1780. Check if Number is a Sum of Powers of Three

Medium Math

Problem Description

Given an integer `n`, the task is to determine if `n` can be represented as the sum of distinct powers of three. According to mathematical background, a number is a power of three if it can be written as `3^x`, where `x` is an integer. The problem statement requires us to find if this integer `n` can be expressed as the sum of distinct numbers, where each number is a unique power of three. For example, the number 13 can be represented as `3^2 + 3^0` (which is 9 + 1). The output should be `true` if such a representation exists or `false` otherwise.

Intuition

The intuition behind the solution stems from the properties of numbers expressed in base 3 (ternary). In base 3, every digit can only be 0, 1, or 2. If we can convert the number `n` into a base 3 representation without any digit exceeding 1, then it is clear that `n` can be represented as a sum of distinct powers of three, since each digit in the base 3 number would correspond to the coefficient of a power of three (0 for absent, 1 for present). If a digit is greater than 1, it means there's a repeated power of three, which violates the distinctness condition.

To solve this, we iterate through the number by dividing `n` by 3 in each iteration and checking the remainder. If at any point the remainder is greater than 1, it means that the current digit in the base 3 representation is greater than 1 (which indicates a repetition of a power of three) and we return `false`. Otherwise, if we successfully decompose the whole number without finding any digit greater than 1, it means we can represent `n` as the sum of distinct powers of three, hence a return value of `true`.

This process is akin to continually subtracting the largest possible power of three from `n` until it either becomes 0 (possible) or we encounter an infeasible step (remainder > 1), which makes it impossible.

Solution Approach

The implementation provided is straightforward as it relies on properties of numbers when expressed in base 3. The solution does not use any complex algorithms, additional data structures, or design patterns but follows a simple iterative process. Here's a step-by-step explanation:

1. The function `checkPowersOfThree` takes an integer `n` as its argument and enters a loop, which continues until `n` becomes 0.
2. Within the loop, the program checks the remainder of `n` when divided by 3 using `n % 3`. If the remainder is greater than 1, the function immediately returns `False`. This is because in base 3, no digit should be greater than 1 for the sum to be made up of distinct powers of three.
3. If the remainder is 0 or 1, it means the current digit in `n`'s ternary representation fits our requirements, so we continue to decompose `n`. We do this by performing integer division on `n` by 3 using `n //= 3`. This effectively shifts `n` one digit down in its ternary representation.
4. If the loop exits normally (without hitting a remainder greater than 1), it implies that `n` has been fully decomposed without issues, so the function returns `True`.

This is an efficient solution since at each step it reduces the problem's size by a factor of 3, leading to a logarithmic time complexity with respect to the size of the input number in base 3.

Example Walkthrough

Let's consider the integer `n = 31` as our example to illustrate the solution approach. We want to determine if `n` can be expressed as the sum of distinct powers of three.

We start by initializing `n = 31` and entering the loop where we will perform the following steps until `n` becomes 0:

1. Check the remainder of `n` when divided by 3, i.e., `n % 3`.
2. For `n = 31`, `n % 3` equals `1` (since `31 divided by 3` leaves a remainder of `1`), so we continue, as this is a valid digit in base 3.
3. Update `n` to be `n //= 3`, which translates to `n` being `31 // 3` or `10`.

We repeat the steps with the new value of `n = 10`:

1. The remainder when `10` is divided by `3` is `1` again, which is valid.
2. Update `n` again: `n //= 3`, now `n` becomes `10 // 3` or `3`.

Next, we consider `n = 3`:

1. The remainder when `3` is divided by `3` is `0`, which is also valid.
2. Update `n`: `n //= 3`, which makes `n` now `3 // 3` or `1`.

Finally, we consider `n = 1`:

1. The remainder when `1` is divided by `3` is `1`.
2. Update `n` to be `n //= 3`, and `n` becomes `1 // 3` or `0`.

The loop has exited normally, and `n` has been reduced to `0`. Since we have not encountered any remainder greater than `1` during the whole process, it means that the original integer `31` can indeed be expressed as the sum of distinct powers of three. Hence, the function would return `True`.

In base 3, `31` is actually `1*3^3 + 1*3^2 + 0*3^1 + 1*3^0`, which is a valid sum of distinct powers of three (27 + 9 + 0 + 1).

Solution Implementation

Python

```python
class Solution:
    def check powers of three(self, n: int) -> bool:
        # While 'n' is not zero
        while n:
            # If the remainder when 'n' is divided by 3 is greater than 1,
            # it means that 'n' includes a power of 3 with a coefficient greater than 1,
            # which is not allowed as we want only powers of 3 with a coefficient of 0 or 1
            if n % 3 > 1:
                return False  # 'n' is not a sum of powers of 3

            # Reduce 'n' by dividing by 3 to check the next smaller power of 3
            n //= 3

        # If we successfully reduce 'n' to zero by dividing by 3, then
        # 'n' is a sum of powers of 3 and we return True
        return True

# The method 'check powers of three' checks whether a given integer 'n'
# can be represented as the sum of unique powers of 3.
```

Java

```java
class Solution {
    public boolean checkPowersOfThree(int n) {
        // Loop until n is greater than 0
        while (n > 0) {
            // If the remainder of n divided by 3 is greater than 1,
            // it means n is not a sum of powers of three (since it either
            // has a factor of 3's power greater than 1,
            // or it includes a number that's not a power of 3).
            // So the function returns false.
            if (n % 3 > 1) {
                return false;
            }
            // Divide n by 3 to reduce the problem to a smaller instance
            // of the same problem, checking the next power of 3.
            n /= 3;
        }
        // After the loop, if n has been reduced to 0, it means n can fully be
        // represented as a sum of powers of three.
        // The function returns true in this case.
        return true;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if a given number can be represented as a sum of unique powers of three
    bool checkPowersOfThree(int n) {
        while (n > 0) {             // Loop until the number is reduced to 0
            int remainder = n % 3; // Find remainder when n is divided by 3

            if (remainder > 1) {
                // If remainder is greater than 1, it cannot be represented as a sum of unique powers of 3
                return false;
            }

            n /= 3; // Reduce n by dividing it by 3 for the next iteration
        }
        // If the loop completes, n can be represented as a sum of unique powers of three
        return true;
    }
};
```

TypeScript

```typescript
// Function to check if 'n' can be represented as a sum of distinct powers of three
function checkPowersOfThree(n: number): boolean {
    // Loop until 'n' is reduced to 0
    while (n > 0) {
        // If the remainder of 'n' divided by 3 is greater than 1, return false
        // as we can have only 0 or 1 as coefficients in powers of three representation
        if (n % 3 > 1) {
            return false;
        }
        // Divide 'n' by 3 and floor it to get the next number to check
        n = Math.floor(n / 3);
    }
    // If the loop completes without returning false, 'n' can be represented
    // as a sum of distinct powers of three, hence return true
    return true;
}
```


```python
class Solution:
    def check powers of three(self, n: int) -> bool:
        # While 'n' is not zero
        while n:
            # If the remainder when 'n' is divided by 3 is greater than 1,
            # it means that 'n' includes a power of 3 with a coefficient greater than 1,
            # which is not allowed as we want only powers of 3 with a coefficient of 0 or 1
            if n % 3 > 1:
                return False  # 'n' is not a sum of powers of 3

            # Reduce 'n' by dividing by 3 to check the next smaller power of 3
            n //= 3

        # If we successfully reduce 'n' to zero by dividing by 3, then
        # 'n' is a sum of powers of 3 and we return True
        return True

# The method 'check powers of three' checks whether a given integer 'n'
# can be represented as the sum of unique powers of 3.
```


Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(log_3(n))`. This is because the while loop iterates through the number dividing it by 3 in each step until `n` becomes 0. The number of steps is proportional to the power of 3 that most closely matches `n`.

Space Complexity

The space complexity is `O(1)`, meaning it is constant. The code only uses a fixed number of variables (`n` and the return value), and this usage does not scale with the input size.