338. Counting Bits

Easy | Bit Manipulation | Dynamic Programming

Problem Description

The problem states that given a non-negative integer `n`, we are required to find a list `ans` of length `n + 1` where each element `ans[i]` represents the count of `1` bits in the binary representation of `i`. In other words, for every number from `0` to `n`, we need to calculate how many `1`s are present in its binary form and store those values in an array.

Intuition

To understand the solution, it's important to grasp the concept of **bit manipulation**. In binary form, performing an `&` (bitwise AND) operation between a number `i` and `i - 1` will result in a number that is the same as `i` but with its least significant `1` bit removed. For example, `10` in binary is `1010`, and `9` is `1001`; performing `1010 & 1001` equals `1000`, which removes the least significant `1` bit in `1010`.

Knowing this, we can build an array `ans` that uses this property to count the number of `1`s. Start with `ans[0] = 0` because the binary representation of `0` has zero `1` bits. Then, iterate through numbers from `1` to `n`, and for each number `i`, calculate `ans[i]` as `ans[i & (i - 1)] + 1`. Here, `ans[i & (i - 1)]` gives us the count of `1`s for the number we get after removing the least significant `1` from `i`, and we add `1` because we've removed one `1` bit. This solution is efficient because it avoids recalculating the number of `1` bits for each number from scratch, using previous results instead.

Solution Approach

The solution provided adopts a dynamic programming approach, where past information is reused to optimize the calculation of current values. The algorithm uses a simple but powerful bit manipulation trick based on the property that for any given number `i`, the number `i & (i - 1)` results in `i` with the least significant `1` bit turned off. With this, we can recursively calculate the number of `1` bits for `i` if we already know the number of `1` bits for `i & (i - 1)`.

Here are the steps involved in the solution:

1. Initialize the array `ans` to have length `n + 1`, all initialized to `0`. This array is to hold the count of `1` bits for each index, which corresponds to a number from `0` to `n`.
2. Start a loop from `1` up to `n` because we already know `ans[0]` is `0`:
   - Inside the loop, for each number `i`, assign `ans[i] = ans[i & (i - 1)] + 1`.
     - Here, `i & (i - 1)` is the previously mentioned bit manipulation, which gives us a number less than `i`, which has one less `1` bit.
     - We then take the count of `1`s for that number (`ans[i & (i - 1)]`), which we've already calculated, and add `1` to account for the `1` bit we've just stripped off.
     - Hence, `ans[i]` accurately stores the count of `1` bits for the number `i`.
3. After the loop completes, return the `ans` array. It now contains the number of `1`s in the binary representation of each number from `0` to `n`.

This algorithm runs in O(n) time because it computes each entry in `ans` in constant time relative to `i`. The space complexity is also O(n), as additional storage proportional to the input size is created for the `ans` array. No other data structures are used, and the problem does not require any other specific algorithms or complex patterns.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have `n = 5`. Our goal is to find out how many `1` bits are present in the binary representation of numbers `0` to `5` and store it in an array `ans`.

1. Initialize `ans` to an array of length `n + 1`, filled with zeros: `ans = [0, 0, 0, 0, 0, 0]`.
2. Now perform the loop from `1` to `n` and fill in the `ans` array:
   - `i = 1`: The binary representation of `1` is `1`. Using the formula `ans[i] = ans[i & (i - 1)] + 1` means:
     - `ans[1] = ans[1 & 0] + 1 = ans[0] + 1 = 0 + 1 = 1`.
     - So, `ans` is now `[0, 1, 0, 0, 0, 0]`.
   - `i = 2`: The binary representation of `2` is `10`. Using the formula:
     - `ans[2] = ans[2 & 1] + 1 = ans[0] + 1 = 0 + 1 = 1`.
     - So, `ans` is now `[0, 1, 1, 0, 0, 0]`.
   - `i = 3`: The binary representation of `3` is `11`. Using the formula:
     - `ans[3] = ans[3 & 2] + 1 = ans[2] + 1 = 1 + 1 = 2`.
     - So, `ans` is now `[0, 1, 1, 2, 0, 0]`.
   - `i = 4`: The binary representation of `4` is `100`. Using the formula:
     - `ans[4] = ans[4 & 3] + 1 = ans[0] + 1 = 0 + 1 = 1`.
     - So, `ans` is now `[0, 1, 1, 2, 1, 0]`.
   - `i = 5`: The binary representation of `5` is `101`. Using the formula:
     - `ans[5] = ans[5 & 4] + 1 = ans[4] + 1 = 1 + 1 = 2`.
     - So, `ans` is now `[0, 1, 1, 2, 1, 2]`.
3. Our final `ans` array is `[0, 1, 1, 2, 1, 2]`, which represents the count of `1` bits in the binary representations of the numbers from `0` to `5`.

As we can see from this example, we successfully calculated the count of `1` bits for each number without recalculating from scratch each time. Instead, we relied on previously computed values to determine the count of `1`s for each subsequent number. The result is a highly efficient algorithm with O(n) time complexity and O(n) space complexity.

Solution Implementation

Python

```python
class Solution:
    def countBits(self, num: int) -> List[int]:
        # Create a list initialized with zeros for all elements up to num
        bit_counts = [0] * (num + 1)

        # Loop through all numbers from 1 to num
        for i in range(1, num + 1):
            # Use the bit count of the previous number that has the same bits
            # except the last set bit (i & (i - 1)) and add 1 for the last set bit
            # This works because i & (i - 1) drops the lowest set bit of i
            # For example, if i = 10100 (binary), i & (i - 1) = 10000, which is i without the last set bit.
            # ans[i & (i - 1)] already contains the count of 1s for 10000,
            # so we just need to add 1 for the dropped bit to get the count for 10100.
            bit_counts[i] = bit_counts[i & (i - 1)] + 1

        # Return the list of bit counts for all numbers from 0 to num
        return bit_counts
```

Java

```java
class Solution {
    public int[] countBits(int n) {
        // Create an array 'bitCounts' of size n+1 to hold the number of 1s for each number from 0 to n.
        int[] bitCounts = new int[n + 1];

        // Iterate over each number from 1 to n to calculate bit count.
        for (int i = 1; i <= n; ++i) {
            // Use the previously computed results to find the current number's bit count.
            // 'i & (i - 1)' drops the lowest set bit of i. So 'bitCounts[i & (i - 1)]' gives us
            // the count of bits for the current number without the lowest set bit.
            // Then, we add 1 for the dropped bit to get the final count for the current number.
            bitCounts[i] = bitCounts[i & (i - 1)] + 1;
        }

        // Return the populated array containing bit counts for all numbers from 0 to n.
        return bitCounts;
    }
}
```

C++

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

class Solution {
public:
    // This function generates a vector containing the number of 1-bits for all numbers from 0 to n
    vector<int> countBits(int n) {
        vector<int> bitCounts(n + 1); // Initialize a vector to store the count of 1-bits for each number

        for (int num = 0; num <= n; ++num) {
            // Use the built-in function __builtin_popcount to count the number of 1-bits in the binary representation of num
            bitCounts[num] = __builtin_popcount(num);
        }

        return bitCounts; // Return the vector of bit counts
    }
};
```

TypeScript

```typescript
// This function calculates the number of 1-bits for every number from 0 to n and returns them as an array.
// Each array index corresponds to a number, and its value is the count of 1s in the binary representation.
function countBits(n: number): number[] {
    // Initialize an array to store the bit counts with a size of n + 1, filled with zeros.
    const bitCounts: number[] = new Array(n + 1).fill(0);

    // Loop from 1 to n to calculate bit counts.
    for (let i = 1; i <= n; ++i) {
        // The number of bits in the current number is equal to the number of bits in the number obtained
        // by turning off the rightmost 1-bit in i's binary representation, i.e., i & (i - 1),
        // and then adding 1 (since we've removed one bit).
        bitCounts[i] = bitCounts[i & (i - 1)] + 1;
    }

    // Return the array containing bit counts for all numbers from 0 to n.
    return bitCounts;
}
```

```python
class Solution:
    def countBits(self, num: int) -> List[int]:
        # Create a list initialized with zeros for all elements up to num
        bit_counts = [0] * (num + 1)

        # Loop through all numbers from 1 to num
        for i in range(1, num + 1):
            # Use the bit count of the previous number that has the same bits
            # except the last set bit (i & (i - 1)) and add 1 for the last set bit
            # This works because i & (i - 1) drops the lowest set bit of i
            # For example, if i = 10100 (binary), i & (i - 1) = 10000, which is i without the last set bit.
            # ans[i & (i - 1)] already contains the count of 1s for 10000,
            # so we just need to add 1 for the dropped bit to get the count for 10100.
            bit_counts[i] = bit_counts[i & (i - 1)] + 1

        # Return the list of bit counts for all numbers from 0 to num
        return bit_counts
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`. This is because the loop runs from `1` to `n`, and within the loop, the operation `i & (i - 1)` is computed in constant time, as well as the increment `+ 1`. Thus, the loop constitutes the main factor in the time complexity, which linearly depends on `n`.

The space complexity is also `O(n)`. The additional space is used to store the result in the list `ans`, which contains `n + 1` elements. Apart from the `ans` list, only a constant amount of extra space is used for indices and temporary variables in the loop.