2165. Smallest Value of the Rearranged Number

Medium Math Sorting

Leetcode Link

Problem Description

You are given an integer `num`. The task is to rearrange the digits of `num` in such a way that the resulting number is the smallest possible value and doesn't have any leading zeros. Importantly, you must maintain the sign of the original number; if the input number is negative, the result should also be negative, and the same for a positive number. The requirement is to return this minimum possible rearranged number.

Intuition

To solve this problem, we can utilize a counting sort-like approach since we're dealing with individual digits which are only from 0 to 9. Here's the intuition broken down into steps:

1. **Handling Zero:** If the number is 0, return it directly since 0 cannot be rearranged to get a smaller number.
2. **Digit Counting:** Count how many times each digit from 0 to 9 appears in the number using a list where the index corresponds to the digit and the value at that index represents the count.
3. **Handle Negative Numbers:** If the number is negative, to get the smallest possible number after rearrangement, we should start the number with the largest digit available and proceed to the smallest. Since leading zeros are not allowed, we only arrange the nonzero digits.
4. **Handle Positive Numbers:** a. If there are any zeros (as recorded in the digit counting step), make sure to place a nonzero digit at the beginning to avoid leading zeros. This is achieved by looking for the first nonzero count from 1 to 9, adding it to the answer string, and decreasing its count. b. After placing the first nonzero digit, append the remaining digits starting from 0 to 9 to ensure a minimal value.
5. **Returning the Result:** Combine the arranged digits into a single integer, preserving the sign, and return this integer as the answer.

This approach effectively sorts the digits to form the smallest (or largest if negative) possible number without using any extra sorting function, and the counting steps ensure that no leading zeros will appear in the final arrangement.

Solution Approach

The solution follows a direct approach, implementing the intuition we described earlier. Here's how the code breaks down the problem and applies an algorithm to solve it:

1. **Initialization and Handling of Zero:**
   - If `num` is 0, we return 0 immediately as we cannot rearrange it for a smaller value.
   - Initialize a counter array `cnt` with 10 zeroes, which will serve to count the occurrences of each digit (0-9).
   - Record the sign of `num` in a boolean `neg` and use `abs(num)` to work with the absolute value, simplifying our logic for rearrangement.
2. **Counting Digits:**
   - A while loop is used to extract each digit from `num` using `divmod(num, 10)`, which provides quotient and remainder - the latter being the digit we're counting.
   - Increment the count of the extracted digit in the `cnt` array.
3. **Constructing the Smallest/Largest Number Based on Sign:**
   - Initialize an empty string `ans` to build the final digit string.
   - If the number is negative (`neg` is `True`), iterate from 9 down to 0:
     - For each digit `i`, if the count `cnt[i]` is non-zero, append the digit `i` converted to a string, multiplied by its count, to `ans`. This assembles the largest possible number from the digits.
   - If the number is positive (`neg` is `False`):
     - First, find and place a non-zero digit if there are any zeros to avoid leading zeros. Iterate from 1 to 9 and on finding a non-zero count, append that digit to `ans` and decrement its count.
     - Next, append the rest of the digits in ascending order to build the smallest possible number. Iterate from 0 to 9 and for each digit `i`, if the count `cnt[i]` is non-zero, append the digit multiplied by its count to `ans`.
4. **Finalizing the Answer:**
   - Since `ans` is a string, it is converted back to an integer. The sign is reintroduced by negating the result if the original number was negative.

In summary, the code uses the counting sort technique to tally the occurrences of each digit and then rearranges the digits according to whether the number is positive or negative, to return the smallest possible value without leading zeros. It leverages simple array manipulation and string concatenation to avoid more complex sorting algorithms or additional data structures.

Example Walkthrough

Consider the integer `num = 310`. We need to find the smallest possible value by rearranging `num`'s digits without leading zeros.

1. **Handling Zero:**
   - `num` is not 0, so we proceed with the solution.
2. **Digit Counting:**
   - We count the occurrences of each digit:
     - 3 appears once.
     - 1 appears once.
     - 0 appears once.
   - The counter array `cnt` is thus `[1, 1, 0, 1, 0, 0, 0, 0, 0, 0]`.
3. **Handle Positive Numbers:**
   - Since `num` is positive, we follow the steps for positive numbers. a) We pick the smallest nonzero digit to avoid a leading zero. This is the digit `1`.
   - We place `1` at the beginning, and the `cnt` array updates to `[1, 0, 0, 1, 0, 0, 0, 0, 0, 0]`. b) Now we append the rest of the digits sorted in ascending order. The next smallest digit is `0`, followed by `3`.
   - We append these to get the number `103`.
4. **Returning the Result:**
   - Now, we combine the digits to form the integer `103`, which is the smallest variation of `310` with no leading zero.

On the other hand, consider the integer `num = -310`. For negative numbers, the smallest value is achieved by arranging digits in reverse order, from largest to smallest.

1. **Handling Zero:**
   - `num` is not 0, so we proceed with the solution.
2. **Digit Counting:**
   - The occurrences of each digit are the same as above, and the counter array `cnt` doesn't change.
3. **Handle Negative Numbers:**
   - Since `num` is negative, the largest digit, which is `3`, goes first, followed by `1`, then `0`.
   - We get the negative number `-310` since rearranging the digits in descending order gives the same number. We don't need to change anything for this specific example, as it is already in the required form.

Thus, by following the counting sort-like approach, we can efficiently determine the smallest possible rearrangement of `num` while preserving the original sign and avoiding any leading zeros.

Python Solution

```python
class Solution:
    def smallest_number(self, num: int) -> int:
        # If the number is zero, just return it
        if num == 0:
            return 0

        # Initialize a list to store the count of each digit
        digit_count = [0] * 10

        # Determine if the number is negative
        is_negative = num < 0

        # Work with the absolute value of the number
        num = abs(num)

        # Count the occurrences of each digit in the number
        while num:
            num, remainder = divmod(num, 10)
            digit_count[remainder] += 1

        # Create an empty string to build the answer
        answer_str = ""

        # If the number is negative, we create the largest number with its digits
        if is_negative:
            for i in range(9, -1, -1):  # Start from 9 to 0
                if digit_count[i]:
                    answer_str += str(i) * digit_count[i]
            # Convert the string to a negative integer
            return -int(answer_str)

        # If the number is positive and contains zeros,
        # place the smallest non-zero digit at the start
        if digit_count[0]:
            for i in range(1, 10):
                if digit_count[i]:
                    answer_str += str(i)
                    digit_count[i] -= 1
                    break

        # Append the remaining digits in ascending order
        for i in range(10):
            if digit_count[i]:
                answer_str += str(i) * digit_count[i]

        # Convert the string to an integer
        return int(answer_str)
```

Java Solution

```java
class Solution {

    // This method finds the smallest number that can be formed by rearranging the digits of the given number.
    public long smallestNumber(long num) {
        // If the number is 0, return it as the smallest number without any processing.
        if (num == 0) {
            return 0;
        }

        // An array to count the occurrences of each digit in the number.
        int[] digitCounts = new int[10];

        // Flag to check if the original number is negative.
        boolean isNegative = num < 0;

        // Take the absolute value in case the number is negative for ease of processing.
        num = Math.abs(num);

        // Count the occurrences of each digit in the number.
        while (num > 0) {
            digitCounts[(int) (num % 10)]++;
            num /= 10;
        }

        // This will hold the smallest number that we form by rearranging the digits.
        long result = 0;

        // If the original number is negative, we want the largest absolute value,
        // which when negated, gives the smallest possible negative number.
        if (isNegative) {
            // Construct the number from digits in descending order.
            for (int i = 9; i >= 0; --i) {
                while (digitCounts[i]-- > 0) {
                    result = result * 10 + i;
                }
            }
            // Return the negated result since the original number was negative.
            return -result;
        }

        // If number is positive and contains zeros, we must place the smallest non-zero digit at the
        // front to get the smallest possible number, then follow with any zeros.
        if (digitCounts[0] > 0) {
            for (int i = 1; i < 10; ++i) {
                if (digitCounts[i] > 0) {
                    result = result * 10 + i; // Place the smallest non-zero digit first.
                    digitCounts[i]--;
                    break; // Break after placing the first non-zero digit.
                }
            }
        }

        // After placing the smallest non-zero digit, place the remaining digits in ascending order.
        for (int i = 0; i < 10; ++i) {
            while (digitCounts[i]-- > 0) {
                result = result * 10 + i;
            }
        }

        // Return the final smallest number.
        return result;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cmath>

class Solution {
public:
    // Function to find the smallest permutation of the given number
    long long smallestNumber(long long num) {
        // Return 0 if the number is already 0
        if (num == 0) return 0;

        // Vector to count the occurrences of each digit
        std::vector<int> digitCount(10, 0);

        // Check if the number is negative
        bool isNegative = num < 0;

        // Work with the absolute value of num
        num = std::abs(num);

        // Count occurrences of each digit
        while (num) {
            digitCount[num % 10]++;
            num /= 10;
        }

        long long smallestNum = 0;

        // If the number is negative, build the largest number possible and then add the sign
        if (isNegative) {
            for (int i = 9; i >= 0; --i) {
                while (digitCount[i]--) {
                    smallestNum = smallestNum * 10 + i;
                }
            }
            // Return the negative value, as the original number was negative
            return -smallestNum;
        }

        // If the number has zeroes, we must handle them properly to avoid leading zeroes
        if (digitCount[0]) {
            // Find the smallest non-zero digit to place at the beginning
            for (int i = 1; i < 10; ++i) {
                if (digitCount[i]) {
                    smallestNum = smallestNum * 10 + i;
                    digitCount[i]--;
                    break;
                }
            }
        }

        // Build the smallest number by adding digits in ascending order
        for (int i = 0; i < 10; ++i) {
            while (digitCount[i]--) {
                smallestNum = smallestNum * 10 + i;
            }
        }

        return smallestNum;
    }
};
```

Typescript Solution

```typescript
// Import necessary functionalities from 'math' module
import { abs } from "math";

// Function to find the smallest permutation of the given number
function smallestNumber(num: number): number {
    // Return 0 if the number is already 0
    if (num === 0) return 0;

    // Array to count the occurrences of each digit
    let digitCounts: number[] = Array(10).fill(0);

    // Check if the number is negative
    let isNegative: boolean = num < 0;

    // Work with the absolute value of num
    num = abs(num);

    // Count occurrences of each digit
    while (num > 0) {
        digitCounts[num % 10]++;
        num = Math.floor(num / 10);
    }

    // Variable to hold the smallest (or largest if negative) permutation of the number
    let smallestNum: number = 0;

    // If the number is negative, build the largest number possible and then add the sign
    if (isNegative) {
        for (let i = 9; i >= 0; --i) {
            while (digitCounts[i]-- > 0) {
                smallestNum = smallestNum * 10 + i;
                digitCounts[i]--;
            }
        }
        // Return the negative value, as the original number was negative
        return -smallestNum;
    }

    // If the number has zeroes, we must handle them properly to avoid leading zeroes
    if (digitCounts[0] > 0) {
        // Find the smallest non-zero digit to place at the beginning
        for (let i = 1; i < 10; ++i) {
            if (digitCounts[i] > 0) {
                smallestNum = smallestNum * 10 + i;
                digitCounts[i]--;
                break;
            }
        }
    }

    // Build the smallest number by adding digits in ascending order
    for (let i = 0; i < 10; ++i) {
        while (digitCounts[i]-- > 0) {
            smallestNum = smallestNum * 10 + i;
        }
    }

    // Return the smallest permutation of the original number
    return smallestNum;
}
```

Time and Space Complexity

The time complexity of the provided solution is $O(n + 10)$ which simplifies to $O(n)$ where n represents the number of digits in the integer `num`. This is because we first iterate through all digits of the input `num` to count them (which contributes to $O(n)$), and then we iterate through the count array of 10 digits (which contributes to $O(10)$ but constants are omitted in Big O notation).

The space complexity of the solution is $O(10)$, which simplifies to $O(1)$ - constant space complexity. This is because the count array `cnt` has a fixed size of 10, regardless of the size of `num`. The string `ans` also doesn't count towards space complexity in this analysis as it simply corresponds to the size of the output, which is not considered additional space used by the algorithm.