Find the 3rd Largest in an Unsorted Array

Problem Description

Given a non-empty array of integers, return the third maximum number in this array.
If it does not exist, return the minimum number.

> Example 1:
> Input: [3, 2, 1]
> Output: 1
> Explanation: The third maximum is 1.

> Example 2:
> Input: [1, 2]
> Output: 1
> Explanation: The third maximum does not exist, so the minimum(1) is returned instead.

> Example 3:
> Input: [2, 2, 3, 1] Output: 2
>
> Explanation: Note that the third maximum here means the third maximum number.
> Both numbers with value 2 are considered as different maximum numbers.

Approach I: Use Arrays.sort() function

Algorithm

A quite obvious solution is to sort this array in a natural order and then output the Kth element from the end.

Implementation

Java

```
public class KthLargest_Arrays {
    public static int find(int[] nums, int k) {
        int n = Math.min(k, nums.length);
        Arrays.sort(nums);
        return nums[nums.length - n];
    }
}
```

Unit Test

The following main() is to verify the code.

Java

```
public static void main(String[] args) {
    int k = 3;
    int[] numbers = { 7, 4, 7, 4, 9, 1, 4, 1, 9, 10 };
    System.out.println(find(numbers, k));
}
```

Note:

- Put this function in the class to do unit test
- The main() could be used to test other solutions as well.

Test Case

Input: { 7, 4, 7, 4, 9, 1, 4, 1, 9, 10 } Output: 7

Complexity

Time: $O(log(n))$. Arrays.sort() is a quick sort solution, so the complexity is $O(logn)$.
Space: $O(log(n))$. The Java version of quicksort has a space complexity of O(log n), even in the worst case.

Approach II: Use PriorityQueue

Algorithm

- Create a priority queue with size k
- Put the last k elements in the queue as a start
- Continue to iterator this array forward
    - if the current number nums[i] > the head element in this array (which is also the smallest since the priority queue sorted the elements in a natural order)
    - remove the head element and put this number nums[i] in the queue
- End when it reaches the first element, and the first number in the queue is the result

Implementation

Java

```
public class KthLargest_PriorityQueue {
    public static int find(int[] nums, int k) {
        k = Math.min(nums.length, k);
        PriorityQueue<Integer> queue = new PriorityQueue<>(k);
        int i = nums.length - 1;
        while (k-- > 0) {
            queue.add(nums[i--]);
        }
        while (i-- > 0) {
            if (nums[i] >= queue.peek()) {
                queue.poll();
                queue.add(nums[i]);
            }
        }
        return queue.peek();
      }
}
```

Complexity

Time: $O(n * log(n))$. Like insertion sort, the worst case if the array is in a descending order when k = length. **Space**: $O(k)$. Obviously, we need a queue with size k to hold the first largest k elements.

Approach III: Ultilize quick sort

Algorithm

Here is the modified quick sort algorithm to solve this problem.

- Take a number in this array (nums[ ]) a pivot number, such as the last number
- Partition nums[] so that the left of which is smaller or equal than the pivot, while the right is bigger
- Switch the position of the pivot number and the partition

  The original problem equals to find the n-th smallest number if n = nums.length - k.

  - When left positon equals n, return nums[left]
  - If left is lesss than n,
- Continue to do so until every number is processed

So, we could find the Kth largest element earlier during the process before the quick sort completes.

> Note: Quick sort is in ascending order to the kth largest means the (length-k) smallest

Implementation

Java

```java
import java.util.Arrays;
public class KthLargest {
    public static int find(int[] nums, int k) {
        if (nums == null || nums.length == 0)
            return Integer.MAX_VALUE;
        return find(nums, 0, nums.length - 1, nums.length - k);
    }

    private static int find(int[] nums, int start, int end, int n) {
        n = Math.min(n, nums.length);
        assert start <= end;
        int pivot = nums[end];
        int left = start;
        for (int i = start; i < end; i++) {
            if (nums[i] <= pivot)
                swap(nums, left++, i);
        }
        swap(nums, left, end);
        if (left == n)
            return nums[left];
        else if (left < n)
            return find(nums, left + 1, end, n);
        else
            return find(nums, start, left - 1, n);
    }

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

Complexity

Time: $O(log(n))$. It is similar to the quick sort.

Space: $O(log(n))$. The resursion of find() consume this level of space since it divide the invoking half each time.

Follow-up

What if asking for the Kth distinct largest number?

> Example: Input: [2, 2, 3, 1] Output: 1

> Explanation: Note that the third maximum here means the third maximum distinct number. Both numbers with value 2 are both considered as second maximum.

To achieve this, we need to modify the condition when comparing the data with the pivot/guard. This is the solution using PriorityQueue.

Java

```java
import java.util.PriorityQueue;
public class KthDistinctLargest {
    public static int find(int[] nums, int k) {
        k = Math.min(nums.length, k);
        PriorityQueue<Integer> queue = new PriorityQueue<>(k);
        int i = nums.length;
        while (k-- > 0) {
            queue.add(nums[--i]);
        }
        while (--i >= 0) {
            // pass it when the number is already in the queue
            if (!queue.contains(nums[i]) && nums[i] > queue.peek()) {
                queue.poll();
                queue.add(nums[i]);
            }
        }
        return queue.peek();
    }
}
```

Conclusion

These three implementations have the same level of time complexity $O(n * log(n))$.