LeetCode 215

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

Yifeng Zeng

Description

215. Kth Largest Element in an Array

Find the kth largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

For example,

Given [3,2,1,5,6,4] and k = 2, return 5.

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

Idea Report

I always have a primitive idea to solve "top k" problem by using a heap. We can manually maintain a size k min heap, where this min heap contains k largest number in the input array. When new a number n comes in, we compare n to the smallest of the k numbers in the heap (heap.peek()) and see if n is larger, if it is larger, then the we poll out the smallerst number in the min heap because it cannot be the kth largest number, then put n in the min heap as a candidate. In the end, the smallest number in the min heap is the kth largest number in the array. This takes O(nlogk), because we have n numbers and for each number we need logk time to update the heap.

Code

```
class Solution {
    // AC
    public int findKthLargest(int[] nums, int k) {
```

```
        Queue<Integer> pq = new PriorityQueue<>(); // min heap
        for (int n : nums) {
            if (pq.size() < k) {
                pq.offer(n);
            } else {
                if (pq.peek() < n) {
                    pq.poll();
                    pq.offer(n);
                }
            }
        }
        return pq.peek();
    }
}
```

Another way is to utilize the method of the binary search, basically throw out half of the impossible candidates at a time and sort the other half possible candidates, similar to quick sort. We first decrease k by one then we can just find the element at index k where the array is partially sorted in decreasing order. Like quick sort, each time we choose a pivot (I like to use the mid between start and end), and we move numbers larger to pivot to the left of the array, and move number smaller to pivot to the right of the array. And then we check index k is at the left part (larger than pivot), or at the right part (smaller than pivot). We choose the correct part and look for the kth index recursively, and ditch throw away the other part which is not longer possible. Let me give an example, we can just use the given example [3,2,4,5,6], k = 2. 's' is start, 'e' is end, 'p' is pivot, 'l' is left, 'r' is right. We decrese k by 1 to find the index k = 1.

- Step 1, we see nums[l] = 3 <= p="4," nums[r]="6">= p = 4, so we swap them.
- Step 2, we see nums[l] = 2 <= p="4," nums[r]="5">= p = 4, so we swap them.
- Step 3, we see nums[l] = 4 <= p="4," nums[r]="5">= p = 4, so we swap them as well, after swapping, index r = 1, index l = 3, the array are splitted into three parts.
    - Part 1, [s,r] = [0,1], which has all elements larger or equal to pivot
    - Part 2, (r, l) = (1,3) = [2], which has all elements equal to pivot
    - Part 3, [l,e] = [3,4], which has all elements smaller or equal to pivot
- Step 4, comparing indices s,r,l,e to index k, we know which part index k belongs to, then we recursively do the search in that part.
- Step 5, in our example, k = 1, so k belongs to Part 1, we do search recursively in Part 1. Now s = 0, e = 1, p = 6. Becaues nums[r] = nums[1] = 5 < pivot = 6, we move r to left.
- Step 6, we see nums[l] = 6 <= p="6," nums[r]="6">= p = 6, so we swap them, and now r = -1, l = 1.
- Step 7, we check k = 1, belongs to part 3 [l,e] = [1,1], we seach recursively
- Step 8, we find s == e = 1, so we just return nums[s] = 5.
  So we find the final result 5. The time complexity is O(n), because we throw away about half

of the candidates at a time and only care about the other half which has the correct answer.

```
1. [3,2,4,5,6], p = 4
    s       e
    l       r  -> swap
2. [6,2,4,5,3]
    s       e
      l   r    -> swap
3. [6,5,4,2,3]
    s        e
        lr     -> swap
4. [6,5,4,2,3]
    s        e
      r   l    -> throw away half
5. [6,5,4,2,3], p = 6
    s e
    l r        -> move r to left because nums[r] < pivot
6. [6,5,4,2,3], p = 6
    s e
    l
    r          -> swap
7. [6,5,4,2,3], p = 6
    s e
      l
  r = - 1      -> l <= k && k <= e, recursive
8. [6,5,4,2,3], p = 6
      s
      e        -> s == e, return nums[s] = 5
```

Code

```java
class Solution {
    // AC
    public int findKthLargest(int[] nums, int k) {
        // k is the index now.
        return quickSelect(nums, 0, nums.length - 1, k - 1);
    }

    private int quickSelect(int[] nums, int start, int end, int k) {
        if (start == end) {
            return nums[start];
        }

        int left = start;
        int right = end;
        int mid = (end - start) / 2 + start;
        int pivot = nums[mid];
```

```
        while (left <= right) {
            while (left <= right && nums[left] > pivot) {
                left++;
            }
            while (left <= right && nums[right] < pivot) {
                right--;
            }
            if (left <= right) {
                swap(nums, left, right);
                left++;
                right--;
            }
        }

        if (start <= k && k <= right) {
            return quickSelect(nums, start, right, k);
        } else if (left <= k && k <= end) {
            return quickSelect(nums, left, end, k);
        }
        return nums[right + 1];
    }

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

Summary

- A "top k" problem may be solved by maintaining a heap.
- Throw away half of candidates at a time, binary seach methodology.