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 onto the right 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:

```
1  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.

Python Solution

```python
1  from math import inf
2
3  class Solution:
4      def thirdMax(self, nums):
5          # Initialize three variables to keep track of the max, second max,
6          # and third max values, initialized as negative infinity
7          max_value = second_max_value = third_max_value = -inf
8
9          for num in nums:
10             # Skip the number if it's equal to any of the three tracked values
11             if num in (max_value, second_max_value, third_max_value):
12                 continue
13
14             # If the number is greater than the current max_value, shift the
15             # values and set this number as the new max_value.
16             if num > max_value:
17                 third_max_value, second_max_value, max_value = second_max_value, max_value, num
18
19             # Else if the number is greater than the current second_max_value,
20             # shift the second and third max values and set this number as the new second_max_value.
21             elif num > second_max_value:
22                 third_max_value, second_max_value = second_max_value, num
23
24             # Else if the number is greater than the current third_max_value, set this
25             # number as the new third_max_value.
26             elif num > third_max_value:
27                 third_max_value = num
28
29         # Return third_max_value if it's not negative infinity, otherwise, return max_value.
30         return third_max_value if third_max_value != -inf else max_value
```

Java Solution

```java
1  class Solution {
2      public int thirdMax(int[] nums) {
3          // Initialize three variables to hold the first, second, and third maximum values
4          // We use Long.MIN_VALUE to account for the possibility of all elements being negative
5          // or to flag unassigned state since the problem's constraints allow for ints only.
6          long firstMax = Long.MIN_VALUE;
7          long secondMax = Long.MIN_VALUE;
8          long thirdMax = Long.MIN_VALUE;
9
10         // Loop through all the numbers in the array
11         for (int num : nums) {
12             // Continue if the number is already identified as one of the maxima
13             if (num == firstMax || num == secondMax || num == thirdMax) {
14                 continue;
15             }
16
17             // Update the maxima if the current number is greater than the first maximum
18             if (num > firstMax) {
19                 thirdMax = secondMax; // the old secondMax becomes the new thirdMax
20                 secondMax = firstMax; // the old firstMax becomes the new secondMax
21                 firstMax = num;       // assign the current number to firstMax
22             }
23             // Update the second and third maxima if the current number fits in between
24             else if (num > secondMax) {
25                 thirdMax = secondMax; // the old secondMax becomes the new thirdMax
26                 secondMax = num;      // assign the current number to secondMax
27             }
28             // Update the third maximum if the current number is greater than thirdMax
29             else if (num > thirdMax) {
30                 thirdMax = num;       // assign the current number to thirdMax
31             }
32         }
33
34         // Return the first max if it's different from the initial value; otherwise, return the first max
35         return (int) (thirdMax != Long.MIN_VALUE ? thirdMax : firstMax);
36     }
37  }
38
```

C++ Solution

```cpp
1  class Solution {
2  public:
3      int thirdMax(vector<int>& nums) {
4          // Initialize the top three maximum values with the smallest possible long integers.
5          long firstMax = LONG_MIN, secondMax = LONG_MIN, thirdMax = LONG_MIN;
6
7          // Loop through all numbers in the vector.
8          for (int num : nums) {
9              // Skip the iteration if the current number matches any of the top three maxima.
10             if (num == firstMax || num == secondMax || num == thirdMax) continue;
11
12             // If the current number is greater than the first maximum,
13             // shift the top maxima down and update the first maximum.
14             if (num > firstMax) {
15                 thirdMax = secondMax;
16                 secondMax = firstMax;
17                 firstMax = num;
18             // Else if the current number is greater than the second maximum,
19             // shift the second and third maxima down and update the second maximum.
20             } else if (num > secondMax) {
21                 thirdMax = secondMax;
22                 secondMax = num;
23             // Else if the current number is greater than the third maximum,
24             // update the third maximum.
25             } else if (num > thirdMax) {
26                 thirdMax = num;
27             }
28         }
29
30         // Return the third maximum if it's not LONG_MIN
31         // (which would mean there are at least three distinct numbers).
32         // Otherwise, return the first maximum (the overall maximum number).
33         return (int) (thirdMax != LONG_MIN ? thirdMax : firstMax);
34     }
35  };
```

Typescript Solution

```typescript
1  function thirdMax(nums: number[]): number {
2      // Initialize the top three maximum values with the smallest possible number in JavaScript.
3      let firstMax: number = Number.MIN_SAFE_INTEGER;
4      let secondMax: number = Number.MIN_SAFE_INTEGER;
5      let thirdMax: number = Number.MIN_SAFE_INTEGER;
6
7      // Loop through all numbers in the array.
8      for (let num of nums) {
9          // Skip the iteration if the current number matches any of the top three maxima.
10         if (num === firstMax || num === secondMax || num === thirdMax) continue;
11
12         // If the current number is greater than the first maximum,
13         // shift the top maxima values down and update the first maximum.
14         if (num > firstMax) {
15             thirdMax = secondMax;
16             secondMax = firstMax;
17             firstMax = num;
18         // Else if the current number is greater than the second maximum,
19         // shift the second and third maxima values down and update the second maximum.
20         } else if (num > secondMax) {
21             thirdMax = secondMax;
22             secondMax = num;
23         // Else if the current number is greater than the third maximum,
24         // update the third maximum.
25         } else if (num > thirdMax) {
26             thirdMax = num;
27         }
28     }
29
30     // Return the third maximum if it's greater than the smallest possible number
31     // (which would mean there are at least three distinct numbers).
32     // Otherwise, return the first maximum (the overall maximum number).
33     return thirdMax > Number.MIN_SAFE_INTEGER ? thirdMax : firstMax;
34  }
```

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).