Solution

Approach 0: Sort

The naive solution would be to sort an array first and then return kth element from the end, something like sorted(nums)[-k] on Python. That would be an algorithm of \mathcal{O}(N \log N)O(*N*log*N*) time complexity and \mathcal{O}(1)O(1) space complexity. This time complexity is not really exciting so let's check how to improve it by using some additional space.

Approach 1: Heap

The idea is to init a heap "the smallest element first", and add all elements from the array into this heap one by one keeping the size of the heap always less or equal to k. That would results in a heap containing k largest elements of the array.

The head of this heap is the answer, i.e. the kth largest element of the array.

The time complexity of adding an element in a heap of size k is \mathcal{O}(\log k)O(log*k*), and we do it N times that means \mathcal{O}(N \log k)O(*N*log*k*) time complexity for the algorithm.

In Python there is a method nlargest in heapq library which has the same \mathcal{O}(N \log k)O(*N*log*k*) time complexity and reduces the code to one line.

This algorithm improves time complexity, but one pays with \mathcal{O}(k)O(*k*) space complexity.

class Solution {

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

// init heap 'the smallest element first'

PriorityQueue<Integer> heap =

new PriorityQueue<Integer>((n1, n2) -> n1 - n2);

// keep k largest elements in the heap

for (int n: nums) {

heap.add(n);

if (heap.size() > k)

heap.poll();

}

// output

return heap.poll();

}

}

**Complexity Analysis**

* Time complexity : \mathcal{O}(N \log k)O(*N*log*k*).
* Space complexity : \mathcal{O}(k)O(*k*) to store the heap elements.

#### Approach 2: Quickselect

This [textbook algorthm](https://en.wikipedia.org/wiki/Quickselect) has \mathcal{O}(N)O(*N*) average time complexity. Like quicksort, it was developed by Tony Hoare, and is also known as Hoare's selection algorithm.

The approach is basically the same as for quicksort. For simplicity let's notice that kth largest element is the same as N - kth smallest element, hence one could implement kth smallest algorithm for this problem.

First one chooses a pivot, and defines its position in a sorted array in a linear time. This could be done with the help of partition algorithm.

To implement partition one moves along an array, compares each element with a pivot, and moves all elements smaller than pivot to the left of the pivot.

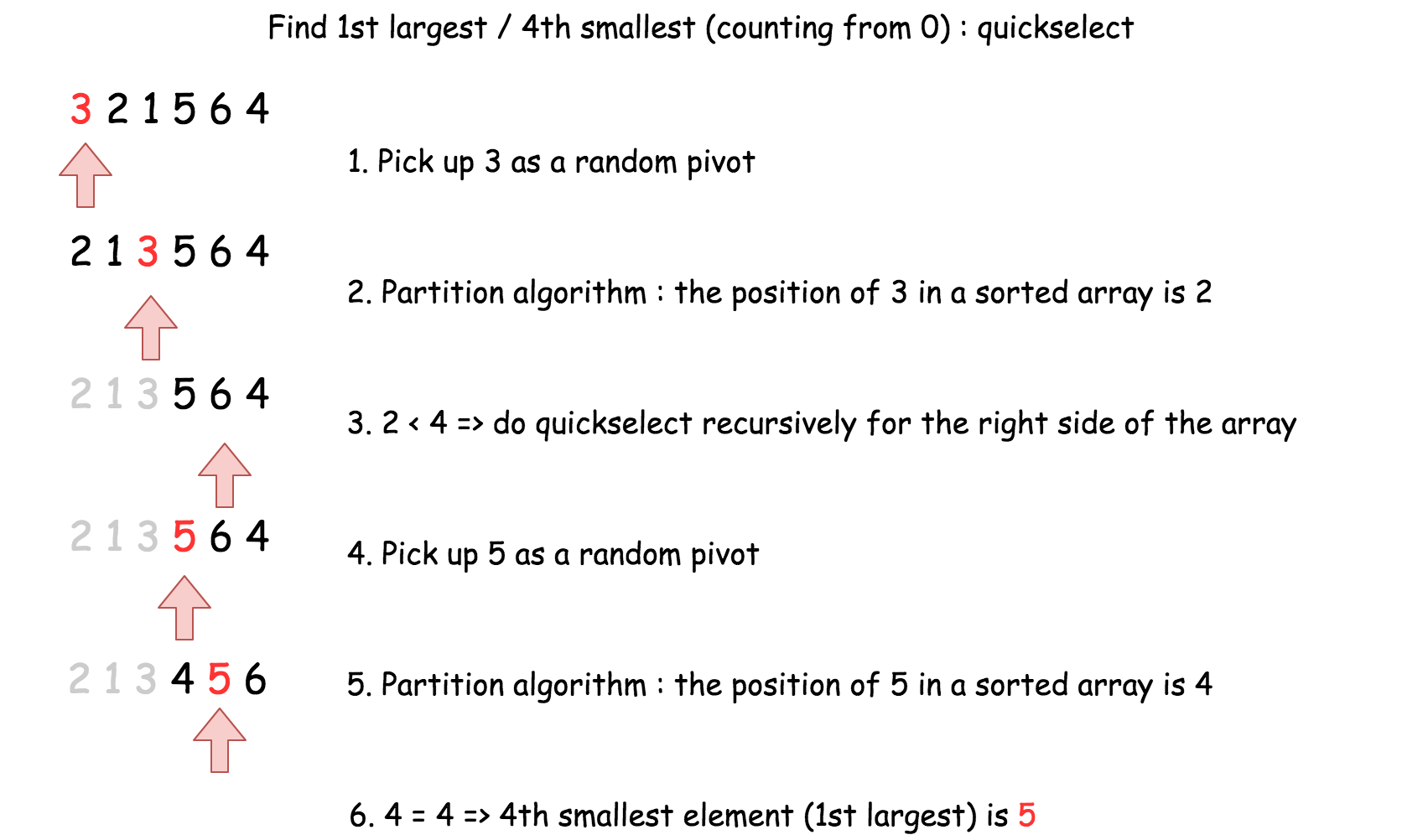
As an output we have an array where pivot is on its perfect position in the ascending sorted array, all elements on the left of the pivot are smaller than pivot, and all elements on the right of the pivot are larger or equal to pivot.

Hence the array is now split into two parts. If that would be a quicksort algorithm, one would proceed recursively to use quicksort for the both parts that would result in \mathcal{O}(N \log N)O(*N*log*N*) time complexity. Here there is no need to deal with both parts since now one knows in which part to search for N - kth smallest element, and that reduces average time complexity to \mathcal{O}(N)O(*N*).

Finally the overall algorithm is quite straightforward :

* Choose a random pivot.
* Use a partition algorithm to place the pivot into its perfect position pos in the sorted array, move smaller elements to the left of pivot, and larger or equal ones - to the right.
* Compare pos and N - k to choose the side of array to proceed recursively.

! Please notice that this algorithm works well even for arrays with duplicates.



import java.util.Random;

class Solution {

int [] nums;

public void swap(int a, int b) {

int tmp = this.nums[a];

this.nums[a] = this.nums[b];

this.nums[b] = tmp;

}

public int partition(int left, int right, int pivot\_index) {

int pivot = this.nums[pivot\_index];

// 1. move pivot to end

swap(pivot\_index, right);

int store\_index = left;

// 2. move all smaller elements to the left

for (int i = left; i <= right; i++) {

if (this.nums[i] < pivot) {

swap(store\_index, i);

store\_index++;

}

}

// 3. move pivot to its final place

swap(store\_index, right);

return store\_index;

}

public int quickselect(int left, int right, int k\_smallest) {

/\*

Returns the k-th smallest element of list within left..right.

\*/

if (left == right) // If the list contains only one element,

return this.nums[left]; // return that element

// select a random pivot\_index

Random random\_num = new Random();

int pivot\_index = left + random\_num.nextInt(right - left);

pivot\_index = partition(left, right, pivot\_index);

// the pivot is on (N - k)th smallest position

if (k\_smallest == pivot\_index)

return this.nums[k\_smallest];

// go left side

else if (k\_smallest < pivot\_index)

return quickselect(left, pivot\_index - 1, k\_smallest);

// go right side

return quickselect(pivot\_index + 1, right, k\_smallest);

}

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

this.nums = nums;

int size = nums.length;

// kth largest is (N - k)th smallest

return quickselect(0, size - 1, size - k);

}

}

* Time complexity : \mathcal{O}(N)O(*N*) in the average case, \mathcal{O}(N^2)O(*N*2) in the worst case.
* Space complexity : \mathcal{O}(1)O(1).