2220. Minimum Bit Flips to Convert Number

Easy Bit Manipulation

Problem Description

In the given LeetCode problem, we are tasked with finding the minimum number of bit flips required to convert one integer (`start`) to another integer (`goal`). A bit flip involves changing a single bit (0 to 1 or 1 to 0) in the binary representation of a number.

For example, if `start` is 7 (binary: 111) and `goal` is 5 (binary: 101), then one way to transform `start` to `goal` is by flipping the second bit from the right, resulting in a total of one bit flip.

It's important to note that we can choose any bit in the binary representation to flip. This includes leading zeros, which are not typically shown in binary representations. Thus, the goal is to find the number of flips that would result in the minimum transformations necessary.

Intuition

To find the solution, we can leverage a concept from binary arithmetic called the XOR operation, denoted by `^`. The XOR operation between two bits results in 1 if the bits are different (i.e., one is 0 and the other is 1) and 0 if the bits are the same.

Using the property that XOR outputs 1 for bits that are different and 0 for bits that are the same, we can XOR the `start` and `goal` numbers. The result will have 1s in all the positions where the bits of `start` and `goal` differ, which directly corresponds to the positions that would need to be flipped.

After that, the problem becomes counting the number of 1s in the resulting binary number. This count will give us the minimum number of bit flips, as each 1 represents a bit that needs to be flipped to convert `start` to `goal`.

The solution approach is as follows:

1. Perform an XOR operation between `start` and `goal` to get a new number that represents the bit differences.
2. Count the number of 1s in the binary representation of that new number. This can be done by repeatedly checking the least significant bit (using `t & 1`) and then right shifting the number (using `t >>= 1`) until all bits have been processed.
3. The count of 1s is the answer to the problem, which is the minimum number of bit flips required.

Solution Approach

The implementation of the solution follows a straightforward approach using simple bitwise operations to determine the number of differing bits between the `start` and `goal` integers.

Here's the step-by-step explanation of the algorithm used in the implementation:

1. The first step is to calculate the XOR of `start` and `goal` using `t = start ^ goal`. This gives us a number `t` where each bit set to `1` represents a difference between the corresponding bits in `start` and `goal`.
2. We need to count how many bits in `t` are set to `1`. To do this, we initialize a counter variable `ans` to 0.
3. We then enter a loop that continues until `t` becomes zero. Within this loop, we do the following:
   - We increment `ans` by the result of `t & 1`. The expression `t & 1` basically checks if the least significant bit (LSB) of `t` is set to `1`. If it is, this means there's a bit flip needed for that position, so we add one to our counter.
   - We then right shift `t` by one position using `t >>= 1`. This effectively moves all bits in `t` one position to the right, thus discarding the LSB we just checked and bringing the next bit into the position of LSB for the next iteration of the loop.
4. Once `t` is zero, this means we have counted all the bits that were set to `1` in `t`, which corresponds to the total number of bit flips needed to convert `start` to `goal`. At this point, we exit the loop.
5. Finally, we return `ans`, which now contains the minimum number of bit flips required.

The algorithm is highly efficient since the number of iterations in the loop is equal to the number of bits in `t`. Since integers in most programming languages (including Python) are represented by a fixed number of bits (e.g., 32 or 64 bits), this leads to a time complexity of O(1), as the loop runs a constant number of times relative to the size of the integer.

No additional data structures or complex patterns are needed for this solution, as it solely relies on bitwise operations, which are fundamental and efficient at the machine code level.

This implementation is elegant due to its simplicity and utilization of the properties of XOR to directly translate the problem of counting bit flips into a standard bit count problem. It showcases how a good grasp of bitwise operations can lead to simple and effective solutions for problems involving binary representations.

Example Walkthrough

Let's walk through a small example to understand how the solution approach works. Suppose we have the integers `start=8` (which is `1000` in binary) and `goal=10` (which is `1010` in binary).

1. First, we calculate the XOR of `start` and `goal` by using `t = start ^ goal`. In binary form, this is `1000 ^ 1010 = 0010`, so `t` would be `2` in decimal.
2. We now need to count the number of bits set to `1` in `t`. We initialize `ans` to `0`.
3. We enter a loop to count the set bits in `t` until `t` becomes zero.
   - We check if the LSB (least significant bit) is `1` by `t & 1`.
   - In the first iteration, `t=2` which is `0010` in binary, `t & 1` is `0`. So, `ans` remains `0`.
   - We right shift `t` by one position using `t >>= 1`, so `t` becomes `1` (binary `0001`).
4. We again check if the LSB is `1` by `t & 1`.
   - Now, `t=1` which is `0001` in binary, `t & 1` is `1`. So, we increment `ans` to `1`.
   - We right shift `t` by one using `t >>= 1`, `t` becomes `0`.
5. Now that `t` is zero, we have counted all the bits set to `1` in `t`. The loop finishes, and `ans` is `1`.
6. Finally, we conclude that the minimum number of bit flips required to convert `start` to `goal` is `1`.

This example illustrates the step-by-step computation required to solve the problem by simply using XOR and bit count operations.

Solution Implementation

Python

```python
class Solution:
    def minBitFlips(self, start: int, goal: int) -> int:
        # XOR operation to find the bits that are different between start and goal
        different_bits = start ^ goal

        # Initialize the count of bit flips required to 0
        bit_flips_count = 0

        # Count the number of bits set to 1 in different_bits
        while different_bits:
            # Increment the counter if the least significant bit is 1
            bit_flips_count += different_bits & 1
            # Right-shift to check the next bit
            different_bits >>= 1

        # Return the total count of flips needed
        return bit_flips_count
```

Java

```java
class Solution {

    // Function to count the minimum number of bit flips required to convert 'start' to 'goal'.
    public int minBitFlips(int start, int goal) {
        // XOR of 'start' and 'goal' will give us the bits that are different.
        int diffBits = start ^ goal;

        // This variable will hold the count of the number of flips required.
        int flipCount = 0;

        // Process each bit of 'diffBits' to count the number of set bits (flips required).
        while (diffBits != 0) {
            // Increment 'flipCount' if the least significant bit of 'diffBits' is 1.
            flipCount += diffBits & 1;

            // Right shift 'diffBits' by 1 to process the next bit.
            diffBits >>= 1;
        }

        // Return the count of flips required.
        return flipCount;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to count the minimum number of bit flips to convert start to goal.
    int minBitFlips(int start, int goal) {
        // XOR the start and goal to find the differences
        int diff = start ^ goal;

        // Initialize count to store the number of flips needed
        int flipCount = 0;

        // Loop through all the bits of diff
        while (diff) {
            // If the least significant bit is 1, it needs to be flipped
            flipCount += diff & 1;

            // Right shift diff to check the next bit in the next iteration
            diff >>= 1;
        }

        // Return the total number of flips needed to convert start to goal
        return flipCount;
    }
};
```

TypeScript

```typescript
/**
 * Calculates the minimum number of bit flips required to convert the `start` number to the `goal` number.
 *
 * @param {number} start - The starting integer to be transformed.
 * @param {number} goal - The goal integer to reach by flipping bits.
 * @return {number} The minimum number of bit flips required.
 */
function minBitFlips(start: number, goal: number): number {
    // Perform an XOR operation between start and goal to determine the difference in bits.
    let difference = start ^ goal;
    let flipsRequired = 0; // Initialize the count of required flips to 0.

    // Loop until all bits of the difference are processed.
    while (difference !== 0) {
        // Increment the count if the least significant bit is a 1, indicating a bit flip is required.
        flipsRequired += difference & 1;
        // Right shift the difference by 1 to check the next bit in the next iteration.
        difference >>= 1;
    }

    // Return the total count of flips required to turn `start` into `goal`.
    return flipsRequired;
}
```

```python
class Solution:
    def minBitFlips(self, start: int, goal: int) -> int:
        # XOR operation to find the bits that are different between start and goal
        different_bits = start ^ goal

        # Initialize the count of bit flips required to 0
        bit_flips_count = 0

        # Count the number of bits set to 1 in different_bits
        while different_bits:
            # Increment the counter if the least significant bit is 1
            bit_flips_count += different_bits & 1
            # Right-shift to check the next bit
            different_bits >>= 1

        # Return the total count of flips needed
        return bit_flips_count
```

Time and Space Complexity

The given Python code snippet defines a function `minBitFlips` which calculates the minimum number of bit flips required to transform an integer `start` into another integer `goal`. To perform this task, it uses bitwise XOR (`^`) and the bitwise AND (`&`) operation followed by right shift operations.

Time Complexity:

The time complexity of the given function mainly depends on the number of bits in the binary representation of the XOR result of `start` and `goal`.

1. Calculating `t = start ^ goal` takes `O(1)` time, assuming that XOR operation on two integers is a constant-time operation as the integers are of fixed size (typically 32 or 64 bits in modern architectures).
2. The while loop runs as many times as there are bits in `t`. In the worst case, `t` has as many bits set as the binary representation of the larger of `start` or `goal`. Therefore, this results in `O(b)` time complexity, where `b` is the number of bits required to represent the numbers `start` or `goal`.

Putting it together, for an integer that is represented using `b` bits (for example, 32 bits in the case of Python ints), the overall time complexity is `O(b)`. Since `b` is fixed, one might also consider this as `O(1)` in a practical sense, but technically speaking with regard to the input size, it's `O(b)`.

Therefore, the time complexity is `O(b)` or `O(1)` if we consider the fixed-size integer representation.

Space Complexity:

For space complexity:

1. The variable `t` requires `O(1)` space because it is an integer with a fixed size.
2. The variable `ans` also requires `O(1)` space, as it is an integer to store the final count of bit flips.

Hence, there are no additional data structures that grow with input size. Therefore, the space complexity of the function is `O(1)`.