# Question

Given integer array nums, return *the third 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 third maximum is 1.

**Example 2:**

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

**Output:** 2

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

**Example 3:**

**Input:** nums = [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.

**Constraints:**

* 1 <= nums.length <= 104
* -231 <= nums[i] <= 231 - 1

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

# Solution

#### **Approach 1: Use a Set and Delete Maximums**

**Intuition**

Firstly, note that we can't simply sort the values and then select the third-to-end one, because the time complexity of sorting is O(n \, \log \, n)*O*(*n*log*n*) (where n*n* is the length of the input Array). This problem clearly states our solution must have a time complexity of O(n)*O*(*n*) though.

Also, note that there are 2 important pieces of information in the problem description that are easily overlooked:

1. If the third maximum doesn't exist, we must return the ***maximum*** (never the second maximum like one might assume!).
2. Duplicates should be ignored. We want the third maximum distinct value. i.e. for [8, 8, 8, 3, 1] the third maximum is 1, despite the fact that 8 appears three times.

We'll work with the following example Array.

[12, 3, 8, 9, 12, 12, 7, 8, 12, 4, 3, 8, 1]

If there were no duplicates in the Array, then a logical strategy would be as follows:

Find the maximum. Delete it.

Find the new maximum. Delete it.

Return the \*new\* maximum.

However, the input Array we're working with could have duplicates. To handle this, we can convert the input into a Set first to remove the duplicates.

Converting our input Array example into a Set gives us the following:

{12, 3, 8, 9, 7, 4, 1}

We then need to find the maximum in the Set. This can be done using a library function, or if necessary, your own function that loops through the list keeping track of the maximum seen so far, and then returns the maximum at the end.

The maximum from our example is 12.

Now, we need to delete 12 from the Set. This leaves us with:

{3, 8, 9, 7, 4, 1}

We can then find and remove the second maximum, following the same process.

The second maximum is 9 (the maximum of what's left in the Set).

Removing it leaves us with the following:

{3, 8, 7, 4, 1}

Finally, we can return the maximum of what's left, which is 8.

Remember that if the third maximum doesn't exist, then we need to return the maximum of the original Array. We can detect this situation as soon as we have converted the input Array into a Set, because it will contain less than 3 values.

**Algorithm**

|  |
| --- |
| **public int thirdMax(int[] nums) {**  **// Put the input integers into a HashSet.**  **Set<Integer> setNums = new HashSet<>();**  **for (int num : nums) setNums.add(num);**  **// Find the maximum.**  **int maximum = Collections.max(setNums);**  **// Check whether or not this is a case where we**  **// need to return the \*maximum\*.**  **if (setNums.size() < 3) {**  **return maximum;**  **}**  **// Otherwise, continue on to finding the third maximum.**  **setNums.remove(maximum);**  **int secondMaximum = Collections.max(setNums);**  **setNums.remove(secondMaximum);**  **return Collections.max(setNums);**  **}** |

**Complexity Analysis**

* Time Complexity : O(n)*O*(*n*).

Putting the input Array values into a HashSet has a cost of O(n)*O*(*n*), as each value costs O(1)*O*(1) to place, and there are n*n* of them.

Finding the maximum in a HashSet has a cost of O(n)*O*(*n*), as all the values need to be looped through. We do this 33 times, giving O(3 \cdot n) = O(n)*O*(3⋅*n*)=*O*(*n*) as we drop constants in big-oh notation.

Deleting a value from a HashSet has a cost of O(1)*O*(1), so we can ignore this.

In total, we're left with O(n) + O(n) = O(n)*O*(*n*)+*O*(*n*)=*O*(*n*).

* Space Complexity : O(n)*O*(*n*).

In the worst case, the HashSet is the same size as the input Array, and so requires O(n)*O*(*n*) space to store.

#### **Approach 2: Seen-Maximums Set**

**Intuition**

In the previous approach, we deleted the maximum and second maximum so that we could easily find the third maximum. We had to convert the input Array into a Set so that duplicates weren't super complicated to handle.

Instead of deleting items though, we could instead keep a Set of maximums we've already seen. Then when we are searching for a maximum, we can ignore any values that are already in the seen Set.

This will also handle duplicates elegantly—if for example we had the input set [12, 12, 4, 2, 12, 1], then the first value we'd put into the seen maximums Set would be 12. Then when we find the second maximum, the algorithm knows to ignore all the 12s.

**Algorithm**

|  |
| --- |
| **class Solution {**  **public int thirdMax(int[] nums) {**  **Set<Integer> seenMaximums = new HashSet<>();**    **for (int i = 0; i < 3; i++) {**  **Integer curMaximum = maxIgnoringSeenMaximums(nums, seenMaximums);**  **if (curMaximum == null) {**  **return Collections.max(seenMaximums);**  **}**  **seenMaximums.add(curMaximum);**  **}**  **return Collections.min(seenMaximums);**  **}**  **private Integer maxIgnoringSeenMaximums(int[] nums, Set<Integer> seenMaximums) {**    **Integer maximum = null;**  **for (int num : nums) {**  **if (seenMaximums.contains(num)) {**  **continue;**  **}**  **if (maximum == null || num > maximum) {**  **maximum = num;**  **}**  **}**  **return maximum;**  **}**  **}** |

**Complexity Analysis**

* Time Complexity : O(n)*O*(*n*).

For each of the three times we find the next maximum, we need to perform an O(n)*O*(*n*) scan. Because there are only, at most, three scans the total time complexity is just O(n)*O*(*n*).

The Set operations are all O(1)*O*(1) because there are only at most 3 items in the Set.

* Space Complexity : O(1)*O*(1).

Because seenMaximums can contain at most 3 items, the space complexity is only O(1)*O*(1).

#### **Approach 3: Keep Track of 3 Maximums Using a Set**

**Intuition**

So far, our approaches have required multiple parses through the input array. While this is still O(n)*O*(*n*) in big-oh notation, it'd be good if we could solve it in a single parse. One way is to simply use a Set to keep track of the 3 maximum values we've seen so far. While you could achieve something similar using 3 variables (maximum, secondMaximum, and thirdMaximum), this is messy to work with and is poor programming practice.

For each number in the Array, we add it into the Set of maximums. If this causes there to be more than 3 numbers in the Set, then we evict the smallest number.

At the end, we check whether or not there are 3 numbers in the Set. If there are, this means the third maximum exists, and will be the minimum in the Set. If not, this means there was no third maximum, and so we should return the maximum of the Set, as per the problem requirements.

Here is an animation showing the approach.

1 / 26

**Algorithm**

|  |
| --- |
| **public int thirdMax(int[] nums) {**  **Set<Integer> maximums = new HashSet<Integer>();**  **for (int num : nums) {**  **maximums.add(num);**  **if (maximums.size() > 3) {**  **maximums.remove(Collections.min(maximums));**  **}**  **}**  **if (maximums.size() == 3) {**  **return Collections.min(maximums);**  **}**  **return Collections.max(maximums);**  **}** |

**Complexity Analysis**

* Time Complexity : O(n)*O*(*n*).

For each of the n*n* values in the input Array, we insert it into a Set for a cost of O(1)*O*(1). We then sometimes find and remove the minimum of the Set. Because there are never more than 33 items in the Set, the time complexity of doing this is O(1)*O*(1).

In total, we're left with O(n)*O*(*n*).

* Space Complexity : O(1)*O*(1).

Because maximums never holds more than 33 items at a time, it is considered to be constant O(1)*O*(1).