2239. Find Closest Number to Zero

Easy Array

Leetcode Link

Problem Description

In this problem, you are provided with an integer array called `nums`, which has `n` number of elements. Your task is to find a number in `nums` that is closest to zero. If there is more than one such number, you will have to return the one with the largest value. Specifically, the closeness to zero is determined by the absolute value of the numbers, where the absolute value is the distance a number is from zero on the number line, without considering the direction (positive or negative).

Intuition

Approaching this problem, we should consider two key observations:

1. We can determine how close a number is to zero by looking at its absolute value. The smaller the absolute value, the closer the number is to zero.
2. In case of a tie—where two numbers are equally close to zero—we should return the larger number.

Given these observations, the solution involves iterating through each number in the array and tracking the one that is closest to zero. To keep track of the number closest to zero (let's call it `ans`), we also need to keep track of its absolute value (let's call it `d`, which stands for distance).

During each iteration, we check if the current number (`x`) has a smaller absolute value (`y := abs(x)`) than the smallest absolute we have seen so far (`d`). If it does, we update `ans` with `x` and `d` with `y`. In the case where `y` is equal to `d`, we perform an additional check: if `x` is greater than `ans`, we update `ans` to `x` because, as per the problem's requirement, we need to return the larger value in the event of a tie.

The solution initializes `ans` to zero and `d` with positive infinity (`inf`), which effectively means any number encountered first will replace these initial values. By iteratively updating our answers, once we go through the whole array, `ans` will hold the number closest to zero or the largest number in case of a tie.

Solution Approach

The implementation uses a simple linear scan algorithm that iterates through all the elements in the given integer array `nums`. It does not rely on any complex data structures and only requires a couple of variables to keep track of the state as it processes the array. The pattern used is straightforward and only requires basic conditional logic.

Here's a breakdown of the solution approach:

1. Initialize `ans` to `0` and `d` to positive infinity (`inf`). These variables are used to store the closest number to zero (`ans`) and its absolute value (`d`), respectively.
2. Iterate over each number `x` in the array `nums`.
3. In each iteration, calculate the absolute value of the current number and store it in a temporary variable `y` using the expression `y := abs(x)`.
4. Compare the absolute value `y` with the current minimum distance `d`.
5. If `y` is less than `d`, it means the current number `x` is closer to zero than any previous number we've encountered. So, update `ans` to `x` and `d` to `y`.
6. If `y` is equal to `d`, it means there is a tie. In this case, check if the current number `x` is greater than `ans`. If `x` is greater, it means we have found a larger number that is equally close to zero, so update `ans` to `x`. This step ensures that in the event of a tie, the larger number is chosen.
7. Continue this process until the loop has finished iterating through all the elements.
8. Return the value of `ans` as it now contains the number closest to zero (or the largest number in case of a tie).

The simplicity of the algorithm makes it efficient—it runs in O(n) time, where n is the length of the array since it requires only one pass through the array. It has O(1) space complexity as it only uses a fixed amount of extra space regardless of the input size.

Example Walkthrough

Suppose we have the following array `nums`: `[3, -7, 2, 5, -2, 4]`. Let's go through the solution step by step:

1. We begin by initializing our answer `ans` to `0` and the minimum distance `d` to positive infinity.
2. Starting with the first number in our array, `3`, we calculate its absolute value which is also `3`. Now, we compare this with `d`. Since `3` is less than positive infinity, we update `ans` to `3` and `d` to `3`.
3. We then move to the next number, which is `-7`. Its absolute value is `7`. This is greater than our current minimum distance `d` of `3`, so we do nothing.
4. The next number is `2`. Its absolute value is smaller than our current `d`. So we update `ans` to `2` and `d` to `2`.
5. We now consider `5`. Its absolute value is greater than `d`, so, again, we do nothing.
6. Next is `-2`. Its absolute value is the same as our current `d`. However, `-2` is not greater than our current `ans` of `2`, so we do not update `ans`.
7. Finally, we consider `4`. Its absolute value is greater than `d`, so there is no change to `ans` or `d`.

After scanning through all the elements in the array, we find that the number in `nums` closest to zero is `2`, and that's what we return.

Here, step 6 is particularly important to note; even though `-2` is as close to zero as `2`, the problem statement asks us to prioritize the largest number in case of ties, which has been correctly maintained as `2` in `ans`.

Using the algorithm outlined, we have successfully found and would return the closest number to zero from the array.

Python Solution

```python
from typing import List

class Solution:
    def findClosestNumber(self, nums: List[int]) -> int:
        # Initialize the answer and the smallest absolute difference.
        closest_number = 0
        smallest_diff = float('inf')

        # Iterate through each number in the list.
        for num in nums:
            # Calculate the absolute difference of the current number.
            current_diff = abs(num)

            # If the absolute difference is smaller than the smallest difference found so far,
            # or if the absolute difference is equal but the number is greater (closer to zero),
            # update the answer and the smallest difference.
            if current_diff < smallest_diff or (current_diff == smallest_diff and num > closest_number):
                closest_number = num
                smallest_diff = current_diff

        # Return the number that is closest to zero.
        return closest_number

```

Java Solution

```java
class Solution {
    // Function to find the number closest to zero
    public int findClosestNumber(int[] nums) {
        int closestNumber = 0; // Stores the closest number to zero found so far
        int minDistance = Integer.MAX_VALUE; // Initialize the minimum distance to the largest value possible

        // Loop through each number in the array
        for (int number : nums) {
            // Calculate the absolute value of the current number
            int absValue = Math.abs(number);

            // Check if the absolute value is less than the currently found minimum distance
            // Or if it is equal and the number is greater than the closest number found
            if (absValue < minDistance || (absValue == minDistance && number > closestNumber)) {
                closestNumber = number; // The current number is now the closest to zero
                minDistance = absValue; // Update the minimum distance
            }
        }

        // Return the number closest to zero found in the array
        return closestNumber;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <climits> // For using INT_MAX

class Solution {
public:
    // Function to find the closest number to zero in the given vector.
    // In case of a tie, returns the number that is greater (more positive).
    int findClosestNumber(vector<int>& nums) {
        int closestNumber = 0; // This will hold the number closest to zero
        int minDistance = INT_MAX; // This will hold the smallest distance from zero

        // Iterate through each number in the vector
        for (int number : nums) {
            int distance = abs(number); // Find the absolute value to get the distance from zero

            // If the current number is closer to zero or it is the positive number in case of a tie
            if (distance < minDistance || (distance == minDistance && number > closestNumber)) {
                closestNumber = number; // Update the closest number
                minDistance = distance; // Update the minimum distance
            }
        }

        // After the loop, return the number that is closest to zero
        return closestNumber;
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the closest number to zero in the array. If there are two numbers with
 * the same distance from zero, the positive one will be prioritized.
 *
 * @param {number[]} numbers The array of numbers to search through.
 * @return {number} The number closest to zero.
 */
function findClosestNumber(numbers: number[]): number {
    // Initialize answer and smallest difference `delta`
    // with a large number for starting comparisons.
    let [closestNumber, smallestDelta] = [0, Number.MAX_SAFE_INTEGER];

    // Iterate through each number in the array.
    for (const num of numbers) {
        // Calculate the absolute value of the current number.
        const currentDelta = Math.abs(num);

        // Check if the current number is closer to zero than the previous closest number,
        // or if it's equally close to zero but positive.
        if (currentDelta < smallestDelta || (currentDelta === smallestDelta && num > closestNumber)) {
            // Update closest number and smallest difference.
            [closestNumber, smallestDelta] = [num, currentDelta];
        }
    }

    // Return the number closest to zero from the list.
    return closestNumber;
}

```

Time and Space Complexity

The code provided consists of a single loop that iterates over each element in the list `nums`. Inside this loop, we perform a constant number of operations: calculating the absolute value of the current element `x`, comparing it to the minimum distance found so far `d`, and possibly updating the answer `ans` and the distance `d`. These operations are constant time operations that don't depend on the size of the input list.

Thus, the time complexity of this loop is `O(n)`, where `n` is the number of elements in the input list `nums`, since we have to look at each number exactly once to determine the answer.

In terms of space complexity, the code uses a fixed number of variables (`ans` and `d`) and does not utilize any additional data structures that scale with the size of the input. As a result, the space complexity is `O(1)`, which means that it requires a constant amount of additional memory regardless of the size of the input list.