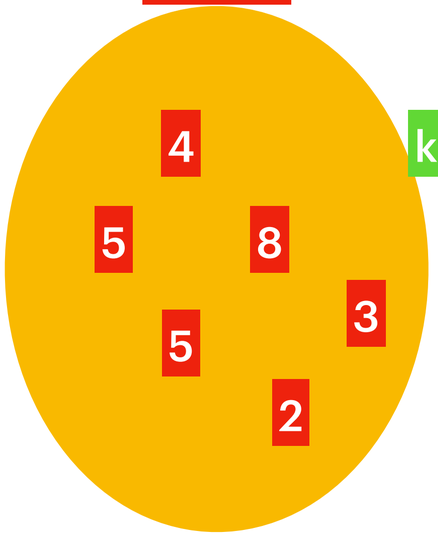


add[3]



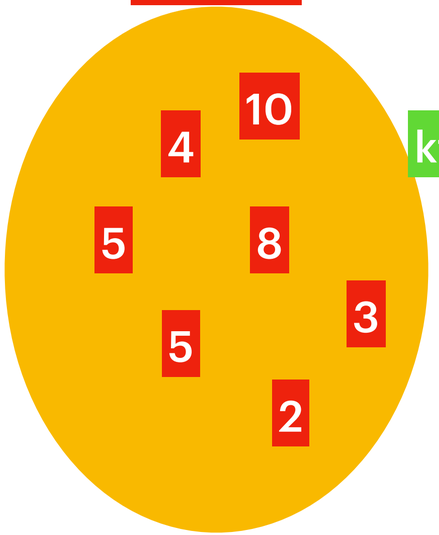
kth Largest 4

add[5]



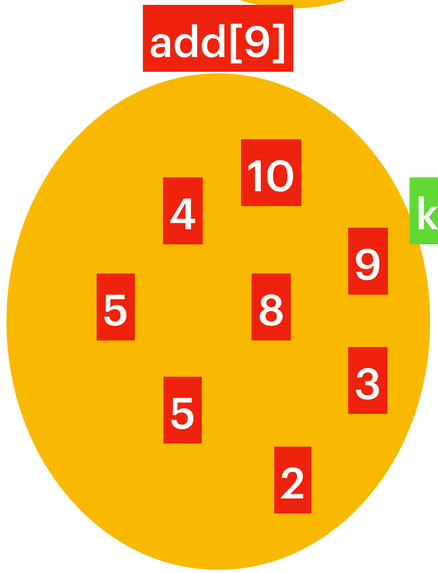
kth Largest 5

add[10]



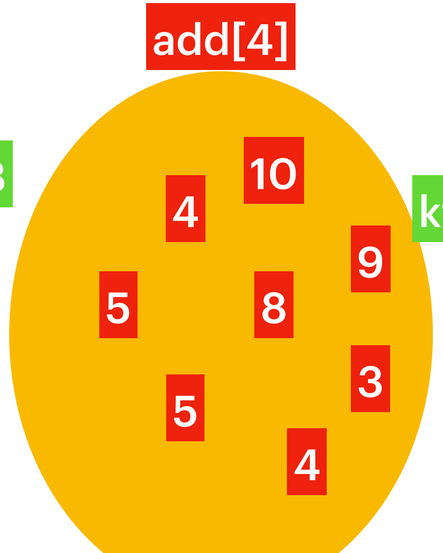
kth Largest 5

add[9]



kth Largest 8

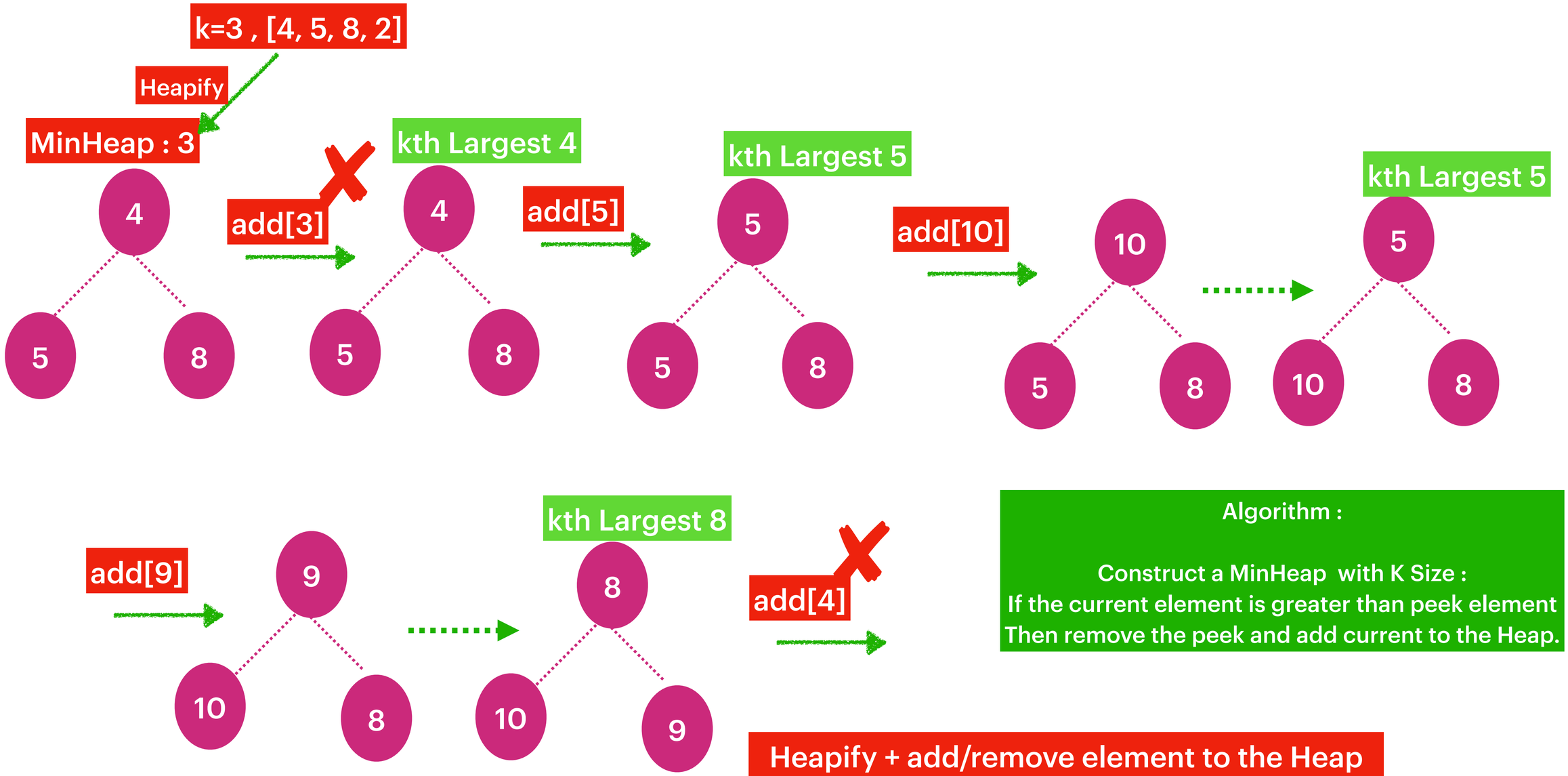
add[4]



kth Largest 8

[4,5,5,8,8]

[[k=3, construct[4, 5, 8, 2]], add[3], add[5], add[10], add[9], add[4]]



Algorithm :

Construct a MinHeap with K Size :
If the current element is greater than peek element
Then remove the peek and add current to the Heap.

Heapify + add/remove element to the Heap
Time Complexity : O(nlogk) + O(nlogk) =nlogk
Space Complexity : O(k)

Meeting Rooms II

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return the minimum number of conference rooms required.

Example 1:

Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2

Example 2:

Input: intervals = [[7,10],[2,4]]
Output: 1

Constraints:

1 <= intervals.length <= 104
0 <= starti < end

Example 1:
Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2

By the time meeting3 starts
Meeting2 would over in Room2 so that we can use Room2

Schedule in Room1	Schedule in Room2	Schedule in Room2
Meeting 1 [0,30]	Meeting 2 [5,10]	Meeting 3 [15,20]

Output : Total Minimal No.of Rooms required = 2

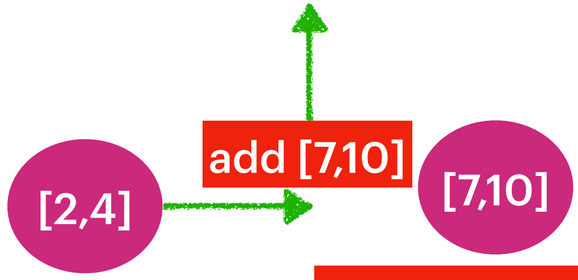
Example 2:

Input: intervals = [[7,10],[2,4]]
Output: 1

Input: intervals = [[2,4],[7,10]]

Sort the time intervals based on start time

Current meeting startTime >= peek().endTime
So we can use same room.
Remove the peek & add the current one.



HeapSize : 1 i.e Output : 1

[[7,10],[45,60], [2,4],[3,7],[41,44],[9,15],[15,30]]

Sort the intervals based startTime

[[2,4],[3,7],[7,10],[9,15],[15,30] ,[41,44],[45,60]]

MinHeap based on endTime

Algorithm :
Sort the intervals based on startTime.

Construct minHeap based on endTime,
While adding interval to the heap, if the current interval startTime >= peek() endTime then remove the peek() & add the currentInterval.

Output would be the heapSize:

Time Complexity : sorting + adding/removing element to Heap
O(nlogn) + O(nlogk) where k <= n = O(nlogn)