2571. Minimum Operations to Reduce an Integer to 0

`Medium` `Greedy` `Bit Manipulation` `Dynamic Programming`

Problem Description

You are given a positive integer n. The task is to transform n to 0 by repeatedly performing the following operation: add or subtract a power of 2 to/from n. The available powers of 2 range from 2^0 up any higher power (as 2^1, 2^2, and so on). The goal is to find the *minimum* number of such operations required to achieve $n = 0$.

In simpler words, you're essentially allowed to repeatedly increase or decrease the integer n by any value that is a power of 2 (like 1, 2, 4, 8, and so on), and you need to find the least number of times you have to do this to get to 0.

Intuition

The solution involves a strategic approach rather than trying out different powers of 2 at random. We look at the binary representation of the given integer n. Since binary representation is based on powers of 2, incrementing or decrementing by powers of 2 can be visualized as flipping specific bits in the binary form.

So, we scan through the binary digits (bits) of n from least significant to most significant (from the right-hand side to the left).

- If we encounter a 1 bit, this represents a power of 2 that we need to cancel out. We keep a running total of consecutive 1 bits because a string of 1s may be dealt with more efficiently together rather than individually.

- When we encounter a 0 bit, this provides an opportunity to address the consecutive 1 bits encountered before it. If there's only one 1 bit, we can perform a single subtract operation to remove it. If there are multiple 1s, we can turn all but one of them into 0 by adding a power of 2 that's just higher than the string of 1s. This adds 1 to the count of subtract operations needed.

In the end, we might be left with a string of 1s at the most significant end. We'll need two operations if there are multiple 1s (add and then subtract the next higher power of 2), or just one if there's a single 1.

The algorithm hence devised is cautious; it chooses the most effective operation at each step due to its bitwise considerations, ensuring fewer total operations.

Solution Approach

The Python code provided implements the algorithm through a mix of bitwise operations and a greedy approach.

Let's walk through the implementation:

1. Initialize `ans` and `cnt`. The variable `ans` keeps track of the total number of operations required, and `cnt` counts the consecutive 1 bits in the binary representation of n.

2. Process the binary digits of n from least significant to most significant. This is done by examining n bit by bit in a loop where the iteration involves a right shift operation `n >>= 1`, effectively moving to the next bit towards the left.

3. When the current bit is 1 (checked by `if n & 1`), increment the `cnt` by 1. This is because each 1 represents a power of 2 that must be addressed.

4. If the current bit is 0 and `cnt` is not zero, this indicates we have just finished processing a sequence of 1 bits. Now, determine how to handle them:
   - If `cnt` equals 1, increment `ans` by 1, as we need one operation to subtract this power of 2.
   - If `cnt` is more than 1, increment `ans` by 1 and set `cnt` to 1. The rationale here is that we've added a power of 2 that turns all but one of the consecutive 1 bits into 0. We're left with one last subtraction operation to perform later.

5. After processing all bits, there may be a leftover `cnt`:
   - If `cnt` is 1, we only need one more operation, so add 1 to `ans`.
   - If `cnt` is greater than 1, we need a two-step operation where we first add and then subtract the next power of 2, hence add 2 to `ans`.

6. Finally, return the accumulated result `ans`, which gives us the minimum number of operations needed to transform n into 0.

By leveraging bitwise operations, the solution is both efficient and straightforward. There's no need for iteration or recursion that directly considers powers of 2; instead, the solution operates implicitly on powers of 2 by manipulating binary digits, which are intrinsically linked to powers of 2.

Example Walkthrough

Let's illustrate the solution approach using the example of $n = 23$. The binary representation of 23 is 10111. We will apply the solution approach step by step:

1. Initialize `ans = 0` and `cnt = 0`. No operations have been done yet.
2. As n is 10111, we process from the rightmost bit to the left:
   - Bit 1 (rightmost): `n & 1` is true so increment `cnt` to 1.
   - Bit 2: `n & 1` is true so increment `cnt` to 2.
   - Bit 3: `n & 1` is true so increment `cnt` to 3.
   - Bit 4: `n & 1` is false, and `cnt` is 3. Therefore, increment `ans` to 1 and reset `cnt` to 1 (because we perform an add operation to turn the sequence 111 into 1000, then add another power of 2 for later).
   - Last shift will happen, and now n becomes 2 (10 in binary), and we've processed all bits.
3. Leftover `cnt` is 1 because our binary representation now starts with 1. We increment `ans` by 1 because a final subtract operation is needed.
4. Finally, `ans` is 2 because we performed 2 operations in total: converting 10111 to 11000 (+1 operation) and then to 0 (-1 operation).

The minimum number of operations required to transform $n = 23$ into 0 using the solution approach is 2.

Python Solution

```python
class Solution:
    def minOperations(self, n: int) -> int:
        # Initialize operations count (ops) and a counter for consecutive ones (consecutive_ones).
        ops = 0
        consecutive_ones = 0

        # Process all bits of the integer n from right to left (LSB to MSB).
        while n:
            # Check if the least significant bit (LSB) is 1.
            if n & 1:
                # If it is, increase the counter for consecutive ones.
                consecutive_ones += 1
            # If it is 0 and the counter for consecutive ones is not zero.
            elif consecutive_ones:
                # A zero after some ones means a completed set of "10" or "110."
                # Operation needed to convert this set to "00" or "100".
                ops += 1
                # Reset the counter for consecutive ones.
                # If '1' in binary is already minimized, and for '11' we only need one operation
                # to change it to '10' and then we 'carry the 1' so to speak.
                consecutive_ones = 0 if consecutive_ones > 1 else 1

            # Right shift n by 1 to check the next bit.
            n >>= 1

        # If there is a residual 1 then an operation is needed;
        # this could be a trailing '1' or '10' after the end of the loop.
        if consecutive_ones == 1:
            # Increment the operations count by 1.
            ops += 1
        # If there are more than 1 ones, an extra operation is needed to handle the carry.
        elif consecutive_ones > 1:
            # Increment the operations count by 2. This is the case for '11' in binary,
            # where one operation is to change '11' to '10' and second operation to change
            # '10' to '00'.
            ops += 2

        # Return the number of operations required.
        return ops
```

Java Solution

```java
class Solution {

    public int minOperations(int num) {
        // Initialize variables for answer and contiguous ones count
        int operationsCount = 0;
        int contiguousOnesCount = 0;

        // Loop while number is greater than zero
        for (; num > 0; num >>= 1) {
            // If the least significant bit is 1
            if ((num & 1) == 1) {
                // Increment count of contiguous ones
                contiguousOnesCount++;
            } else if (contiguousOnesCount > 0) {
                // If the least significant bit is 0 and we have seen a contiguous sequence of ones
                operationsCount++; // Increase operations, since this is an operation to flip a zero after a one
                // Reset/modify contiguous ones count based on the previous count
                contiguousOnesCount = (contiguousOnesCount == 1) ? 0 : 1;
            }
        }

        // After processing all bits, if there is a single one left, it is a valid operation to flip it
        operationsCount += (contiguousOnesCount == 1) ? 1 : 0;

        // If there are more than one contiguous ones left, we need two operations:
        // one to flip the first one, and another one for the last one.
        operationsCount += (contiguousOnesCount > 1) ? 2 : 0;

        // Return the total number of operations required
        return operationsCount;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int minOperations(int n) {
        int operations = 0; // This will hold the total number of operations required.
        int onesCount = 0; // This will count the number of consecutive '1' bits.

        // Process each bit, starting with the least significant bit (LSB).
        while (n > 0) {
            if ((n & 1) == 1) {
                // If the current bit is a '1', increment the count of consecutive ones.
                ++onesCount;
            } else if (onesCount > 0) {
                // If we kit a '0' after counting some ones, we need an operation (either a flip or a move).
                ++operations;

                // If there was only one '1', we reset the count. Otherwise, we keep the count as '1',
                // because the bit flip would transform "01" to "10", leaving us with one '1' to move.
                onesCount = (onesCount == 1) ? 0 : 1;
            }
            // Right shift the bits in n to process the next bit in the next iteration.
            n >>= 1;
        }

        // After processing all bits, if we have one '1' left, we need one more operation to remove it.
        if (onesCount == 1) {
            ++operations;
        }

        // If we have more than one '1', we need an additional two operations: one for flipping and one for moving.
        if (onesCount > 1) {
            operations += 2;
        }

        return operations; // Return the total number of operations calculated.
    }
};
```

Typescript Solution

```typescript
function minOperations(n: number): number {
    let operationsCount = 0; // Will hold the total number of operations required.
    let consecOnesCount = 0;  // Counter for consecutive 1's in binary representation of n.

    // Iterate through the bits of n.
    while (n !== 0) {
        // If the least significant bit is a 1.
        if (n & 1) {
            consecOnesCount += 1; // We found a consecutive one, increase the counter.
        }
        // If it's a 0 and there are consecutive 1's counted before.
        else if (consecOnesCount > 0) {
            operationsCount += 1; // We need an operation to separate the consecutive 1's.

            // Reset consecOnesCount or set to 1 based on whether we had a single consecutive one or more.
            consecOnesCount = (consecOnesCount == 1) ? 0 : 1;
        }
        // Right shift n to check the next bit.
        n >>= 1;
    }

    // If there's a single 1 left.
    if (consecOnesCount === 1) {
        operationsCount += 1; // We'll need one more operation.
    }
    // If there are more than one consecutive 1's left.
    else if (consecOnesCount > 1) {
        operationsCount += 2; // We'll need two operations to handle them.
    }

    // Return the total operations count.
    return operationsCount;
}
```

Time and Space Complexity

The time complexity of the provided code is $O(\log n)$ because the primary operation within the `while` loop is a bitwise right shift on n, which effectively divides n by 2. The number of iterations is directly proportional to the number of bits in the binary representation of n, which is $\log n$ in base 2.

The space complexity of the code is $O(1)$ since the extra space used is constant, irrespective of the size of n. The variables `ans` and `cnt` use a fixed amount of space during the execution of the algorithm.