342. Power of Four

Easy | Bit Manipulation | Recursion | Math

Leetcode Link

Problem Description

The problem is to identify whether a given integer `n` is a power of four. That is, we need to determine if there exists an integer `x` such that when 4 is raised to the power `x` (4^x), the result is `n`. This determination must result in a boolean value of either `true` if `n` is indeed a power of four or `false` if it is not.

Intuition

The intuition behind the solution to this problem comes from understanding the properties of numbers that are powers of four:

1. **Greater than zero**: A power of four must be a positive integer because raising four to any power yields a positive result.
2. **Unique bit pattern**: Powers of four in binary representation have a unique pattern. They have one and only one bit set (1), and the number of zeros following this bit is even. For example, 4 in binary is `100`, and 16 is `10000`.
3. **Checking single bit**: To check for the single bit pattern, a common technique is to use the bitwise AND operation with `n` and `n - 1`. If `n` is a power of two (which is a subset condition for being a power of four), then `n & (n - 1)` will be zero because there will be only one '1' in the binary representation of `n`.
4. **Differentiating power of four from power of two**: Simply being a power of two does not guarantee that a number is a power of four. To differentiate, we notice that the only '1' bit in a power of four is at an even position, counting from right (starting at 1). One way to check this is to use a bitmask. The bitmask `0xAAAAAAAA` contains '1's in the odd positions of a 32-bit number, which will never overlap with a power of four. Therefore, the condition `(n & 0xAAAAAAAA) == 0` must hold true for powers of four, ensuring the only '1' bit is in the correct position.

Putting these checks together gives us a concise way to determine if `n` is a power of four. Therefore, the solution uses a composite condition: `n > 0` (positive check) and `(n & (n - 1)) == 0` (single bit check) and `(n & 0xAAAAAAAA) == 0` (correct bit position check) to verify if `n` is a power of four.

Solution Approach

The solution provided uses bitwise operators to efficiently determine if a given integer `n` is a power of four. Here's the step-by-step implementation approach:

1. **Positive check**: We first check if `n` is greater than zero since a power of four cannot be zero or negative. This is simply checked using `n > 0`.
2. **Power of two check**: Next, we want to confirm if the number is a power of two. This property is required since all powers of four are also powers of two (but not all powers of two are powers of four). The bitwise operation `n & (n - 1)` is used to determine this. For numbers that are powers of two, there is precisely one '1' in their binary representation. Subtracting one from such a number changes the rightmost '1' to '0' and all the bits to the right of it to '1's. When we perform an AND operation between such a number and its predecessor (which has a '0' in that position and '1's after), the result will be zero. So `(n & (n - 1)) == 0` must be true for powers of two.
3. **Power of four check**: To ensure that `n` is not just a power of two but specifically a power of four, another condition must be verified. In binary terms, the single '1' bit of a power of four appears only in even positions (1-indexed). To test this, a bitmask `0xAAAAAAAA` is created, where all the odd positions have a '1' (when counting from right and starting from zero). If `n` is a power of four, ANDing it with this bitmask should yield zero, since its only '1' should be in an even position and hence not overlap with the '1's in the bitmask. Thus, we need `(n & 0xAAAAAAAA) == 0` to be true as well.

The combination of these three checks `(n > 0)`, `(n & (n - 1)) == 0`, and `(n & 0xAAAAAAAA) == 0`, used in a single return statement with logical ANDs, ensures that we only return `true` if all conditions are satisfied, confirming `n` is a power of four.

The efficiency of this approach comes from using bitwise operations, which are very fast and allow us to avoid expensive computations like loops or recursion that might be used in alternative solutions.

Example Walkthrough

Let's walk through the solution approach using the example where `n = 64`.

First, we want to verify if `64` is greater than zero. Since `64 > 0` is true, we pass the first check.

Next, to check if `64` is a power of two, we use the "power of two check" by computing `64 & (64 - 1)`. The binary representation of 64 is `1000000` and the binary for 63 is `111111`, so:

64 (in binary): `1000000` 63 (in binary): `0111111` ----------------AND Result : `0000000`

Since the result is zero, `64` is confirmed to be a power of two, satisfying the second condition.

Finally, we must verify if `64` is specifically a power of four. For this, we check if the only '1' bit in `64` is in an even position, using the "power of four check" with the bitmask `0xAAAAAAAA`. This bitmask in binary is `10101010101010101010101010101010`. Performing an AND operation between `64` and `0xAAAAAAAA`:

64 (in binary) : `00000000000000000000000001000000` 0xAAAAAAAA (in binary): `10101010101010101010101010101010` ----------------------------------AND Result : `00000000000000000000000001000000`

Since the result is zero, the '1' is at an even position (the sixth position from right, 1-indexed), fulfilling the last condition.

Because all three conditions pass:

- `n > 0`
- `(n & (n - 1)) == 0`
- `(n & 0xAAAAAAAA) == 0`

We conclude that `64` is indeed a power of four and the final evaluation will correctly return `true`.

Python Solution

```python
class Solution:
    def is_power_of_four(self, num):
        """
        Check if a number is a power of four.

        The method applies bitwise operations to determine if the number is a power of four.
        This is done in two steps:
        1. Check if the number is greater than zero.
        2. Confirm that the number is a power of two by checking whether it has only one bit set.
           This is done by ensuring that the number has no bits in common with its predecessor (num & (num - 1)) == 0.
        3. Check that the single bit is in the correct position for a power of four ((num & 0xAAAAAAAA) == 0).

        :param num: Integer, the number to check
        :return: boolean, True if num is a power of four, False otherwise
        """

        # 0xAAAAAAAA is a hexadecimal number with a pattern where all even bits (starting from the least significant bit) are set,
        # which are not set for powers of four. In other words, when the bit of the power-of-four number is at an even position,
        # it should not intersect with the 0xAAAAAAAA mask.

        # First condition: num should be positive.
        # Second condition: there should only be a single bit set in num.
        # Third condition: the bit must not be in a position that would intersect with 0xAAAAAAAA.
        return num > 0 and (num & (num - 1)) == 0 and (num & 0xAAAAAAAA) == 0

# This would be used as follows:
# sol = Solution()
# result = sol.is_power_of_four(16)
# print(result) # This would print True, since 16 is indeed a power of four (4^2)

```

Java Solution

```java
class Solution {

    /* Function to check if a given number is a power of four */
    public boolean isPowerOfFour(int num) {
        // First, check if 'num' is positive since powers of four are always positive.
        // Then, check if 'num' is a power of two by confirming that only one bit is set in its binary representation.
        // This is done by using the bitwise AND operation between 'num' and 'num - 1', which should equal zero.
        // Finally, check if the only set bit is in the correct position to represent a power of four.
        // This is done by using bitwise AND with the hex value 0xAAAAAAAA, which has set bits at the positions
        // corresponding to the powers of two but not four. If 'num' is a power of four, this result should be zero.

        return num > 0 && (num & (num - 1)) == 0 && (num & 0xAAAAAAAA) == 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    bool isPowerOfFour(int num) {
        // Check if num is greater than 0, because powers of 4 are positive numbers.
        // Also check if num is a power of 2 by ensuring it has only one bit set using (num & (num - 1)) == 0.
        // Finally check if the set bit is in the correct place for a power of 4 by checking that none
        // of the bits in the odd positions are set using (num & 0xAAAAAAAA) == 0.
        // 0xAAAAAAAA is a hexadecimal constant which has 1s in the odd positions and 0s in the even positions.

        return num > 0 && (num & (num - 1)) == 0 && (num & 0xAAAAAAAA) == 0;
    }
};

```

Typescript Solution

```typescript
function isPowerOfFour(n: number): boolean {
    // Check if 'n' is greater than 0 because powers of four must be positive.
    // The expression 'n & (n - 1)) == 0' checks if 'n' is a power of 2
    // because only powers of 2 have a single bit set in binary.
    // But since we are interested in powers of 4, we need another check.
    // The number must not have bits set in places that would indicate it's also a power of 2 but not a power of 4.
    // In binary, powers of 4 are of the form 1 followed by an even number of 0s.
    // The mask '0xaaaaaaaa' has 1s in the odd positions (starting from 1).
    // By &-ing it with 'n' and checking for 0, we are ensuring 'n' has 1s only in even positions - i.e., it's a power of 4.
    return n > 0 && (n & (n - 1)) === 0 && (n & 0xaaaaaaaa) === 0;
}

```

Time and Space Complexity

The time complexity of the function `isPowerOfFour` is `O(1)`, which means it runs in constant time. This is because it only involves a finite number of bitwise operations and comparisons, which do not depend on the size of the input number `n`.

The space complexity of the function `isPowerOfFour` is also `O(1)`, indicating that it uses a constant amount of space. The amount of memory used does not increase with the input size as it only requires space for a fixed number of variables to process the operations.