<https://leetcode.com/problems/third-maximum-number/description/>

Given an integer array nums, return *the* ***third distinct maximum*** *number in this array. If the third maximum does not exist, return the* ***maximum*** *number*.

**Example 1:**

Input: nums = [3,2,1]

Output: 1

Explanation:

The first distinct maximum is 3.

The second distinct maximum is 2.

The third distinct maximum is 1.

**Example 2:**

Input: nums = [1,2]

Output: 2

Explanation:

The first distinct maximum is 2.

The second distinct maximum is 1.

The third distinct maximum does not exist, so the maximum (2) is returned instead.

**Example 3:**

Input: nums = [2,2,3,1]

Output: 1

Explanation:

The first distinct maximum is 3.

The second distinct maximum is 2 (both 2's are counted together since they have the same value).

The third distinct maximum is 1.

**Constraints:**

1 <= nums.length <= 104

-231 <= nums[i] <= 231 - 1

**Follow up:** Can you find anO(n)solution?

**Attempt 1: 2023-11-09**

**Solution 1: Priority Queue (60 min)**

**Note: To find Kth largest element always use MinPQ, to find Kth smallest element always use MaxPQ, refer L378/P14.2/P15.2.Kth Smallest Element in a Sorted Matrix (use MaxPQ with size limit) & L703/P14.7.Kth Largest Element in a Stream (use MinPQ with size limit)**

**Style 1: Set + PriorityQueue (capacity limit as 3, no more number will add on element, only update one of these 3 numbers when condition match) + No need remove number from set**

class Solution {

public int thirdMax(int[] nums) {

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

Set<Integer> set = new HashSet<>();

int max = Integer.MIN\_VALUE;

for(int num : nums) {

max = Math.max(max, num);

if(!set.contains(num)) {

if(minPQ.size() < 3) {

minPQ.offer(num);

set.add(num);

} else if(minPQ.peek() < num) {

// Any number obsolete from minPQ even still in set

// will have no chance to add to minPQ again, so no

// need to remove number from set

//set.remove(minPQ.poll());

minPQ.poll();

minPQ.offer(num);

set.add(num);

}

}

}

return minPQ.size() == 3 ? minPQ.peek() : max;

}

}

Time Complexity: O(N), not O(N\*logN) because minPQ size not exceed 3, log3 can be treated as constant

Space Complexity: O(N)

**Style 2: Only PriorityQueue (capacity limit as 3, no more number will add on element, only update one of these 3 numbers when condition match)**

class Solution {

public int thirdMax(int[] nums) {

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

int max = Integer.MIN\_VALUE;

for(int num : nums) {

max = Math.max(max, num);

if(!minPQ.contains(num)) {

if(minPQ.size() < 3) {

minPQ.offer(num);

} else if(minPQ.peek() < num) {

// Any number obsolete from minPQ even still in set

// will have no chance to add to minPQ again, so no

// need to remove number from set

//set.remove(minPQ.poll());

minPQ.poll();

minPQ.offer(num);

}

}

}

return minPQ.size() == 3 ? minPQ.peek() : max;

}

}

Time Complexity: O(N) ~ O(N\*3), not O(N\*logN) because minPQ size not exceed 3, log3 can be treated as constant, but multiply 3 because of PriorityQueue contains() method cost O(N) and N = 3 here, but O(3) is also a constant

Space Complexity: O(N)

**Style 3: Set + PriorityQueue (capacity NOT limit as 3, 4th number will add on queue, always need to update one of these 3 numbers by removing operation, which means Style 3 significantly slower than Style 1) + No need remove number from set**

class Solution {

public int thirdMax(int[] nums) {

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

Set<Integer> set = new HashSet<>();

int max = Integer.MIN\_VALUE;

for(int num : nums) {

max = Math.max(max, num);

if(!set.contains(num)) {

minPQ.offer(num);

set.add(num);

if(minPQ.size() > 3) {

// No need to remove from set since set only

// used to filter unique key which should add

// onto minPQ, even it store previous number

// and not removed won't impact how minPQ store

// --------------------------------------------

// Example 1: nums = {1,2,5,3,1}

// minPQ = {1,2,5}, set = {1,2,5},

// when 3 coming, minPQ = {1,2,3,5} -> then move

// out 1 because minPQ.size() > 3, minPQ = {2,3,5}

// two choices for set: not move or move 1 out

// (1) set = {1,2,3,5} if not move out 1

// (2) set = {2,3,5} if also move out 1

// Then if another 1 coming, minPQ will change

// to {1,2,3,5}, now for set will be

// (1) set = {1,2,3,5} no change because still

// have previous 1 and not add new coming 1

// (2) set = {1,2,3,5} add new coming 1 because

// we move previous 1 out

// Then because of minPQ.size() > 3, we will move new

// coming 1 out of minPQ again, and set will repeat

// previous two choices as either remove 1 or not

// We can see the obsolete {1} from set will have no

// chance to exist on minPQ at any case, even if any new

// coming 1 won't add onto minPQ anymore as 2 is

// the threshold as existing third maximum number on minPQ,

// no matter {1} on or off the set, it obsolete from

// minPQ forever

// --------------------------------------------

// Example 2: nums = {1,2,5,3,3}

// when first 3 coming, minPQ = {2,3,5} -> move out 1, now

// (1) set = {1,2,3,5} if not move out 1

// (2) set = {2,3,5} if also move out 1

// Then if another 3 coming, minPQ will not change by

// not adding 3 because set already contains 3 in both

// (1) and (2) scenarios

// We can see the obsolete {1} from set will have no

// chance to exist on minPQ at any case, even if any new

// coming 1 won't add onto minPQ anymore as 2 is

// the threshold as existing third maximum number on minPQ,

// no matter {1} on or off the set, it obsolete from

// minPQ forever

// --------------------------------------------

// Example 3: nums = {1,2,5,3,4}

// when 3 coming, minPQ = {1,2,3,5} -> then move

// out 1 because minPQ.size() > 3, minPQ = {2,3,5}

// two choices for set: not move or move 1 out

// (1) set = {1,2,3,5} if not move out 1

// (2) set = {2,3,5} if also move out 1

// Then if 4 coming, minPQ will change to {2,3,4,5}-> then move

// out 2 because minPQ.size() > 3, minPQ = {3,4,5}

// two choices for set: not move or move 2 out

// (1) set = {1,2,3,4,5} if not move out 2

// (2) set = {3,4,5} if also move out 2

// We can see the obsolete {1,2} from set will have no

// chance to exist on minPQ at any case, even if any new

// coming 1 or 2 won't add onto minPQ anymore as 3 is

// the threshold as existing third maximum number on minPQ,

// no matter {1,2} on or off the set, it obsolete from

// minPQ forever

// --------------------------------------------

// Conclusion:

// Any number obsolete from minPQ even still in set

// will have no chance to add to minPQ again, so no

// need to remove number from set

//int val = minPQ.poll();

//set.remove(val);

minPQ.poll();

}

}

}

return minPQ.size() == 3 ? minPQ.peek() : max;

}

}

Time Complexity: O(N), not O(N\*logN) because minPQ size not exceed 4, log4 can be treated as constant

Space Complexity: O(N)

**Style 4: Only PriorityQueue (capacity NOT limit as 3, 4th number will add on queue, always need to update one of these 3 numbers by removing operation, which means Style 3 significantly slower than Style 1)**

class Solution {

public int thirdMax(int[] nums) {

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

int max = Integer.MIN\_VALUE;

for(int num : nums) {

max = Math.max(max, num);

if(!minPQ.contains(num)) {

minPQ.offer(num);

if(minPQ.size() > 3) {

minPQ.poll();

}

}

}

return minPQ.size() == 3 ? minPQ.peek() : max;

}

}

Time Complexity: O(N) ~ O(N\*4), not O(N\*logN) because minPQ size not exceed 4, log4 can be treated as constant, but multiply 4 because of PriorityQueue contains() method cost O(N) and N = 4 here, but O(4) is also a constant

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/third-maximum-number/solutions/90190/java-priorityqueue-o-n-o-1/comments/94679>

The PriorityQueue has a contains method, so you don't need an extra Set. Since the pq's is no more than 3, the actual time complexity for the contains method is O(1) in our case. But Priority Queue contains method take o(n) time while set.contains take O(1).

Normally in case where k is not fixed it takes logk to insert an element into heap and total running complexity is O(nlogk) and space complexity is O(k)but here we are maintaining max 4 elements in heap which O(log4) to add one element and total would be O(nlog4) ~ O(n) and space O(4) which is constant.

This is general approach to find kth largest element in a array. You can also solve it using quickselect algo which would be O(n ) again.

**Solution 2: One Pass (10 min)**

class Solution {

public int thirdMax(int[] nums) {

// Why change to long ?

// Test out by nums = {1,2,-2147483648}

long max = Long.MIN\_VALUE;

long second\_max = Long.MIN\_VALUE;

long third\_max = Long.MIN\_VALUE;

for(int num : nums) {

if((long)num > max) {

third\_max = second\_max;

second\_max = max;

max = num;

} else if((long)num > second\_max && (long)num != max) {

third\_max = second\_max;

second\_max = num;

} else if((long)num > third\_max && (long)num != max && (long)num != second\_max) {

third\_max = num;

}

}

return (int)(third\_max == Long.MIN\_VALUE ? max : third\_max);

}

}

Time Complexity: O(N)

Space Complexity: O(1)