215 数组中的第K个最大元素

Label：堆、快排

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4

- API 排序

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        Arrays.sort(nums);
        return nums[nums.length - k];
    }
}
```

- API 建立大根堆

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a,b)->b-a);
        for (Integer i : nums) {
            maxHeap.add(i);
        }
        for (int i = 1; i < k; i++) {
            maxHeap.poll();
        }
        return maxHeap.poll();
    }
}
```

- 大根堆

```java
class Solution {
    public int findKthLargest(int[] nums, int k) {
        int heapSize = nums.length;
        buildMaxHeap(nums, heapSize);  // 建立大根堆

        for (int i = nums.length - 1; i >= nums.length - k + 1; --i) { // 出堆
            swap(nums, 0, i);
            --heapSize;
            maxHeapify(nums, 0, heapSize);
        }
        return nums[0];
    }

    public void buildMaxHeap(int[] a, int heapSize) {
        for (int i = heapSize / 2; i >= 0; --i) {
            maxHeapify(a, i, heapSize);
        }
    }

    public void maxHeapify(int[] a, int i, int heapSize) {
        int l = i * 2 + 1, r = i * 2 + 2, largest = i;
        if (l < heapSize && a[l] > a[largest]) {
            largest = l;
        }
        if (r < heapSize && a[r] > a[largest]) {
            largest = r;
        }
        if (largest != i) {
            swap(a, i, largest);
            maxHeapify(a, largest, heapSize);
        }
    }

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

- 快排

```java
public class Solution {

    public int findKthLargest(int[] nums, int k) {
        int len = nums.length;
        int left = 0;
        int right = len - 1;

        // 转换一下，第 k 大元素的索引是 len - k
        int target = len - k;
        while (true) {
            int index = partition(nums, left, right);
            if (index == target) {
                return nums[index];
            } else if (index < target) {
                left = index + 1;
            } else {
                right = index - 1;
            }
        }
    }

    /**
     * 在数组 nums 的子区间 [left, right] 执行 partition 操作，返回 nums[left] 排序以后应该在的位置
     * 在遍历过程中保持循环不变量的语义
     * 1、[left + 1, j] < nums[left]
     * 2、(j, i] >= nums[left]
     */
    public int partition(int[] nums, int left, int right) {
        int pivot = nums[left];
        int j = left;
        for (int i = left + 1; i <= right; i++) {
            if (nums[i] < pivot) {
                // 小于 pivot 的元素都被交换到前面
                j++;
                swap(nums, j, i);
            }
        }
        // 在之前遍历的过程中，满足 [left + 1, j] < pivot，并且 (j, i] >= pivot
        swap(nums, j, left);
        // 交换以后 [left, j - 1] < pivot, nums[j] = pivot, [j + 1, right] >= pivot
        return j;
    }

    private void swap(int[] nums, int index1, int index2) {
        int temp = nums[index1];
        nums[index1] = nums[index2];
        nums[index2] = temp;
    }
}
```