

Kth Largest Element

Find the Kth largest without full sorting — Min-Heap of size K ■

● MEDIUM

■ Hook

'Find the Kth largest element' is the most classic heap interview question. You could sort and index: $O(n \log n)$. But there's a smarter way using a **min-heap of exactly K elements** that runs in $O(n \log K)$ — significantly faster when K is small. This pattern appears in dozens of variants: Kth smallest, K closest, top-K frequency...

■ Core Intuition — The K-Size Min-Heap Trick

Keep a min-heap of size exactly K. It holds the K largest elements seen so far. The **root (minimum of the heap) = Kth largest overall**. Why? Because the K largest elements are in the heap, and the smallest of those K is the Kth largest.

■ The Key Insight

A min-heap of size K acts as a 'bouncer': only the top-K elements stay inside. Every new element competes with the weakest current member (the root). If it's stronger (larger), it kicks out the root and joins. Otherwise, it's rejected. After processing all elements, the root is exactly the Kth largest.

■ Visual — [3,2,1,5,6,4], K=2

```
nums=[3,2,1,5,6,4]  K=2  → find 2nd largest
Process each number, maintain min-heap of size 2:
Add 3:  heap=[3]          size=1 < K, just add
Add 2:  heap=[2,3]        size=2 = K, full
Add 1:  1 < heap.peek()=2 → REJECT (1 is too small)
Add 5:  5 > heap.peek()=2 → poll 2, add 5 → heap=[3,5]
Add 6:  6 > heap.peek()=3 → poll 3, add 6 → heap=[5,6]
Add 4:  4 < heap.peek()=5 → REJECT
FINAL heap=[5,6] → heap.peek() = 5
★ ANSWER = 5 (5 is the 2nd largest in [3,2,1,5,6,4])
Heap contents represent: {5, 6} = the 2 largest elements
Root (min of these K) = 5 = Kth largest ✓
```

■ Algorithm Steps

- 1 Create min-heap**
PriorityQueue minHeap = new PriorityQueue<>();
- 2 Process each number**
For every number in nums: add to heap.
- 3 Maintain size K**
After adding, if heap.size() > K: poll() to remove the minimum.
- 4 Return root**
heap.peek() is the Kth largest — it's the minimum of the K largest elements.

■ Code Implementation (Java)

```
import java.util.PriorityQueue;
public class KthLargest {
    // Approach 1: Min-Heap of size K  O(n log K)
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();
        for (int num : nums) {
            minHeap.add(num);
            if (minHeap.size() > k) {
                minHeap.poll(); // remove smallest — keep only K largest
            }
        }
        return minHeap.peek(); // min of K largest = Kth largest
    }
    // Approach 2: QuickSelect  O(n) average (bonus — no heap)
    public int findKthLargestQS(int[] nums, int k) {
        return quickSelect(nums, 0, nums.length-1, nums.length-k);
    }
    private int quickSelect(int[] nums, int lo, int hi, int k) {
        int pivot = partition(nums, lo, hi);
        if (pivot == k) return nums[pivot];
        return pivot < k ? quickSelect(nums, pivot+1, hi, k)
            : quickSelect(nums, lo, pivot-1, k);
    }
    private int partition(int[] nums, int lo, int hi) {
        int pivot = nums[hi], i = lo;
        for (int j = lo; j < hi; j++)
            if (nums[j] <= pivot) { int t=nums[i]; nums[i++]=nums[j]; nums[j]=t; }
        int t=nums[i]; nums[i]=nums[hi]; nums[hi]=t;
        return i;
    }
}
```

■ Time & Space Complexity

■ TIME COMPLEXITY

$O(n \log K)$

■ SPACE COMPLEXITY

$O(K)$

Each of n elements: one add $O(\log K)$ + possibly one poll $O(\log K) \rightarrow O(n \log K)$. Space: only K elements in heap. QuickSelect: $O(n)$ average, $O(n^2)$ worst — great for one-time queries.

■ Common Mistakes

■ Using a MAX-heap (wrong approach)

Using a max-heap of all n elements and polling K times: $O(n + K \log n)$. While correct, it uses $O(n)$ space and is slower. The min-heap of size K is the optimal $O(n \log K) / O(K)$ approach.

■ Kth largest vs Kth smallest confusion

Kth LARGEST $\rightarrow$ min-heap of size $K \rightarrow$ return `peek()`. Kth SMALLEST $\rightarrow$ max-heap of size $K \rightarrow$ return `peek()`. The heap type is opposite to what you're looking for!

■ Practice Problems

#	Problem	Diff	Pattern
1	Kth Largest Element in an Array (LC 215)	Medium	Min-Heap size K
2	Kth Smallest Element in a Sorted Matrix (LC 378)	Medium	Min-Heap
3	Kth Largest Element in a Stream (LC 703)	Easy	Min-Heap size K
4	K Closest Points to Origin (LC 973)	Medium	Max-Heap size K
5	Top K Frequent Elements (LC 347)	Medium	Min-Heap + HashMap

■ Quick Revision

- ✓ Kth Largest $\rightarrow$ min-heap of size $K \rightarrow$ root = answer
- ✓ Add each element; if size $> K$, `poll()` (removes minimum)
- ✓ After all elements: `heap.peek()` = Kth largest
- ✓ Time: $O(n \log K)$ | Space: $O(K)$
- ✓ Kth Smallest $\rightarrow$ max-heap of size K (opposite!)
- ✓ QuickSelect = $O(n)$ average — best for single queries

■ MENTAL MODEL — Remember This Forever

Think of it as an exclusive VIP club with exactly K spots. Each new person at the door is compared to the least impressive current member (the root of the min-heap). If they're more impressive, they kick out the weakest and join. Otherwise, they're turned away. After everyone's been evaluated, the weakest person still in the club is the K th most impressive overall.