693. Binary Number with Alternating Bits

Easy Bit Manipulation

Problem Description

In this problem, we are given a positive integer, and our task is to determine whether this number represents a binary sequence of alternating bits. In other words, we have to check if every pair of adjacent bits in the binary representation of the positive integer is different. For instance, the integer `5`, which in binary is `101`, has alternating bits, whereas the integer `7`, which in binary is `111`, does not.

Intuition

To solve this problem, the solution utilizes bitwise operations, which are perfect for manipulating binary numbers directly. Here is the intuition broken down into steps:

1. We perform an XOR operation on the number `n` with itself shifted one bit to the right, `n >> 1`. An XOR operation will give us a `1` only when the two bits being compared are different. With the shifting, every pair of adjacent bits will be compared. If `n` has alternating bits, then this operation should result in a binary number composed entirely of `1`s.
2. Now we need to verify that after the XOR operation, the number indeed consists of all `1`s. For a binary number composed entirely of `1`s (let's call it `m`), adding one to it, `m + 1`, will result in a binary number with a `1` followed by `0`s (because it carries over). For example, if `m` is `111` in binary, `m + 1` would be `1000`.
3. If we perform a binary AND operation between `m` and `m + 1`, it should give us `0` because there are no common 1 bits between, for example, `111` (`m`) and `1000` (`m + 1`). The code checks this with `(n & (n + 1)) == 0`.

Solution Approach

The given solution's implementation uses bitwise operations, which is a common and efficient way to solve problems that involve bit manipulation due to their low-level nature and high speed of execution.

Let's walk through the code:

```
class Solution:
    def hasAlternatingBits(self, n: int) -> bool:
        n ^= n >> 1
        return (n & (n + 1)) == 0
```

1. The statement `n ^= n >> 1` uses the XOR (`^=`) operation and the right shift operator (`>>`). The right shift operator moves each bit of `n` to the right by one position. The XOR operation then compares the original `n` with its shifted version; if `n` had alternating bits, this operation will yield a number with all `1`s because the different adjacent bits (when compared with XOR) return `1`.
2. After the XOR operation, the solution checks if the resulting number has all `1`s. To do this, it adds `1` to the number (`n + 1`) and then does a bitwise AND (`&`) with the original number (`n`). If the resulting number is `0`, that confirms that all the bits in `n` were `1`, due to the fact that a binary number with all `1`s plus 1 will result in a power of two, which in binary is represented as `1000...0` (a one followed by zeroes).
3. The expression `(n & (n + 1)) == 0` does the final check. If it evaluates to `True`, then the number `n` originally passed in had alternating bits. If not, the bits did not alternate.

This approach cleverly leverages the characteristics of binary arithmetic to solve the problem with just two operations, highlighting the efficiency and elegance of bitwise manipulation.

Example Walkthrough

Let's use the number `10` as a small example to illustrate the solution approach. The binary representation of `10` is `1010`.

Step 1: Perform an XOR between `n` and `n >> 1`

- Original `n` in binary: `1010`
- n >> 1 (shifted to right by one): `0101`
- XOR operation: `n ^ (n >> 1) = 1010 ^ 0101 = 1111`

After the XOR operation, the number `n` becomes `1111`, which is composed entirely of `1`s. This outcome is expected for a number with alternating bits.

Step 2: Add `1` to `n` and perform a bitwise AND between `n` and `n + 1`

- `n`: `1111`
- `n + 1`: `10000`
- AND operation: `n & (n + 1) = 1111 & 10000 = 0`

Step 3: Conclusion

Since `n & (n + 1)` equals `0`, we can conclude that the original number `10` does indeed have alternating bits. The bitwise operations have confirmed that every adjacent pair of bits in the binary representation of `10` was different. The solution works as intended for the example of `10`.

In the case of a number without alternating bits, these two steps would lead to a non-zero result when performing the AND operation. Therefore, the condition would not be met, and the method would correctly indicate that the number does not have alternating bits.

Solution Implementation

Python

```
class Solution:
    def has_alternating_bits(self, number: int) -> bool:
        # XOR the number with itself right-shifted by one bit.
        # This combines each pair of bits (starting from the least significant bit)
        # and turns them into 1 if they were different (10 or 01), and 0 if they were the same (00 or 11).
        number ^= number >> 1

        # After the XOR operation, a number that had alternating bits like ...010101
        # becomes ...111111. To check if this is the case, we can add 1 to the number
        # resulting in ...1000000 (if the number was truly all 1s).

        # Then we do an 'AND' operation with its original value. If the number had
        # all 1s after the XOR, this operation will result in 0 because of no overlapping bits.
        # For example, for a number with alternating bits:
        #   After XOR:  011111 (31 in decimal, for number 21 which is 10101 in binary)
        #   Number + 1: 100000 (32 in decimal)
        #   AND op:     000000 (0 in binary)
        # If the resulting number is 0, then the input number had alternating bits.
        return (number & (number + 1)) == 0
```

Java

```
class Solution {

    public boolean hasAlternatingBits(int number) {
        // XOR the number with itself shifted right by 1 bit.
        // This will convert the number into a binary representation of all 1's
        // if it has alternating bits (e.g., 5 in binary is 101, and 5 ^ (5 >> 1) = 7, which is 111 in binary).
        number ^= (number >> 1);

        // Check if the number after XOR (now in the form of all 1's if bits are alternating)
        // AND the number incremented by 1 is zero.
        // Incrementing will shift all 1's to a leading 1 followed by all 0's (e.g., 111 becomes 1000).
        // If the bits in the number were alternating, then (number & (number + 1)) must be 0.
        return (number & (number + 1)) == 0;
    }
}
```

C++

```
class Solution {
public:
    bool hasAlternatingBits(int n) {
        // Perform XOR between number and its one bit right shift.
        // This will convert a sequence like '1010' into '1111' if the bits alternate,
        // or into some other pattern that is not all 1's if they don't alternate.
        n ^= (n >> 1);

        // Add 1 to the transformed number to set the rightmost 0 bit to 1 (if any).
        // This will result in a number with all bits set to 1 only if n was already all 1's.
        long nPlusOne = static_cast<long>(n) + 1;

        // Perform bitwise AND between n and nPlusOne.
        // If n had all bits set to 1 before, then n & nPlusOne will be zero,
        // indicating that the original number had alternating bits.
        // If n had any 0s, this operation will produce a non-zero result.
        return (n & nPlusOne) == 0;
    }
};
```

TypeScript

```
function hasAlternatingBits(n: number): boolean {
    // Perform XOR between the number n and its one bit right-shifted self.
    // This operation will convert a sequence like '1010' into '1111' if the bits alternate,
    // or into some other pattern that contains '0's if they don't alternate.
    n ^= (n >> 1);

    // Cast n to a number representation that can safely hold a larger value (using bitwise OR with 0).
    // Then add 1 to the transformed number (from above) to set the least significant 0 bit to 1 (if any exist).
    // This will result in a number with all bits set to 1 only if n was already composed exclusively of 1's.
    let nPlusOne: number = (n | 0) + 1;

    // Perform a bitwise AND between n and nPlusOne.
    // If n was composed exclusively of 1's before adding one, then n & nPlusOne will be zero,
    // indicating that the original number indeed had alternating bits.
    // If n contained any 0's before adding one, this operation will produce a non-zero result.
    return (n & nPlusOne) === 0;
}


class Solution:
    def has_alternating_bits(self, number: int) -> bool:
        # XOR the number with itself right-shifted by one bit.
        # This combines each pair of bits (starting from the least significant bit)
        # and turns them into 1 if they were different (10 or 01), and 0 if they were the same (00 or 11).
        number ^= number >> 1

        # After the XOR operation, a number that had alternating bits like ...010101
        # becomes ...111111. To check if this is the case, we can add 1 to the number
        # resulting in ...1000000 (if the number was truly all 1s).

        # Then we do an 'AND' operation with its original value. If the number had
        # all 1s after the XOR, this operation will result in 0 because of no overlapping bits.
        # For example, for a number with alternating bits:
        #   After XOR:  011111 (31 in decimal, for number 21 which is 10101 in binary)
        #   Number + 1: 100000 (32 in decimal)
        #   AND op:     000000 (0 in binary)
        # If the resulting number is 0, then the input number had alternating bits.
        return (number & (number + 1)) == 0
```

Time and Space Complexity

The given code is a method to check whether a number has alternating bits or not. Let's analyze the time and space complexity:

Time Complexity

The code performs the following operations:

1. `n ^= n >> 1`: This is a bitwise XOR operation applied to the number and its right-shifted version by one bit. It runs in `O(1)` time since the operation is performed on a fixed number of bits that constitutes the number `n`.
2. `(n & (n + 1)) == 0`: This operation checks if the result from the first operation+1 is a power of two which guarantees alternating bits (since it would set all bits to zero). This also runs in `O(1)` time for the same reason as the bitwise XOR operation.

Since both operations are constant time operations, the **overall time complexity** of the function is `O(1)`.

Space Complexity

The space complexity refers to the amount of extra memory used aside from the input. In this case, the function uses a fixed amount of memory to store the results of the operations, regardless of the size of `n`.

There are no additional data structures used that grow with the size of the input. Therefore, the **space complexity** is `O(1)`.