2980. Check if Bitwise OR Has Trailing Zeros

Easy Bit Manipulation Array

Problem Description

This LeetCode problem requires us to look at an array of positive integers and determine whether we can select two or more elements from this array such that the binary representation of their bitwise OR (`|`) operation ends with at least one trailing zero. A trailing zero in this context means that the least significant bit (rightmost bit) in the binary representation is a 0. For instance, if we take the binary representation of 4 which is "100", it clearly has two trailing zeros. This contrasts with the number 5 whose binary representation is "101", with no trailing zeros because the last bit is a 1.

The essential challenge here is to check the given array and return `true` if such a selection is possible and return `false` otherwise.

Intuition

The intuition behind the solution can be drawn directly from the nature of the bitwise OR operation and the structure of binary numbers. When we perform a bitwise OR between two numbers, each bit in the result is a 1 if any of the corresponding bits of the operands is a 1. This implies that if we have a trailing zero in the result, both operands must have a zero in the corresponding least significant bit position.

Now, focusing on even and odd numbers, we know that in binary, an even number always ends with a 0, and an odd number always ends with a 1. Therefore, when we OR an even number with another even number, the result will also end with a 0, because both numbers have a 0 at the least significant bit. However, if we OR an odd number (ending in 1) with any other number, the result will always end with a 1, thus not having a trailing zero.

Solution Approach

The approach to the solution is straightforward, employing a simple counting strategy without the need for any complex data structures or algorithms. The algorithms or patterns used in this solution utilize basic arithmetic and bitwise operations that are fundamental to programming.

Here's a brief rundown of the steps involved in the solution:

1. Iterate through each number in the `nums` array.
2. For each number, check if it is even. This can be done by looking at the least significant bit of the number. If the last bit is a 0, the number is even, and if it is a 1, the number is odd. In terms of bitwise operations, this check is equivalent to `x & 1`, which will be `0` for even numbers and `1` for odd numbers.
3. Count the total number of even numbers. Since we want to inverse the result of `x & 1` to count even numbers, we use `x & 1 ^ 1`. The `^ 1` serves as a bitwise NOT operation for the least significant bit. This count can be achieved with the expression `sum(x & 1 ^ 1 for x in nums)` which adds up 1 for every even number and 0 for every odd number.
4. Check if the count of even numbers is at least 2. This corresponds to having at least two numbers with a trailing zero in their binary form.
5. Return `true` if there are two or more even numbers, and `false` otherwise.

The pythonic way showcased in the sum expression handles the iteration and counting elegantly in a single line. The use of bitwise AND (`&`) and XOR (`^`) operations provides an efficient way to check and count even numbers without resorting to conditional statements, while the neatness of list comprehension allows the entire operation to be composed concisely.

Therefore, the crux of our solution lies in the expression `sum(x & 1 ^ 1 for x in nums) >= 2` which sums up a transformed list where even numbers contribute a count of 1 and odd numbers contribute a count of 0, and the comparison checks if the sum is greater than or equal to 2. If so, it implies the presence of two or more even numbers, which enables us to satisfy the condition set by the problem.

In conclusion, the solution is simple yet effective, leveraging basic bitwise operations to perform an even-number count in the array, which aligns with the necessary condition to solve the original problem.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the array `nums = [3, 5, 6, 8]`.

We will follow the steps outlined in the solution approach to determine if we can select two or more elements whose bitwise OR ends with at least one trailing zero.

Step-by-step Process:

1. We start by iterating through the array `nums`. Our array elements are 3, 5, 6, and 8.
2. Now, we inspect each number to see if it is even. Remember, even numbers in binary end with a 0, and odd numbers end with a 1.
   - For `3` (binary `11`), `3 & 1` yields `1`, indicating it is odd.
   - For `5` (binary `101`), `5 & 1` yields `1`, indicating it is odd.
   - For `6` (binary `110`), `6 & 1` yields `0`, indicating it is even.
   - For `8` (binary `1000`), `8 & 1` yields `0`, indicating it is even.
3. We count the number of even numbers by adding up 1s for each even number we encounter and 0s for each odd number. In our example, we have two even numbers (6 and 8), so our count is 2.
4. We check if our even count is at least 2. In our case, it is exactly 2, which satisfies the condition.
5. Since we found at least two even numbers, we return `true`. This indicates that it's possible to select elements from the array (specifically, 6 and 8 in our case) such that their bitwise OR will have at least one trailing zero. The binary OR of 6 (`110`) and 8 (`1000`) is `1110`, which indeed has a trailing zero.

Thus, following the steps, we can see that our example input array `[3, 5, 6, 8]` allows us to select two numbers whose bitwise OR operation ends with a trailing zero. The approach yields a `true` result, confirming the solution.

Solution Implementation

Python

```python
from typing import List  # Importing List type from typing module for type hinting

class Solution:
    def hasTrailingZeros(self, nums: List[int]) -> bool:
        # Initialize a counter for the number of elements with trailing zeros
        trailing_zero_count = 0

        # Iterate over each number in the list
        for number in nums:
            # Check if the last bit is 0 (which means the number is even)
            if (number & 1) == 0:
                # Increment the counter for numbers with trailing zero
                trailing_zero_count += 1

            # If we already have at least two numbers with trailing zeros, return True
            if trailing_zero_count >= 2:
                return True

        # If the function hasn't returned yet, it means we have less than 2 numbers with trailing zeros
        return False

# This code assumes that having trailing zeros refers to numbers being even
# or in binary form, the least sionificant bit is 0).
# The function returns True if there are 2 or more even numbers in the input list.
```

Java

```java
class Solution {
    // Method to determine if there are at least two trailing zeros in the binary representation of the numbers in the array
    public boolean hasTrailingZeros(int[] nums) {
        int countTrailingZeros = 0; // Initialize counter for trailing zeros

        // Iterate through each number in the array
        for (int number : nums) {
            // Add 1 to the counter if the least significant bit is 0 (even number), achieved by bitwise AND and XOR operations
            countTrailingZeros += (number & 1 ^ 1);
        }

        // Check if there are at least two numbers with trailing zeros (even numbers)
        return countTrailingZeros >= 2;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to determine if there exists two or more trailing zeros in the binary representation of any number in the array.
    bool hasTrailingZeros(vector<int>& nums) {
        // Initialize a counter for trailing zeros.
        int countTrailingZeros = 0;

        // Iterate over each number in the array.
        for (int number : nums) {
            // Increment the counter if the least significant bit is not set (number is even).
            countTrailingZeros += ((number & 1) ^ 1);
        }

        // If there are two or more numbers with trailing zeros, return true.
        return countTrailingZeros >= 2;
    }
};
```

TypeScript

```typescript
/**
 * Check if the input array of numbers has at least two trailing zeros when
 * represented in binary form.
 *
 * @param {number[]} numbers - The array of numbers to check.
 * @return {boolean} - Returns true if there are at least two trailing zeros, otherwise false.
 */
function hasTrailingZeros(numbers: number[]): boolean {
    // Initialize a counter for the number of zeros
    let zeroCount = 0;

    // Loop through each number in the input array
    for (const number of numbers) {
        // For each number, we perform a bitwise AND with 1 (check if the number is even),
        // and then XOR with 1 to flip the result. The count is incremented for even
        // numbers (which have a trailing zero in binary).
        zeroCount += (number & 1) ^ 1;
    }

    // If the count of trailing zeros is at least two, return true
    return zeroCount >= 2;
}
```

```python
from typing import List  # Importing List type from typing module for type hinting

class Solution:
    def hasTrailingZeros(self, nums: List[int]) -> bool:
        # Initialize a counter for the number of elements with trailing zeros
        trailing_zero_count = 0

        # Iterate over each number in the list
        for number in nums:
            # Check if the last bit is 0 (which means the number is even)
            if (number & 1) == 0:
                # Increment the counter for numbers with trailing zero
                trailing_zero_count += 1

            # If we already have at least two numbers with trailing zeros, return True
            if trailing_zero_count >= 2:
                return True

        # If the function hasn't returned yet, it means we have less than 2 numbers with trailing zeros
        return False

# This code assumes that having trailing zeros refers to numbers being even
# or in binary form, the least sionificant bit is 0).
# The function returns True if there are 2 or more even numbers in the input list.
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `nums` array. This complexity arises because the function iterates exactly once over all elements in the array to calculate the sum of `(x & 1 ^ 1)` for each `x` in `nums`. No other loops or nested iterations are present, so the entire operation scales linearly with the size of the input.

Space Complexity

The space complexity of the given code is `O(1)` which signifies constant space. This is because the space required does not increase with the size of the input array. The only extra space used is for the accumulator in the `sum` function, which does not depend on `n`, and basic variables needed to iterate through the array and compute the bitwise operations.