<https://leetcode.com/problems/kth-largest-element-in-an-array/>

Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

**Example 1:**

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

Output: 5

**Example 2:**

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

Output: 4

**Constraints:**

* 1 <= k <= nums.length <= 105
* -104 <= nums[i] <= 104

**Attempt 1: 2023-08-03**

**Solution 1: Sort (10min)**

class Solution {

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

Arrays.sort(nums);

return nums[nums.length - k];

}

}

Time complexity: O(n⋅log⁡n)

Sorting nums requires O(n⋅log⁡n) time.

Space Complexity: O(log⁡n)

The space complexity of the sorting algorithm depends on the implementation of each programming language:

- In Java, Arrays.sort() for primitives is implemented using a variant of the Quick Sort algorithm, which has a space complexity of O(log⁡n)

**Refer to**

<https://leetcode.com/problems/kth-largest-element-in-an-array/editorial/>

#### **Approach 1: Sort**

**Intuition**

Sort the array in descending order and then return the kth

element. Note that this is the "trivial" approach and if asked this question in an interview, you would be expected to come up with a better solution than this.

**Implementation**

Note: kis 1-indexed, not 0-indexed. As such, we need to return the element at index k - 1after sorting, not index k.

class Solution {

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

Arrays.sort(nums);

// Can't sort int[] in descending order in Java;

// Sort ascending and then return the kth element from the end

return nums[nums.length - k];

}

}

**Complexity Analysis**

Given n as the length of nums,

Time complexity: O(n⋅log⁡n)

Sorting nums requires O(n⋅log⁡n) time.

Space Complexity: O(log⁡n)

The space complexity of the sorting algorithm depends on the implementation of each programming language:

In Java, Arrays.sort() for primitives is implemented using a variant of the Quick Sort algorithm, which has a space complexity of O(log⁡n)

**Solution 2: Min-heap or Max-heap (10min)**

**Style 1: Use build in Java PriorityQueue**

Minimum PriorityQueue

class Solution {

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

// Java default PriorityQueue as minimum PriorityQueue

PriorityQueue<Integer> minPQ = new PriorityQueue<Integer>();

for(int num : nums) {

minPQ.offer(num);

if(minPQ.size() > k) {

// The poll() is to remove the head/root which is

// the minimum value at the peek of current minimum

// PriorityQueue, it takes O(logN) time

minPQ.poll();

}

}

return minPQ.peek();

}

}

=========================================================================

Maximum PriorityQueue

class Solution {

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

// Maximum PriorityQueue

// kth largest in nums means (nums.length - k)th smallest

// e.g [3,2,1,5,6,4], k = 2

// To find 2nd largest number equal to find 5th smallest number

// the 5th comes from (nums.length - k + 1) = 6 - 2 + 1 = 5

PriorityQueue<Integer> maxPQ = new PriorityQueue<Integer>((a, b) -> b - a);

for(int num : nums) {

maxPQ.offer(num);

if(maxPQ.size() > nums.length - k + 1) {

maxPQ.poll();

}

}

return maxPQ.peek();

}

}

Time complexity: O(n⋅log⁡k)

Given n as the length of nums. Operations on a heap cost logarithmic time relative to its size. Because our heap is limited to a size of k, operations cost at most O(log⁡k). We iterate over nums, performing one or two heap operations at each iteration.

We iterate n times, performing up to log⁡k work at each iteration, giving us a time complexity of O(n⋅log⁡k)

Because k≤n , this is an improvement on the previous approach.

Space complexity: O(k) The heap uses O(k) space.

**Style 2: Use customized PriorityQueue**

class Solution {

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

int length = nums.length;

MaxPQ maxPQ = new MaxPQ(length);

for(int i = 0; i < length; i++) {

maxPQ.insert(nums[i]);

}

for(int i = 0; i < k - 1; i++) {

maxPQ.delMax();

}

return maxPQ.delMax();

}

private class MaxPQ {

int[] pq;

int n;

public MaxPQ(int initialCapacity) {

pq = new int[initialCapacity + 1];

n = 0;

}

public void insert(int x) {

pq[++n] = x;

swim(n);

}

public int delMax() {

int max = pq[1];

exch(1, n--);

sink(1);

return max;

}

public void swim(int k) {

while(k > 1 && less(k/2, k)) {

exch(k/2, k);

k = k/2;

}

}

public void sink(int k) {

while(k <= n/2) {

int j = 2 \* k;

if(j < n && less(j, j + 1)) {

j++;

}

if(!less(k, j)) {

break;

}

exch(k, j);

k = j;

}

}

public void exch(int v, int w) {

int swap = pq[v];

pq[v] = pq[w];

pq[w] = swap;

}

public boolean less(int v, int w) {

// The error prone format here

// return pq[v] - pq[w] < 0;

// but as this question will find the kth largest number, and -2147483648 is surely

// sit at the last position, so will not reflect problem here

// Refer to

// https://github.com/lampardchelsea/hello-world/blob/master/leetcode/array/ThirdMaximumNumber.java

return pq[v] < pq[w];

}

}

}

**Refer to**

<https://leetcode.com/problems/kth-largest-element-in-an-array/editorial/>

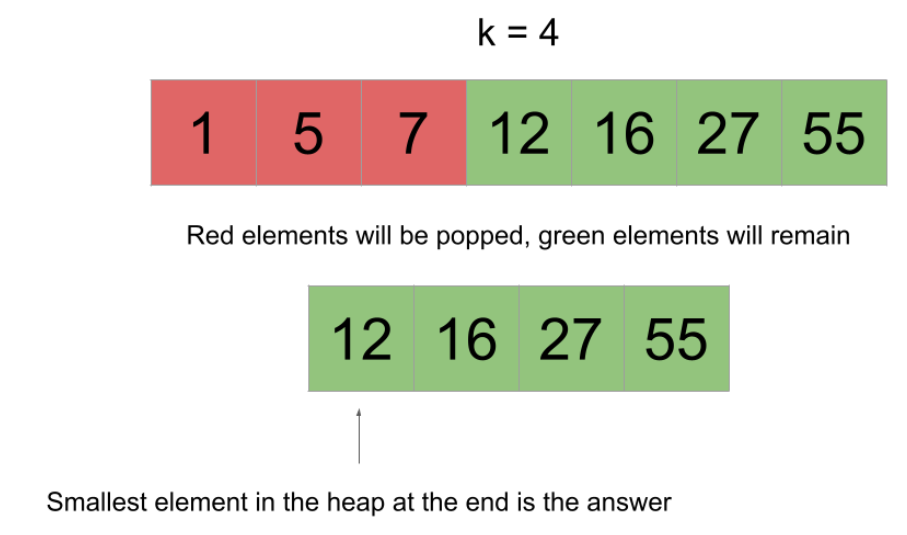
#### **Approach 2: Min-Heap**

**Intuition**

A heap is a very powerful data structure that allows us to efficiently find the maximum or minimum value in a dynamic dataset.

If you are not familiar with heaps, we recommend checking out the [Heap Explore Card](https://leetcode.com/explore/learn/card/heap/).

The problem is asking for the kth largest element. Let's push all the elements onto a min-heap, but pop from the heap when the size exceeds k. When we pop, the smallest element is removed. By limiting the heap's size to k, after handling all elements, the heap will contain exactly the k largest elements from the array.



It is impossible for one of the green elements to be popped because that would imply there are at least k elements in the array greater than it. This is because we only pop when the heap's size exceeds k, and popping removes the smallest element.

After we handle all the elements, we can just check the top of the heap. Because the heap is holding the k largest elements and the top of the heap is the smallest element, the top of the heap would be the kth largest element, which is what the problem is asking for.

**Algorithm**

Initialize a min-heap heap.

Iterate over the input. For each num:

Push num onto the heap.

If the size of heap exceeds k, pop from heap.

Return the top of the heap.

**Implementation**

Note: C++ [std::priority\_queue](https://en.cppreference.com/w/cpp/container/priority_queue) implements a max-heap. To achieve min-heap functionality, we will multiply the values by -1 before pushing them onto the heap.

class Solution {

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

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

for (int num: nums) {

heap.add(num);

if (heap.size() > k) {

heap.remove();

}

}

return heap.peek();

}

}

**Complexity Analysis**

Given n as the length of nums,

Time complexity: O(n⋅log⁡k)

Operations on a heap cost logarithmic time relative to its size. Because our heap is limited to a size of k, operations cost at most O(log⁡k). We iterate over nums, performing one or two heap operations at each iteration.

We iterate n times, performing up to log⁡k work at each iteration, giving us a time complexity of O(n⋅log⁡k)

Because k≤n , this is an improvement on the previous approach.

Space complexity: O(k)

The heap uses O(k) space.

**Solution 3: Quick Select (60 min)**

class Solution {

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

// Should not use Arrays.asList() to work with primitive

// type array such as int[] nums, it only work with objects

// array such as String[] strs, use for loop to finish

// or below

// List<Integer> list = Arrays.stream(ints).boxed().collect(Collectors.toList());

// Refer to

// https://stackoverflow.com/questions/1073919/how-to-convert-int-into-listinteger-in-java

List<Integer> list = new ArrayList<Integer>();

for(int num : nums) {

list.add(num);

}

return quickSelect(list, k);

}

private int quickSelect(List<Integer> nums, int k) {

// Select a random value in an ArrayList

// Random.nextInt() method, it accepts a bound parameter,

// which sets the upper bound, and sets the lower bound

// to 0 by default, so the randome number select between 0

// (inclusive) and n (exclusive)

int pivotIndex = new Random().nextInt(nums.size());

int pivot = nums.get(pivotIndex);

// Normally the quick select algorithm use to find kth minimum

// element, we will store all numbers smaller than 'pivot'

// in left list and all numbers larger than 'pivot' in right

// list, but in problem here we have to find kth maximum element,

// we have to swap the storage of the two lists, which means

// store all numbers larger than 'pivot' in left list and all

// numbers smaller than 'pivot' in right list

List<Integer> left = new ArrayList<Integer>();

List<Integer> mid = new ArrayList<Integer>();

List<Integer> right = new ArrayList<Integer>();

for(int num : nums) {

if(num > pivot) {

left.add(num);

} else if(num < pivot) {

right.add(num);

} else {

mid.add(num);

}

}

// If number of elements in left list >= k, the answer must be

// in left list, all remain elements would be less than kth

// largest element, we restart the process against left list

// e.g nums = {12,5,16,1,7,55,27}, k = 4

// randomly select 7 as pivot

// left = {12,16,55,27}

// mid = {7}

// right = {5,1}

// left.size() = 4 >= k, answer(= 12) must be in left list

if(left.size() >= k) {

return quickSelect(left, k);

}

// If total number of elements in left + mid list < k, the answer

// must be in right list, all elements in left + mid are too large

// for kth largest element, we restart the process against right

// list, but compare to (left list size >= k) condition, we need

// extra step to truncate the total elements number in left + mid

// list from k, which means during the reprocess we only looking for

// (k - left.size() - mid.size())th element in right list

// e.g nums = {12,5,16,1,7,55,27}, k = 4

// randomly select 27 as pivot

// left = {55}

// mid = {27}

// right = {12,5,16,1,7}

// left.size() + mid.size() = 2 < 4, answer(= 12) must be in right list

// but instead of search for 4th largest element in right list, we only

// need to find 4 - (1 + 1) = 2nd largest element by truncating two

// larger than pivot elements as {55} and {27} from left and mid list

if(left.size() + mid.size() < k) {

return quickSelect(right, k - left.size() - mid.size());

}

return pivot;

}

}

Time complexity: O(n) on average, O(n^2) in the worst case

Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/kth-largest-element-in-an-array/editorial/>

#### **Approach 3: Quickselect**

**Intuition**

This is a more advanced/esoteric algorithm. Do not feel discouraged if you are unable to derive it yourself. It is highly unlikely that you would be expected to come up with this solution in an interview without any help from the interviewer.

[Quickselect](https://en.wikipedia.org/wiki/Quickselect), also known as **Hoare's selection algorithm**, is an algorithm for finding the kth smallest element in an unordered list. It is significant because it has an **average** runtime of O(n).

Quickselect uses the same idea as Quicksort. First, we choose a pivot index. The most common way to choose the pivot is randomly. We partition nums into 3 sections: elements equal to the pivot, elements greater than the pivot, and elements less than the pivot.

Next, we count the elements in each section. Let's denote the sections as follows:

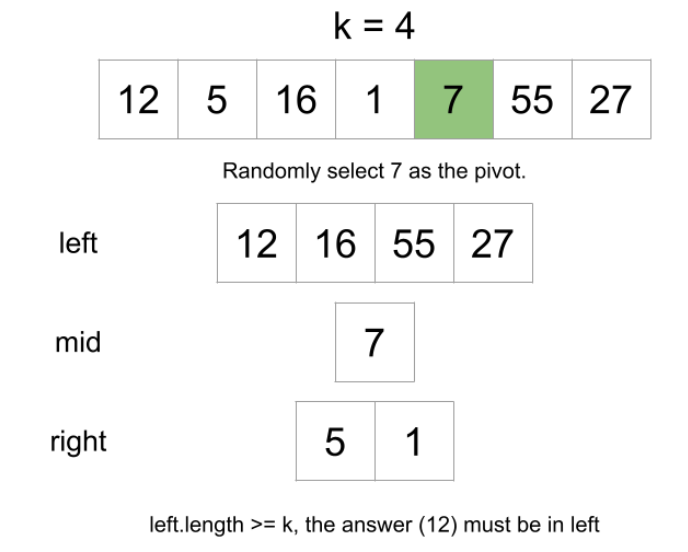
left is the section with elements less than the pivot

mid is the section with elements equal to the pivot

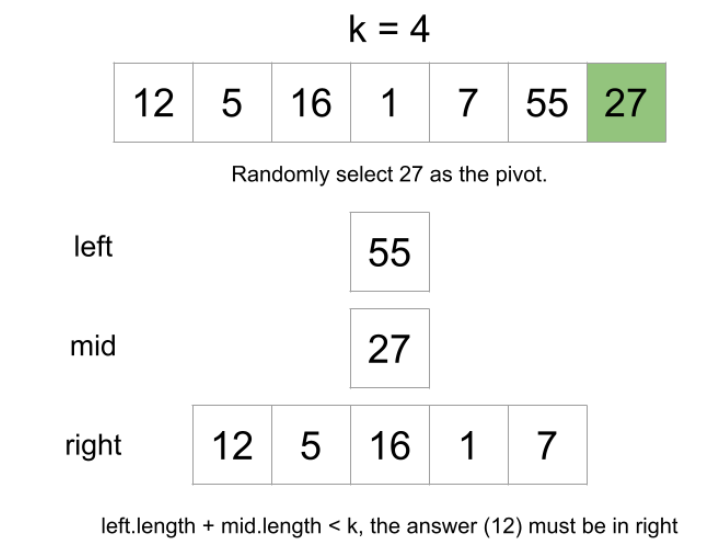
right is the section with elements greater than the pivot

Quickselect is normally used to find the kth **smallest** element, but we want the kth **largest**. To account for this, we will swap what left and right represent - left will be the section with elements greater than the pivot and right will be the section with elements less than the pivot.

If the number of elements in left is greater than or equal to k, the answer must be in left- any other element would be less than the kth largest element. We restart the process in left.

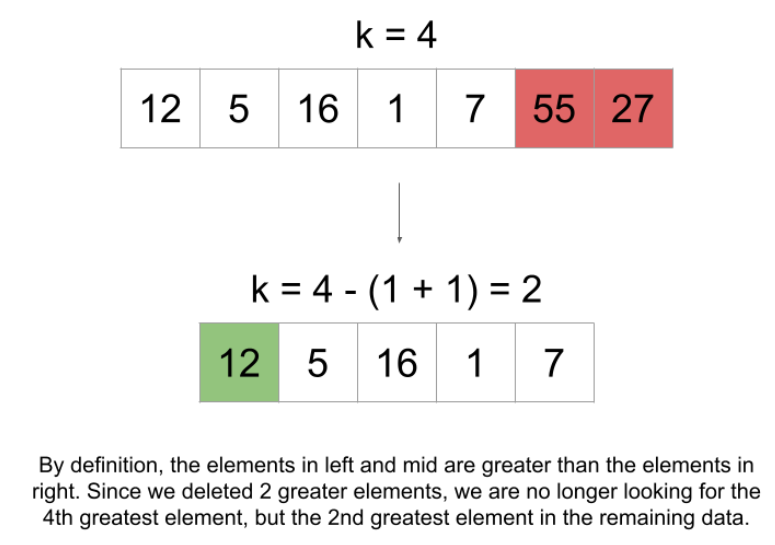


If the number of elements in left and mid is less than k, the answer must be in right - any element in left or mid would be too large to be the kth largest element. We restart the process in right



There's one extra step if the answer is in right. When we go to search in right, we are effectively "deleting" the elements in left and mid(since they will never be considered again). Because the elements in left and mid are greater than the answer, deleting them means we must shift k. Let's say we're looking for the 8th greatest element, but then we delete the 4 greatest elements. In the remaining data, we would be looking for the 4th greatest element, not the 8th. Therefore, we need to subtract the length of left and mid from k when we search in right.

We don't need to modify k when we search in left because when we search in left, we delete elements smaller than the answer, which doesn't affect k.



If the answer is in neither left or right, it must be in mid. Since mid only has elements equal to the pivot, the pivot must be the answer.

The easiest way to implement this repetitive process is by using recursion.

**Algorithm**

Note: the implementation we use here is not a standard Quickselect implementation. We will be using slightly more space (still the same complexity), but in exchange, we will be writing significantly less code.

Define a quickSelect function that takes arguments nums and k. This function will return the kth greatest element in nums(the nums and k given to it as input, not the original nums and k).

Select a random element as the pivot.

Create left, mid, and right as described above.

If k <= left.length, return quickSelect(left, k).

If left.length + mid.length < k, return quickSelect(right, k - left.length - mid.length).

Otherwise, return pivot.

Call quickSelect with the original nums and k, and return the answer.

**Implementation**

class Solution {

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

List<Integer> list = new ArrayList<>();

for (int num: nums) {

list.add(num);

}

return quickSelect(list, k);

}

public int quickSelect(List<Integer> nums, int k) {

int pivotIndex = new Random().nextInt(nums.size());

int pivot = nums.get(pivotIndex);

List<Integer> left = new ArrayList<>();

List<Integer> mid = new ArrayList<>();

List<Integer> right = new ArrayList<>();

for (int num: nums) {

if (num > pivot) {

left.add(num);

} else if (num < pivot) {

right.add(num);

} else {

mid.add(num);

}

}

if (k <= left.size()) {

return quickSelect(left, k);

}

if (left.size() + mid.size() < k) {

return quickSelect(right, k - left.size() - mid.size());

}

return pivot;

}

}

**Complexity Analysis**

Given n as the length of nums,

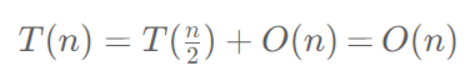
* Time complexity: O(n) on average, O(n^2) in the worst case

Each call we make to quickSelect will cost O(n) since we need to iterate over nums to create left, mid, and right. The number of times we call quickSelect is dependent on how the pivots are chosen. The worst pivots to choose are the extreme (greatest/smallest) ones because they reduce our search space by the least amount. Because we are randomly generating pivots, we may end up calling quickSelect O(n)times, leading to a time complexity of O(n^2).

However, the algorithm mathematically [almost surely](https://en.wikipedia.org/wiki/Almost_surely) has a linear runtime. For any decent size of nums, the probability of the pivots being chosen in a way that we need to call quickSelect O(n) times is so low that we can ignore it.

On average, the size of nums will decrease by a factor of ~2 on each call. You may think: that means we call quickSelect O(log⁡n) times, wouldn't that give us a time complexity of O(n⋅log⁡n) ? Well, each successive call to quickSelect would also be on a nums

that is a factor of ~2 smaller. This recurrence can be analyzed using the [master theorem](https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)) with  a = 1, b = 2, k = 1:



* Space complexity: O(n)

We need O(n) space to create left, mid, and right. Other implementations of Quickselect can avoid creating these three in memory, but in the worst-case scenario, those implementations would still require O(n) space for the recursion call stack.

**Bonus**

When we randomly choose pivots, Quickselect has a worst-case scenario time complexity of O(n^2).

By using the [median of medians](https://en.wikipedia.org/wiki/Median_of_medians) algorithm, we can improve to a worst-case scenario time complexity of O(n).

This approach is way out of scope for an interview, and practically it isn't even worth implementing because there is a large constant factor. As stated above, the random pivot approach will yield a linear runtime with mathematical certainty, so in all practical scenarios, it is sufficient.

The median of medians approach should only be appreciated for its theoretical beauty. Those who are interested can read more using the link above.