869. Reordered Power of 2

Medium Math Counting Enumeration Sorting

Leetcode Link

Problem Description

The problem requires us to determine if it's possible to reorder the digits of a given integer `n` in such a way that the leading digit is not zero and the result is a power of two. We need to return `true` if we can achieve this and `false` otherwise. A power of two is any number in the form of `2^k`, where `k` is a non-negative integer.

Intuition

To solve this problem, we recognize that the order of the digits does not change whether a number is a power of two or not. Only the digits themselves and their counts are important. Therefore, if we can find a power of two that has exactly the same digits as the original number, then we have our solution. Here's how we arrive at the solution approach:

1. Count the frequency of each digit in the given number `n` using a function `convert(n)`. This function creates a count array, where each index represents a digit from 0 to 9, and the value at that index represents how many times that digit appears in `n`.
2. Generate powers of two and check against the converted count array of `n`. Starting from `1` (which is `2^0`), we generate subsequent powers of two by left-shifting (`i <<= 1` which is equivalent to multiplying by 2).
3. For each power of two, we convert it to the count array form and compare it to the count array of the original number `n`. If they match, it means we can reorder the digits of `n` to form this power of two, and we return `true`.
4. Continue this process until the power of two exceeds the largest possible integer formed by the digits of `n`, which is when the power of two surpasses `10^9`.
5. If none of the generated powers of two match the count array of `n`, we return `false`, as it is not possible to reorder the digits of `n` to form a power of two.

This solution ensures that we check all the relevant powers of two up to the size of the original number, without having to generate all permutations of the digits of `n`, which would be impractical for larger numbers.

Solution Approach

The solution implements a straightforward algorithm that leverages integer operations and simple data structures to determine if the input integer `n` can be reordered to form a power of two.

The algorithm can be broken down into two main functions:

1. `convert(n)`: This function takes an integer and converts it into a count array. Here is what the function does in detail:
   - Initialize a count array `cnt` with 10 elements, all set to `0`. Each element `cnt[i]` corresponds to the frequency of digit `i` in the input number.
   - Use a while loop to iterate through each digit of `n`. In each iteration, we perform the operation `n, v = divmod(n, 10)`, which extracts the last digit `v` of `n` and reduces `n` by one decimal place. The `divmod` function returns a pair of quotient and remainder from division.
   - Increment the count of the extracted digit `v` in the count array `cnt`.
   - Return the final count array, which represents the digit frequency of the original integer.
2. `reorderedPowerOf2(n)`: This function uses the `convert(n)` function to determine if the number `n` can be transformed into a power of two.
   - We start with `i = 1`, which is the smallest power of two (`2^0`), and a count array `s` which is the result of `convert(n)`.
   - In a while loop, we keep generating powers of two by left-shifting `i` (using `i <<= 1`), and for each `i`, we convert it to its count array form.
   - Compare the count array of the current power of two to the count array `s`, if they match (`convert(i) == s`), it means we can rearrange the digits of `n` to form this power of two, and we return `true`.
   - Continue the loop until `i` exceeds `10^9`, at which point we're sure that no power of two of this length exists that could match our original number's digits.
   - If we complete the loop without finding a match, we return `false`, indicating that `n` cannot be rearranged into a power of two.

This solution avoids generating permutations, which would be computationally exhausting, especially for larger integers. Instead, it cleverly checks through powers of two by incrementing in a controlled fashion and comparing digit frequencies, thus significantly reducing complexity while providing an accurate answer.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose the input integer `n` is `46`.

First, we need to convert the number `46` into its count array form using the `convert(n)` function:

1. Initialize the count array `cnt` with 10 elements all set to `0`.
2. Extract the digits of `46` iteratively and update their respective counts in `cnt`.
   - Extract the last digit `6`; `cnt[6]` becomes `1`.
   - Update the remaining value of `n` which is now `4`; `cnt[4]` becomes `1`.
3. The count array `cnt` for the number `46` is `[0, 0, 0, 0, 1, 0, 1, 0, 0, 0]`.

Now that we have the count array for `n`, we proceed with the `reorderedPowerOf2(n)` function:

1. Start with `i = 1`, and the count array `s` representing the number `46`, which is `[0, 0, 0, 0, 1, 0, 1, 0, 0, 0]`.
2. Generate powers of two by left-shifting `i` and convert each power of two into a count array to compare with `s`. Here are the first few iterations:
   - For `i = 1` (which is `2^0`), count array is `[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]`, not a match.
   - For `i = 2` (which is `2^1`), count array is `[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]`, not a match.
   - For `i = 4` (which is `2^2`), count array is `[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]`, not a match.
   - Continue incrementing `i` to find a power of two that matches the count array `[0, 0, 0, 0, 1, 0, 1, 0, 0, 0]`.
3. When `i = 64` (which is `2^6`), the count array is `[0, 0, 0, 0, 1, 0, 1, 0, 0, 0]`, which is a match for `s`.

Since we found a power of two with a count array that matches the count array `s` of our original number `46`, we can confidently return `true`. This demonstrates that the digits of `46` can be reordered to form the number `64`, which is a power of two.

Python Solution

```python
class Solution:
    def reorderedPowerOf2(self, n: int) -> bool:
        # Function to convert an integer to a count array of its digits
        def count_digits(num):
            count = [0] * 10  # Initialize count array for digits from 0 to 9
            while num:
                num, digit = divmod(num, 10)  # Split the number into its last digit and the rest
                count[digit] += 1  # Increment the corresponding digit count
            return count

        # Convert the input number to its digits count array
        input_digit_count = count_digits(n)

        # Start with the smallest power of 2 (2^0 = 1)
        power_of_two = 1

        # Loop through powers of 2 within the 32-bit integer range
        while power_of_two <= 10**9:
            # Compare the digit count array of the current power of 2 with the input number
            if count_digits(power_of_two) == input_digit_count:
                return True  # Found a power of 2 that can be reordered to the input number
            power_of_two <<= 1  # Go to the next power of 2 by left shifting bits

        # If no power of 2 matches after going through all 32-bit powers of 2
        return False
```

Java Solution

```java
class Solution {
    // Method to determine if the digits of the input number can be reordered to form a power of two
    public boolean reorderedPowerOf2(int n) {
        // Convert the number into a string representation of its digit count
        String targetDigitCount = convertToDigitCount(n);
        // Iterate through powers of 2
        for (int i = 1; i <= 1e9; i <<= 1) {
            // Compare the digit count of the current power of 2 with the input number's digit count
            if (targetDigitCount.equals(convertToDigitCount(i))) {
                // If both counts match, the digits can be reordered to form a power of two
                return true;
            }
        }
        // If no power of 2 matches, return false
        return false;
    }

    // Helper method to convert a number into a string representing the count of each digit
    private String convertToDigitCount(int n) {
        // Array to store the count of digits from 0 to 9
        char[] digitCount = new char[10];
        // Divide the number into digits and count them
        while (n > 0) {
            digitCount[n % 10]++;
            n /= 10;
        }
        // Return a new string constructed from the array of digit counts
        return new String(digitCount);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // This function checks if it's possible to reorder digits of 'n' to make it a power of two
    bool reorderedPowerOf2(int n) {
        // Convert the original number 'n' to its digits count array
        vector<int> originalDigitsCount = countDigits(n);
        // Loop through the powers of 2 within the int range
        for (int i = 1; i <= 1e9; i <<= 1) {
            // Compare the digits count of the current power of 2 with the original number
            if (originalDigitsCount == countDigits(i))
                return true;
        }
        return false;
    }

    // Helper function to count the digits of a number and return them as a vector
    vector<int> countDigits(int n) {
        vector<int> digitCount(10, 0);
        while (n != 0) {
            // Increase the count of the rightmost digit
            digitCount[n % 10]++;
            // Remove the rightmost digit
            n /= 10;
        }
        return digitCount;
    }
};
```

Typescript Solution

```typescript
// This function checks if it's possible to reorder digits of 'n' to make it a power of two
function reorderedPowerOf2(n: number): boolean {
    // Convert the original number 'n' to its digit's count array
    const originalDigitsCount: number[] = countDigits(n);
    // Loop through the powers of 2 within the range of 32-bit signed integer
    for (let i = 1; i <= 1e9; i <<= 1) {
        // Compare the digits count of the current power of 2 with the original number
        if (arraysEqual(originalDigitsCount, countDigits(i))) {
            return true;
        }
    }
    return false;
}

// Helper function to count the digits of a number and return them as an array
function countDigits(n: number): number[] {
    const digitCount: number[] = new Array(10).fill(0);
    while (n !== 0) {
        // Increase the count of the rightmost digit
        digitCount[n % 10]++;
        // Remove the rightmost digit
        n = Math.floor(n / 10);
    }
    return digitCount;
}

// Helper function to compare if two arrays have the same elements in the same order
function arraysEqual(a: number[], b: number[]): boolean {
    if (a.length !== b.length) return false;
    for (let i = 0; i < a.length; i++) {
        if (a[i] !== b[i]) return false;
    }
    return true;
}
```

Time and Space Complexity

The time complexity of the code is `O(log^2n)` where `n` is the input number. This is because there are `log n` (base 2) numbers that we compare with the input number's digit count after it's converted. The conversion itself takes `O(log n)` time since we iterate through each digit in the number (`log n` digits for a base 10 representation). The space complexity is `O(1)` because the count array has a fixed size irrespective of the input size.