231. Power of Two

Easy | Bit Manipulation | Recursion | Math

Leetcode Link

Problem Description

The problem provides us with a single integer `n` and asks us to determine if this number is a power of two. To rephrase, the question is asking if there exists another integer `x` such that `2` raised to the power `x` equals `n` (mathematically, this can be represented as `n == 2^x`). We need to return `true` if `n` is indeed a power of two, and `false` otherwise.

Intuition

To determine whether a given number is a power of two, we can use a simple observation about binary representations of powers of two. A power of two in binary is represented as a `1` followed by zeros. For example, `2^2` is `4` in decimal and `100` in binary, and `2^3` is `8` in decimal and `1000` in binary.

We know that subtracting 1 from a power of two will result in a binary number where the set bit (1) of the power of two turns to 0, and all following bits turn to 1. For example, if `n` is `4` (`100` in binary), then `n - 1` is `3` (`011` in binary). Using bitwise AND operation (`&`) between `n` and `n - 1`, every corresponding pair of bits are compared; the result is `1` only if both bits are `1`. Since `n - 1` has 1s where `n` has 0s, their AND result will always be 0 if `n` is a power of two.

The condition `n > 0` ensures that we exclude non-positive numbers, as only positive integers can be powers of two. Combining these two conditions with the logical AND operator (the `and` keyword in Python), which only results in `true` when both expressions are true, provides us with the correct check for whether `n` is a power of two.

In conclusion, `n > 0 and (n & (n - 1)) == 0` succinctly checks whether `n` is a positive integer and a power of two by exploiting the characteristics of powers of two in binary representation.

Solution Approach

The implementation of the solution is quite straightforward and does not involve complex data structures or algorithms. Instead, it uses bitwise operations, which are a fundamental part of lower-level programming languages and concepts.

Here's a step-by-step breakdown of the implementation:

1. Check if `n` is greater than 0. This step is essential because we're only interested in positive integers since negative integers and zero cannot be powers of two.
2. Perform a bitwise AND operation between `n` and `n-1`. Here's how it works:
    - When `n` is a power of two, its binary representation has exactly one bit set to `1`, and all other bits set to `0`. Example: `4` in binary is `100`.
    - Subtracting 1 from `n` flips the least significant `1` to `0` and all the bits to the right of it to `1` (if any). Example: `3` is `011` in binary.
    - A bitwise AND operation between these two numbers `n & (n - 1)` results in zero because there are no positions with a `1` in both numbers. This is unique to powers of two.
3. The result of the bitwise AND operation is compared with 0 using the equality `==` operator.
4. The logical AND operator `and` is finally used to return `True` only if both the above checks pass, which means `n` is greater than 0, and the result of `n & (n - 1)` is equal to 0, verifying that `n` is indeed a power of two.

To summarize, the one-liner `n > 0 and (n & (n - 1)) == 0` elegantly combines the necessary checks using bitwise and logical operators to determine if an integer is a power of two. It's efficient since these operations are performed at the bit level and usually take constant time.

Example Walkthrough

Let's illustrate the solution approach using the example of `n = 8`.

1. First, we check if `n` is greater than 0. For `n = 8`, this is true since 8 is a positive number.
2. Next, we perform a bitwise AND operation between `n` and `n-1`. In binary:
    - `n` is `8` which is `1000` in binary.
    - `n-1` is `7` which is `0111` in binary.
    - Performing a bitwise AND operation: `1000 & 0111` gives `0000`.
3. The result of this bitwise AND operation is `0`.
4. Finally, we combine the checks: `n > 0` and `(n & (n - 1)) == 0`. In this case, both are `true`.

So, for `n = 8`, the conditions hold true. The result of `8 > 0 and (8 & (7)) == 0` is `true`, which means that the number 8 is indeed a power of two. This method works efficiently for any positive integer and is especially quick to compute using bitwise operations.

Python Solution

```python
class Solution:
    def is_power_of_two(self, n: int) -> bool:
        # Check if the number is greater than 0 and only one bit is set.
        # The expression (n & (n - 1)) will be 0 only for powers of two,
        # because there is only one bit set in a power of two, and (n - 1)
        # will have all the bits set before that single bit in n.
        return n > 0 and (n & (n - 1)) == 0

# The method is_power_of_two can be used as:
# result = Solution().is_power_of_two(16)
# print(result)  # Output: True, because 16 is a power of two

```

Java Solution

```java
class Solution {
    /**
     * Checks if a number is a power of two.
     *
     * A power of two has exactly one bit set in binary representation.
     * If we subtract 1 from a number which is a power of 2 (e.g., 8 is 1000 in binary),
     * we get a number which has all the bits set after the bit that was set in original number (e.g., 7 is 0111).
     * Taking an AND of n and n-1 will yield 0 if n is a power of two.
     *
     * @param n The number to be checked.
     * @return true if n is a power of two, false otherwise.
     */
    public boolean isPowerOfTwo(int n) {
        // Check if n is greater than 0 and if n AND (n-1) is 0
        return n > 0 && (n & (n - 1)) == 0;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if a number is a power of two
    bool isPowerOfTwo(int number) {
        // A power of two has only one bit set in binary representation.
        // Subtracting one from such a number flips all the bits after
        // the set bit (including the set bit). Performing an AND operation
        // between the number and number - 1 would then result in zero.
        // Additionally, the number must be positive to be a power of two.
        return number > 0 && (number & (number - 1)) == 0;
    }
};

```

Typescript Solution

```typescript
// This function checks if a given number is a power of two.
// It uses bitwise AND to determine if there is only one bit set in the binary representation.
// The number should be greater than zero since 0 is not a power of two.
// Parameters:
// n - The number to check if it is a power of two.
// Returns:
// A boolean indicating whether the number is a power of two (true) or not (false).
function isPowerOfTwo(n: number): boolean {
    // Check if 'n' is greater than 0 to avoid non-positive integers.
    // (n & (n - 1)) will be 0 for powers of two since these numbers have a single high bit.
    // For example, 4 in binary is 100, and 3 is 011. Bitwise AND is 000.
    return n > 0 && (n & (n - 1)) === 0;
}

```

Time and Space Complexity

The time complexity of the code is `O(1)` because the operation performed is a bitwise AND operation and a comparison, both of which take constant time regardless of the size of the integer `n`.

The space complexity of the code is also `O(1)` as it does not use any additional memory that grows with the input size. The function only uses a fixed amount of space to store the input and output variables.