2595. Number of Even and Odd Bits

Easy Bit Manipulation

Problem Description

This problem requires us to examine the binary representation of a given positive integer `n`. Specifically, we need to count the number of bits set to `1` that are located at even and odd indices separately in this representation. Indices are 0-indexed, meaning they start counting from zero, and the least significant bit is at index 0. The ultimate goal is to return an array with two numbers: the first is the count of `1`s at even indices, and the second is the count of `1`s at odd indices in the binary representation of `n`.

Intuition

Solution Approach

The solution is implemented using a simple while loop, an array to keep track of the counts of `1`s at even and odd indices, and a variable `i` to alternate between even and odd. The initial setup defines an array `ans` initialized to `[0, 0]`, representing the counts at even and odd indices respectively.

The `i` variable is initialized to `0`, which will help us toggle between even (0) and odd (1) positions. In every iteration of the while loop, the expression `n & 1` is evaluated, which uses the bitwise AND operation to check if the least significant bit of `n` is set to `1`. If it is, `ans[i]` is incremented, effectively counting the `1` for the current index.

We use the expression `i ^= 1` to toggle the value of `i` between `0` and `1`. This is a bitwise XOR operation which flips `i` from `0` to `1` or from `1` to `0`, aligning with the even and odd index we are currently at.

Then, `n >>= 1` shifts the binary representation of `n` to the right by one, essentially moving to the next bit in the binary representation of the number as the next least significant bit to check.

This loop continues until `n` becomes `0`, meaning all bits have been processed. At the end of the loop, `ans` contains the counts of `1`s at even and odd indices, and it is returned as the final result.

Using bitwise operations makes the algorithm efficient since it operates directly on the binary representation of the number. The space complexity is O(1) because we only use a fixed-size array and a few variables, and the time complexity is O(log n) due to the number of bits in `n`.

Example Walkthrough

Let's go through an example to illustrate how the solution approach works. Assume we have the positive integer `n = 11`. In binary, `11` is represented as `1011`.

Now, let's initialize our solution:

- `ans` will be `[0, 0]`, starting with zero counts for both even and odd indices.
- `i` will be initialized to `0` since we begin with index `0` (even).

We'll go through the number bit by bit, checking if each bit is a `1` and then increment the corresponding count in `ans`:

1. n = 11 (binary `1011`), n & 1 = 1 (odd, index 0), so increment `ans[0]` to `[1, 0]`.
2. Shift `n` right by 1 (n >>= 1), n = 5 (binary `101`), toggle `i` (i ^= 1), now `i = 1`.
3. n & 1 = 1 (even, index 1), so increment `ans[1]` to `[1, 1]`.
4. Shift `n` right by 1, n = 2 (binary `10`), toggle `i` (i ^= 1), now `i = 0`.
5. n & 1 = 0, so no increment, `ans` stays the same `[1, 1]`.
6. Shift `n` right by 1, n = 1 (binary `1`), toggle `i` (i ^= 1), now `i = 1`.
7. Finally, n & 1 = 1 (odd, index 3), increment `ans[1]` to `[1, 2]`.
8. Shift `n` right by 1, now n = 0. The loop ends.

Thus, in the binary representation of the number `11`, there is 1 bit set at even indices (index 0) and 2 bits set at odd indices (indices 1 and 3). Therefore, the final `ans` array is `[1, 2]`, which is the result we return.

Solution Implementation

Python

```python
class Solution:
    def even_odd_bit(self, n: int) -> List[int]:
        # Initialize an array with two elements, for even and odd bit counts.
        bit_counts = [0, 0]

        # Initialize the index to point at the even position.
        index = 0

        # Loop until all bits are processed.
        while n:
            # Add the least significant bit (LSB) to the appropriate count (even/odd).
            bit_counts[index] += n & 1

            # Toggle index: 0 becomes 1, 1 becomes 0. (Switch between even and odd).
            index ^= 1

            # Right shift 'n' to process the next bit.
            n >>= 1

        # Return the counts of even and odd position bits.
        return bit_counts
```

Java

```java
class Solution {

    /**
     * This method calculates the number of set bits (1s) present in the even and odd positions
     * of the binary representation of a given number.
     *
     * @param n The integer whose binary representation is to be analyzed.
     * @return An array of two integers where the first element is the count of set bits at
     *         even positions, and the second element is the count of set bits at odd positions.
     */
    public int[] evenOddBit(int n) {
        // Initializing an array to hold the count of set bits at even and odd positions
        int[] countOfSetBits = new int[2];

        // Iterating over each bit of the input number
        for (int i = 0; n > 0; n >>= 1, i ^= 1) {
            // Increment the count at index 'i' where 0 refers to even and 1 refers to odd positions
            // The current bit is determined by 'n & 1', adding to the respective count based on the position
            countOfSetBits[i] += n & 1; // If the current bit is 1, add to the count
        }

        // Returning the array with counts of set bits at even and odd positions
        return countOfSetBits;
    }
}
```

C++

```cpp
#include <vector> // Include the vector header for using the std::vector container.

class Solution {
public:
    // Function that returns the count of even and odd positioned bits set to '1' in the binary representation of n.
    std::vector<int> evenOddBit(int n) {
        std::vector<int> ans(2); // Create a vector with 2 elements initialized to 0 to store the counts.
        // ans[0] will hold the count of bits set in odd positions.
        // ans[1] will hold the count of bits set in even positions.

        // Loop over each bit of the integer 'n'.
        for (int i = 0; n > 0; n >>= 1, i ^= 1) { // i flips between 0 and 1 to represent even and odd positioning.
            ans[i] += n & 1; // Increment the count for even or odd index based on the current bit (0 or 1).
        }

        // Return the result vector containing the counts of set bits at even and odd positions.
        return ans;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the sum of bits at even and odd positions of a binary representation of a number.
 * @param n - The input number to evaluate.
 * @returns An array where the first element is the sum of bits at even positions, and the second is the sum at odd positions.
 */
function evenOddBit(n: number): number[] {
    // Initialize an array with two elements set to zero.
    // The first element is for sum of bits at even positions, and
    // the second is for sum of bits at odd positions.
    const bitSums = new Array(2).fill(0);

    // Initialize an index variable to toggle between 0 (even) and 1 (odd).
    let index = 0;

    // Iterate through each bit of the number.
    while (n > 0) {
        // Add the current bit to the corresponding position sum.
        bitSums[index] += n & 1;

        // Right shift the number to process the next bit.
        n >>= 1;

        // Toggle the index between 0 and 1.
        index ^= 1;
    }

    // Return the array containing sums of even and odd positioned bits.
    return bitSums;
}
```


```python
class Solution:
    def even_odd_bit(self, n: int) -> List[int]:
        # Initialize an array with two elements, for even and odd bit counts.
        bit_counts = [0, 0]

        # Initialize the index to point at the even position.
        index = 0

        # Loop until all bits are processed.
        while n:
            # Add the least significant bit (LSB) to the appropriate count (even/odd).
            bit_counts[index] += n & 1

            # Toggle index: 0 becomes 1, 1 becomes 0. (Switch between even and odd).
            index ^= 1

            # Right shift 'n' to process the next bit.
            n >>= 1

        # Return the counts of even and odd position bits.
        return bit_counts
```


Time and Space Complexity

The function `evenOddBit` computes the number of set bits at even and odd positions in the binary representation of a given integer `n`. To analyze the number of operations, it iterates bit by bit through the entire binary representation of `n`, which in the worst case has a length of `O(log n)`. In each iteration, it performs a constant number of operations: a bitwise AND (`&`), bitwise XOR (`^`), assignment, and right shift (`>>`). Therefore, **the time complexity is `O(log n)`**, as with each bit operation, we move one bit position to the right until we have shifted through all the bits.

The space complexity is determined by the amount of extra space the algorithm uses besides the input. Here, the extra space used is for the variable `i` and the fixed-size list `ans`. Since the size of the list `ans` does not depend on the size of the input integer `n` and is always a list of two elements, the space complexity is constant. **The space complexity is `O(1)`**.