LeetCode 215 — Kth Largest Element in an Array

1. Problem Title & Link

- **Title:** LeetCode 215 — Kth Largest Element in an Array
- **Link:** https://leetcode.com/problems/kth-largest-element-in-an-array/

2. Problem Statement (Short Summary)

Given an integer array nums and an integer k, return the **kth largest element** in the array (the kth largest **distinct index** when array is sorted descending). You must design an algorithm faster than O(n log n) if possible.

Example: nums = [3,2,1,5,6,4], k = 2 → return 5.

3. Examples (Input → Output)

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

Output: 5

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

Output: 4

4. Constraints

- 1 <= k <= nums.length <= 10^5
- -10^9 <= nums[i] <= 10^9
- Time: target average O(n); allowed O(n log n) (sorting) for simplicity; better is Quickselect or heap.

5. Core Concept (Pattern / Topic)

Selection Algorithms — Quickselect (average O(n))

Also: **Min-heap** (O(n log k)) and sorting (O(n log n)). Quickselect (randomized) is the interview-favourite for expected linear time.

6. Thought Process (Step-by-Step Explanation)

Three main approaches:

A. Quickselect (recommended)

- Convert kth largest to index target = n - k (kth largest = (n-k)th smallest).
- Use Quickselect (partition like quicksort — Lomuto or Hoare).
- Recursively partition until pivot index == target.
- Average time O(n), worst-case $O(n^2)$ unless pivot randomized.

B. Min-Heap (simple & robust)

- Maintain a min-heap of size k.

- Push elements; if heap size > k pop smallest.
- Top of heap at the end is kth largest.
- Time O(n log k), space O(k).

C. Sort (straightforward)

- sort nums descending and return nums[k-1] → O(n log n). Use if constraints allow.

We'll give Quickselect code (Python + Java) and describe the heap alternative.

7. Visual / Intuition Diagram

Quickselect partitions array so elements ≤ pivot go left and > pivot go right. If pivot ends at index p:

- if p == target → pivot is answer
- if p < target → search right subarray
- if p > target → search left subarray

This narrows problem quickly — only one side explored each step.

8. Pseudocode (Quickselect)

```
target = n - k

def quickselect(left, right):
    pivot_index = random between left and right
    pivot_index = partition(left, right, pivot_index)
    if pivot_index == target: return nums[target]
    elif pivot_index < target: return quickselect(pivot_index+1, right)
    else: return quickselect(left, pivot_index-1)
Partition (Lomuto):
  • Move pivot to end
  • i = left
  • for j in left..right-1:
    if nums[j] <= pivot: swap(nums[i], nums[j]); i += 1
  • swap(nums[i], nums[right]) → pivot final index i
```

9. Code Implementation

✅ **Python (randomized Quickselect)**

```
import random
from typing import List

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # kth largest -> index target = len(nums) - k (0-based)
        target = len(nums) - k
```

```
        def partition(l, r, pivot_index):
            pivot = nums[pivot_index]
            # move pivot to end
            nums[pivot_index], nums[r] = nums[r], nums[pivot_index]
            store = l
            for j in range(l, r):
                if nums[j] <= pivot:
                    nums[store], nums[j] = nums[j], nums[store]
                    store += 1
            # move pivot to its final place
            nums[store], nums[r] = nums[r], nums[store]
            return store

        def quickselect(l, r):
            if l == r:
                return nums[l]
            # random pivot to avoid worst-case
            pivot_index = random.randint(l, r)
            pivot_index = partition(l, r, pivot_index)
            if pivot_index == target:
                return nums[pivot_index]
            elif pivot_index < target:
                return quickselect(pivot_index + 1, r)
            else:
                return quickselect(l, pivot_index - 1)

        return quickselect(0, len(nums) - 1)
```

✅ **Java (randomized Quickselect)**

```
import java.util.Random;

class Solution {
    private Random rand = new Random();

    public int findKthLargest(int[] nums, int k) {
        int n = nums.length;
        int target = n - k;
        return quickselect(nums, 0, n - 1, target);
    }

    private int quickselect(int[] a, int l, int r, int target) {
```

```
        if (l == r) return a[l];
        int pivotIndex = l + rand.nextInt(r - l + 1);
        pivotIndex = partition(a, l, r, pivotIndex);
        if (pivotIndex == target) return a[pivotIndex];
        else if (pivotIndex < target) return quickselect(a, pivotIndex + 1, r,
target);
        else return quickselect(a, l, pivotIndex - 1, target);
    }

    private int partition(int[] a, int l, int r, int pivotIndex) {
        int pivot = a[pivotIndex];
        swap(a, pivotIndex, r);
        int store = l;
        for (int j = l; j < r; j++) {
            if (a[j] <= pivot) {
                swap(a, store, j);
                store++;
            }
        }
        swap(a, store, r);
        return store;
    }

    private void swap(int[] a, int i, int j) {
        int t = a[i]; a[i] = a[j]; a[j] = t;
    }
}
```

10. Time & Space Complexity

Approach	Time (avg/worst)	Space
Quickselect (randomized)	**O(n)** average, **O(n²)** worst	O(1) extra (recursion stack O(log n) average)
Min-Heap (size k)	O(n log k)	O(k)
Sorting	O(n log n)	O(1) or O(n) depending on sort

Quickselect gives best average time and O(1) space besides input.

11. Common Mistakes / Edge Cases

- Not converting kth largest → target index properly (target = n - k).
- Using deterministic pivot (last element) on already sorted arrays → worst-case $O(n^2)$. Use random pivot to avoid.
- Off-by-one errors when partition indices are wrong.

- Modifying input array unintentionally (acceptable here — Quickselect is in-place).
- Stack overflow for deeply unbalanced recursion (rare with randomized pivots).
- k out of range (problem constraints assure valid k).

Edge cases:

- nums of length 1, k=1 → return nums[0]
- negative numbers, duplicates (works fine)
- all elements equal

12. Detailed Dry Run (Step-by-Step Table)

Example: nums = [3,2,1,5,6,4], k = 2 → expected output 5.

Compute: n = 6, target = n - k = 4 (0-based index)

We'll show deterministic pivot = last (for clarity). (Real code uses random pivot.)

Initial array: [3,2,1,5,6,4], left=0, right=5

Partition 1 (pivot = 4 at index 5 using Lomuto):

- pivot = 4
- store = 0
- j=0: nums[0]=3 ≤ 4 → swap nums[0] with nums[0], store=1 → arr [3,2,1,5,6,4]
- j=1: nums[1]=2 ≤ 4 → swap nums[1] with nums[1], store=2 → arr [3,2,1,5,6,4]
- j=2: nums[2]=1 ≤ 4 → swap nums[2] with nums[2], store=3 → arr [3,2,1,5,6,4]
- j=3: nums[3]=5 > 4 → nothing
- j=4: nums[4]=6 > 4 → nothing
- After loop swap nums[store]=nums[3] and pivot nums[5]: swap 5 and 4
- Array becomes: [3,2,1,4,6,5]
- pivot_index = 3

Compare pivot_index (3) with target (4):

- 3 < 4 → search right subarray indices [4..5]

Partition 2 on subarray [4..5] = [6,5], pivot = 5 (index 5)

- pivot = 5
- store = 4
- j=4: nums[4]=6 > 5 → nothing
- swap nums[store]=nums[4] with pivot nums[5]: swap 6 and 5
- Array becomes: [3,2,1,4,5,6]
- pivot_index = 4

Now pivot_index == target (4) → return nums[4] == **5** ✔

So kth largest = **5**.

(Quickselect would choose random pivots but the logic is identical — we ended at the target after two partitions.)

13. Common Use Cases (Real-Life / Interview)

- Selecting top-k values (scores, revenues, etc.) efficiently
- Median/quantile computation (special case k = n/2)
- Streaming top-k via heap-based approach
- Interview pattern: selection & partitioning algorithms

14. Common Traps

- Confusing kth largest with kth distinct (this problem uses index-based, not distinct-values).
- Not handling duplicates properly when using partition logic (Lomuto handles duplicates with ≤ pivot grouping left).
- Forgetting to randomize pivot or handle pathological inputs if worst-case performance matters.

15. Builds To (Related Problems)

- **LC 215** variants: top-k frequent elements (LC 347) — uses selection or heap.
- Selection algorithms and median of medians for worst-case O(n).
- Streaming top-k: use heaps or reservoir sampling for streams.

16. Alternate Approaches + Comparison

Approach	When to use	Pros	Cons
Quickselect (randomized)	Large arrays, need average O(n)	Fast average, O(1) extra	Worst-case $O(n^2)$ (rare)
Min-Heap (size k)	k much smaller than n	Predictable O(n log k), easy	O(k) extra space
Sort	Simpler, small n	Simple to implement	O(n log n) slower when n large

17. Why This Solution Works (Short Intuition)

Partitioning arranges elements around a pivot so that the pivot's final index tells you how many elements are ≤ it. Recurse only into the side that contains the desired index — each partition reduces the search range, giving expected linear time.

18. Variations / Follow-Up Questions

- Kth **smallest** element (target = k-1)
- Kth largest distinct element (requires deduplication)
- Median-of-medians selection for deterministic O(n) worst-case
- Top-k frequent elements vs top-k values (different problem)

Min-Heap Approach (O(n log k))

Python Implementation (Min-Heap)

```
Python's heapq is min-heap by default.
import heapq

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        min_heap = []

        for num in nums:
            heapq.heappush(min_heap, num)
            if len(min_heap) > k:
                heapq.heappop(min_heap)

        return min_heap[0]
```

✔ **Time: O(n log k)**

✔ **Space: O(k)**

❤️ **Java Implementation (Min-Heap)**

Use PriorityQueue which is a min-heap by default.

```
import java.util.PriorityQueue;

class Solution {
    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
            }
        }

        return minHeap.peek();  // kth largest
    }
```

}

✔ **Time: O(n log k)**

✔ **Space: O(k)**