414. Third Maximum Number

Easy Array Sorting

Problem Description

The task is to look at a list of integers, `nums`, and from that list, identify the third highest distinct number. Distinct numbers are those that are not the same as any others in the list. In other words, if we were to sort all the unique numbers from highest to lowest, we want the number that would be in third place.

However, there's a catch. If it turns out that there aren't enough unique numbers to have a third place, the problem tells us to instead give back the highest number from the original list.

Intuition

The key challenge here is to do this efficiently and with care to ensure that we only consider unique elements from the array. A straightforward idea might be to first remove duplicates and then sort the array but this could be costly in terms of time especially if the array is large.

The solution ingeniously handles this by iterating over the list just once and keeping track of the three highest unique numbers seen so far, which are stored in variables `m1`, `m2`, and `m3`. This mirrors maintaining the first three places in a race where `m1` is the gold medal position, `m2` is the silver, and `m3` is the bronze.

During each iteration, the code checks if the current number is already in one of the medal positions. If it is, it's not distinct and can be ignored. If it's a new number and bigger than the current gold (`m1`), then `m1` takes the value of this number, and the previous `m1` and `m2` values move to second and third place respectively. A similar shift happens if the new number is smaller than `m1` but bigger than `m2` (`m2` takes the new value, and `m3` takes the old `m2`), and again for `m3`.

This way, by the end of the loop, we have the top three distinct values without any unnecessary computations. If `m3` is still set to the smallest value it could be (`-inf` here is used to symbolize the smallest possible value), then there weren't enough distinct numbers to have a third place, and so we return `m1`, the top value.

Solution Approach

The implementation of the solution uses a straightforward single-pass algorithm with constant space complexity—that is, it only needs a handful of variables to keep track of the top three numbers, rather than any additional data structures that depend on the size of the input array.

Here's a step-by-step walk-through of the algorithm:

1. Three variables, `m1`, `m2`, and `m3`, are initialized to hold the top three maximum numbers, but are initially set to negative infinity (`-inf` in Python), representing the lowest possible value. Why? Because we're trying to find the maximum, we can safely assume that any number from the input will be higher than `-inf`.
2. The algorithm iteratively examines each `num` in `nums`:
   - It first checks if `num` is already one of the three maximum values seen so far to avoid duplication. If it is, the algorithm simply continues to the next iteration with the `continue` statement.
   - If `num` is greater than the current maximum `m1`, it means a new highest value has been found. Consequently, `m1` is updated to `num`, and the previous `m1` and `m2` are moved down to `m2` and `m3`, respectively. This is somewhat like shifting the winners of the race down one podium spot to make room for the new champion at the top.
   - If `num` is not higher than `m1` but is higher than `m2`, the second place is updated to `num`, and the previous `m2` is pushed down to `m3`. The gold medalist (`m1`) remains unchanged.
   - If `num` falls below `m1` and `m2` but is higher than `m3`, then `m3` is updated to `num`. There is no effect on `m1` or `m2`.
3. After the loop concludes, the algorithm checks if the bronze position `m3` is still set to `-inf`. If it is, it means we never found a third distinct maximum number, so we return `m1`, the highest value found. If `m3` has been updated with a number from the array, we return it as it represents the third distinct maximum number.

This clever approach allows us to find the answer with a time complexity of O(n) where n is the number of elements in `nums` since we're only going over the elements once. There's no additional space needed that scales with the input size, so the space complexity is O(1).

Additionally, the algorithm benefits from short-circuiting; it skips unnecessary comparisons as soon as it has asserted that a number is not a potential candidate for the top three distinct values. This adds a little extra efficiency, as does the fact that there's no need to deal with sorting or dynamically sized data structures.

Example Walkthrough

Let's apply the solution approach to a small example to better understand how it works. Consider the following list of integers:

```
nums = [5, 2, 2, 4, 1, 5, 6]
```

We want to use the described algorithm to find the third highest distinct number in this list. As per the problem, if there are not enough unique numbers, we should return the highest number.

1. We initialize three variables `m1`, `m2`, and `m3` to `-inf`, representing the three highest unique numbers we're looking for.
2. We start iterating over `nums`:

   a. First element, `5`: - It is not in `m1`, `m2`, or `m3` (all are `-inf`), so `m1` becomes `5`. Now, `m1 = 5, m2 = -inf, m3 = -inf`.

   b. Second element, `2`: - Not already present in `m1`, `m2`, or `m3`, and it's less than `m1`, so `m2` becomes `2`. Now, `m1 = 5, m2 = 2, m3 = -inf`.

   c. Third and fourth elements, both `2`: - They match `m2` so we continue to the next iteration as these are not distinct numbers.

   d. Fifth element, `4`: - Not matched with `m1` or `m2` and is greater than `m2`, so `m2` gets pushed to `m3` and `m2` becomes `4`. Now, `m1 = 5, m2 = 4, m3 = 2`.

   e. Sixth element, again `5`: - Matches `m1` and is skipped as it is not distinct.

   f. Seventh element, `6`: - Greater than `m1`, so `m1` gets pushed to `m2`, `m2` gets pushed to `m3`, and `m1` becomes `6`. Final order is `m1 = 6, m2 = 5, m3 = 4`.
3. After iterating through all elements, we see that `m3` is `4`, which is not `-inf`, indicating that we found enough distinct numbers. Therefore, our answer is `m3`, which is `4`.

In summary, for the list `[5, 2, 2, 4, 1, 5, 6]`, the algorithm correctly identifies `4` as the third highest distinct number.

Solution Implementation

Python

```python
from math import inf

class Solution:
    def thirdMax(self, nums):
        # Initialize three variables to keep track of the max, second max,
        # and third max values, initialized as negative infinity
        max_value = second_max_value = third_max_value = -inf

        for num in nums:
            # Skip the number if it's equal to any of the three tracked values
            if num in {max_value, second_max_value, third_max_value}:
                continue

            # If the number is greater than the current max_value, shift the
            # values and set this number as the new max_value.
            if num > max_value:
                third_max_value, second_max_value, max_value = second_max_value, max_value, num

            # Else if the number is greater than the current second_max_value,
            # shift the second and third max values and set this number as the new second_max_value.
            elif num > second_max_value:
                third_max_value, second_max_value = second_max_value, num

            # Else if the number is greater than the current third_max_value, set this
            # number as the new third_max_value.
            elif num > third_max_value:
                third_max_value = num

        # Return third_max_value if it's not negative infinity; otherwise, return max_value.
        return third_max_value if third_max_value != -inf else max_value
```

Java

```java
class Solution {
    public int thirdMax(int[] nums) {
        // Initialize three variables to hold the first, second, and third maximum values
        // We use Long.MIN_VALUE to account for the possibility of all elements being negative
        // or to flag unassigned state since the problem's constraints allow for ints only.
        long firstMax = Long.MIN_VALUE;
        long secondMax = Long.MIN_VALUE;
        long thirdMax = Long.MIN_VALUE;

        // Loop through all the numbers in the array
        for (int num : nums) {
            // Continue if the number is already identified as one of the maxima
            if (num == firstMax || num == secondMax || num == thirdMax) {
                continue;
            }

            // Update the maxima if the current number is greater than the first maximum
            if (num > firstMax) {
                thirdMax = secondMax; // the old secondMax becomes the new thirdMax
                secondMax = firstMax; // the old firstMax becomes the new secondMax
                firstMax = num;       // assign the current number to firstMax
            }
            // Update the second and third maxima if the current number fits in between
            else if (num > secondMax) {
                thirdMax = secondMax; // the old secondMax becomes the new thirdMax
                secondMax = num;      // assign the current number to secondMax
            }
            // Update the third maximum if the current number is greater than thirdMax
            else if (num > thirdMax) {
                thirdMax = num;       // assign the current number to thirdMax
            }
        }

        // Return the third max if it's different from the initial value; otherwise, return the first max
        return (int) (thirdMax != Long.MIN_VALUE ? thirdMax : firstMax);
    }
}
```

C++

```cpp
class Solution {
public:
    int thirdMax(vector<int>& nums) {
        // Initialize the top three maximum values with the smallest possible long integers.
        long firstMax = LONG_MIN, secondMax = LONG_MIN, thirdMax = LONG_MIN;

        // Loop through all numbers in the vector.
        for (int num : nums) {
            // Skip the iteration if the current number matches any of the top three maxima.
            if (num == firstMax || num == secondMax || num == thirdMax) continue;

            // If the current number is greater than the first maximum,
            // shift the top maxima down and update the first maximum.
            if (num > firstMax) {
                thirdMax = secondMax;
                secondMax = firstMax;
                firstMax = num;
            // Else if the current number is greater than the second maximum,
            // shift the second and third maxima down and update the second maximum.
            } else if (num > secondMax) {
                thirdMax = secondMax;
                secondMax = num;
            // Else if the current number is greater than the third maximum,
            // update the third maximum.
            } else if (num > thirdMax) {
                thirdMax = num;
            }
        }
        // Return the third maximum if it's not LONG_MIN
        // (which would mean there are at least three distinct numbers).
        // Otherwise, return the first maximum (the overall maximum number).
        return (int) (thirdMax != LONG_MIN ? thirdMax : firstMax);
    }
};
```

TypeScript

```typescript
function thirdMax(nums: number[]): number {
    // Initialize the top three maximum values with the smallest possible number in JavaScript.
    let firstMax: number = Number.MIN_SAFE_INTEGER;
    let secondMax: number = Number.MIN_SAFE_INTEGER;
    let thirdMax: number = Number.MIN_SAFE_INTEGER;

    // Loop through all numbers in the array.
    for (let num of nums) {
        // Skip the iteration if the current number matches any of the top three maxima.
        if (num === firstMax || num === secondMax || num === thirdMax) continue;

        // If the current number is greater than the first maximum,
        // shift the top maxima down and update the first maximum.
        if (num > firstMax) {
            thirdMax = secondMax;
            secondMax = firstMax;
            firstMax = num;
        // Else if the current number is greater than the second maximum,
        // shift the second and third maxima values down and update the second maximum.
        } else if (num > secondMax) {
            thirdMax = secondMax;
            secondMax = num;
        // Else if the current number is greater than the third maximum,
        // update the third maximum.
        } else if (num > thirdMax) {
            thirdMax = num;
        }
    }
    // Return the third maximum if it's greater than the smallest possible number
    // (which would mean there are at least three distinct numbers).
    // Otherwise, return the first maximum (the overall maximum number).
    return thirdMax > Number.MIN_SAFE_INTEGER ? thirdMax : firstMax;
}
```

```python
from math import inf

class Solution:
    def thirdMax(self, nums):
        # Initialize three variables to keep track of the max, second max,
        # and third max values, initialized as negative infinity
        max_value = second_max_value = third_max_value = -inf

        for num in nums:
            # Skip the number if it's equal to any of the three tracked values
            if num in {max_value, second_max_value, third_max_value}:
                continue

            # If the number is greater than the current max_value, shift the
            # values and set this number as the new max_value.
            if num > max_value:
                third_max_value, second_max_value, max_value = second_max_value, max_value, num

            # Else if the number is greater than the current second_max_value,
            # shift the second and third max values and set this number as the new second_max_value.
            elif num > second_max_value:
                third_max_value, second_max_value = second_max_value, num

            # Else if the number is greater than the current third_max_value, set this
            # number as the new third_max_value.
            elif num > third_max_value:
                third_max_value = num

        # Return third_max_value if it's not negative infinity; otherwise, return max_value.
        return third_max_value if third_max_value != -inf else max_value
```

Time and Space Complexity

Time Complexity

The provided code only uses a single for-loop over the length of `nums`, and within the for-loop, it performs constant-time checks and assignments. The operations inside the loop include checking if a number is already one of the maxima (`m1`, `m2`, `m3`), and updating these variables accordingly. These operations are O(1) since they take a constant amount of time regardless of the input size. Therefore, the time complexity of the code is directly proportional to the number of elements in the `nums` list, which is `O(n)`.

Space Complexity

The extra space used by the algorithm is constant: three variables `m1`, `m2`, `m3` to keep track of the three maximum values, and a few temporary variables during value swapping. No additional space that scales with input size is used, so the space complexity is `O(1)`.