Previous Next

414. Third Maximum Number

March 1, 2020 | 7.3K views

Average Rating: 4.67 (3 votes)

Given a **non-empty** array of integers, return the **third** maximum number in this array. If it does not exist, return the maximum number. The time complexity must be in O(n).

Example 1:

```
Input: [3, 2, 1]

Output: 1

Explanation: The third maximum is 1.
```

Example 2:

```
Input: [1, 2]

Output: 2

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

Example 3:

```
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.
```

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 $O(n \log n)$ (where n is the length of the input `Array`). This problem clearly states our solution must have a time complexity of $O(n)$ 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

Java | Python | Copy

```
class Solution:
    def thirdMax(self, nums: List[int]) -> int:

        # Make a Set with the input.
        nums = set(nums)

        # Find the maximum.
        maximum = max(nums)

        # Check whether or not this is a case where
        # we need to return the *maximum*.
        if len(nums) < 3:
            return maximum

        # Otherwise, continue on to finding the third maximum.
        nums.remove(maximum)
        second_maximum = max(nums)
        nums.remove(second_maximum)
        return max(nums)
```

Complexity Analysis

- Time Complexity : $O(n)$.

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

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

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

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

- Space Complexity : $O(n)$.

  In the worst case, the `HashSet` is the same size as the input `Array`, and so requires $O(n)$ 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 `12` s.

Algorithm

Java | Python | Copy

```
def thirdMax(self, nums: List[int]) -> int:

    def maximum_ignoring_seen_maximums(nums, seen_maximums):
        maximum = None
        for num in nums:
            if num in seen_maximums:
                continue
            if maximum == None or num > maximum:
                maximum = num
        return maximum

    seen_maximums = set()

    for _ in range(3):
        current_maximum = maximum_ignoring_seen_maximums(nums, seen_maximums)
        if current_maximum == None:
            return max(seen_maximums)
        seen_maximums.add(current_maximum)

    return min(seen_maximums)
```

Complexity Analysis

- Time Complexity : $O(n)$.

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

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

- Space Complexity : $O(1)$.

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

Approach 3: Keep Track of 3 Maximums Using a Set

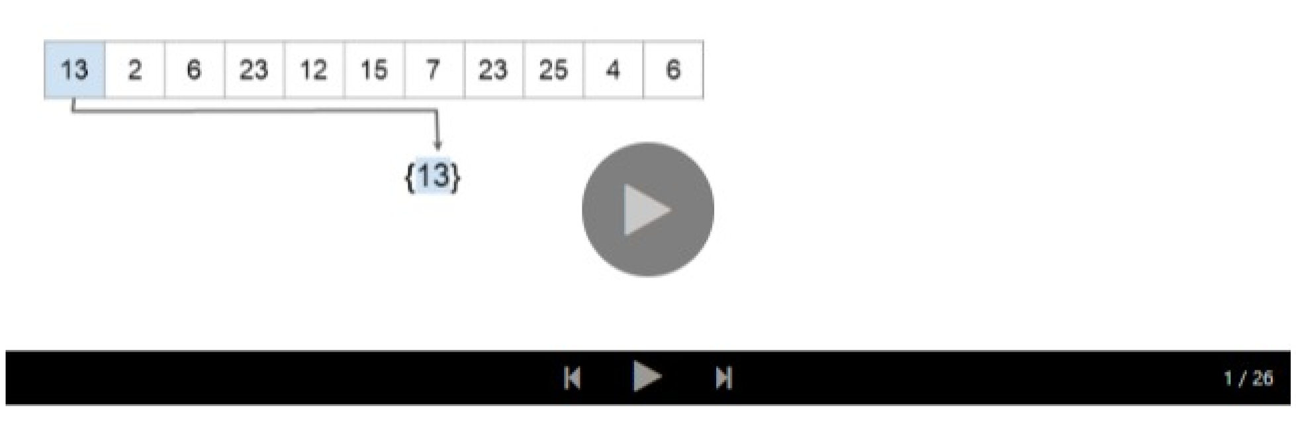
Intuition

So far, our approaches have required multiple parses through the input array. While this is still $O(n)$ 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.



Algorithm

Java | Python | Copy

```
def thirdMax(self, nums: List[int]) -> int:
    maximums = set()
    for num in nums:
        maximums.add(num)
        if len(maximums) > 3:
            maximums.remove(min(maximums))
    if len(maximums) == 3:
        return min(maximums)
    return max(maximums)
```

Complexity Analysis

- Time Complexity : $O(n)$.

  For each of the n values in the input `Array`, we insert it into a `Set` for a cost of $O(1)$. We then sometimes find and remove the minimum of the `Set`. Because there are never more than 3 items in the `Set`, the time complexity of doing this is $O(1)$.

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

- Space Complexity : $O(1)$.

  Because `maximums` never holds more than 3 items at a time, it is considered to be constant $O(1)$.

Related Problem - Kth Largest Element in an Array

A related problem, which would possibly be used as a follow up question in an interview is Kth Largest Element in an Array.

Rate this article:

Previous Next

Comments: 6 Sort By

Type comment here... (Markdown is supported)

Preview Post

happy_coder 21 March 15, 2020 8:59 AM

In approach 3 we can use TreeSet instead:

```
class Solution {
    public int thirdMax(int[] nums) {
        TreeSet<Integer> treeSet = new TreeSet<>();
```

Read More

4 Share Reply

SHOW 2 REPLIES

khallaghi 7 April 2, 2020 7:41 AM

You can also use a minimum heap to keep track of the three largest elements. This takes up O(1) additional space. Given that the heap is constant size, we can push and pop elements from it in O(1) time. So the total runtime is going to be O(n).

```
import heapq
```

Read More

3 Share Reply

SHOW 5 REPLIES

anjali_m 8 April 22, 2020 11:55 PM

min of a set is not O(1) time, a set is unordered, so it would take O(n) time..?

2 Share Reply

SHOW 1 REPLY

wbing520 0 July 6, 2020 7:04 AM

We could simply use three values to track the top 3 max values. O(n) time complexity and O(1) space.

```
class Solution {
    public int thirdMax(int[] nums) {
```

Read More

0 Share Reply

sriharik 170 May 25, 2020 3:45 AM

I'm surprised I didn't see keeping 3 max pointers as a solution. Of course this isn't maintainable if we are asked to find the K max element, but it still is a solution.

```
    public int thirdMax(int[] nums) {
        Set<Integer> set = new HashSet<>();
```

Read More

0 Share Reply

Dhruwat 5 April 9, 2020 12:08 AM

Use a min priority queue, keep its size <= 3, also keep track of maxNum. if priority queue.size() >= 3, return priority queue.peek(), else return maxNum. performance complexity o(n) + o(4log4) = o(n). Space complexity - o(n) + o(3) = o(n)

0 Share Reply