397. Integer Replacement

Medium Greedy Bit Manipulation Memoization Dynamic Programming

Leetcode Link

Problem Description

Given a positive integer `n`, the task is to transform this number to 1. There are two types of operations that you are allowed to perform:

1. If `n` is even, you can divide it by 2 (`n = n / 2`).
2. If `n` is odd, you can either increase it by 1 (`n = n + 1`) or decrease it by 1 (`n = n - 1`).

Each operation counts as a single step, and the goal is to find the minimum number of steps required to reduce `n` to 1.

Intuition

The intuition behind the solution involves understanding parity (odd or even nature) of `n` and using bitwise operations for efficiency. For an even `n`, the decision is straightforward: divide by 2. This halving operation is efficient because it significantly reduces `n`.

For an odd `n`, the decision to subtract or add 1 relies on considering the bits of `n`. Specifically, we observe the following patterns:

- When the least significant two bits of `n` are `01` (meaning the next division will yield an even number), it is often better to subtract 1 than to add 1, because we want to get to an even number to divide by 2 in the next step.
- However, there is a special case when `n` is `3`; the best operation is to subtract 1 twice to reach `1` rather than add 1 and then divide twice.
- If the least significant two bits are `11`, adding 1 converts `n` to an even number, which can then be halved. This is except for the case when `n` is `3`, as mentioned above.

The approach makes use of bitwise operations:

- We use `n & 1` to check if `n` is even or odd (`0` means even and `1` means odd).
- The right shift `n >>= 1` effectively divides `n` by 2.
- We check if the last two bits are `11` by `n & 3`, and if the result is `3`, we know adding 1 is a favorable operation.

The solution employs a loop that iteratively applies the optimal operation until `n` becomes `1`, and a counter `ans` keeps track of the number of operations performed.

Solution Approach

The solution provided uses a greedy approach to minimize the number of operations needed to reduce `n` to 1 with the two allowed operations, substituting and dividing.

Here's a walkthrough of the algorithm using the Reference Solution Approach:

1. Initialize `ans` to 0. This variable will keep track of the total number of operations performed.
2. Enter a loop that continues until `n` is equal to 1.
3. Inside the loop, check if `n` is even by using the bitwise AND operation `n & 1`. If the result is 0, it means `n` is even.
4. If `n` is even, we apply the bitwise right shift operation `n >>= 1` to n. This operation is equivalent to dividing `n` by 2.
5. If `n` is odd, we have to decide whether to increment or decrement `n`. To make this decision:
   - Check if `n` is not equal to 3 and if the last two bits of `n` are `11` by applying the bitwise AND operation `n & 3`. If the result is 3, it's better to increment `n` by 1, because doing so will lead to an even number after this operation.
   - In all other cases (when `n` is odd, and either `n` is 3 or the last two bits of `n` are not `11`), it's better to decrement `n` by 1.
6. After each operation (incrementing, decrementing, or dividing), increment `ans` by 1 to count the operation.
7. Once `n` is reduced to 1, exit the loop, and return `ans` as the total number of operations performed.

The solution applies mathematical operations and understands binary representations to guide the decision-making process. No additional data structures are used, and the algorithm runs in a time complexity that is logarithmic to the value of `n`, specifically O(log n), because each division by two halves the problem size.

Example Walkthrough

Let's illustrate the solution approach with a small example using `n = 15`.

1. Initialize `ans` to 0. At the start, no steps have been taken yet, so `ans` is `0`.
2. Because `n` is not 1, we enter the loop.
3. As `n` is odd (`15 & 1` is `1`), we proceed to check if we should decrement or increment `n`.
4. Since `n` equals 15, which is not 3, and the last two bits of `n` are `11` (`15 & 3` gives `3`), we increment `n` by 1, resulting in `n = 16`. Now `ans = 1`.
5. Now `n` is even (`16 & 1` is `0`), so we right shift `n`, effectively dividing it by 2. `n` becomes `8` (`n >>= 1`). Now `ans = 2`.
6. `n` is still even, so we keep right shifting. `n` becomes `4`. Now `ans = 3`.
7. Continue with the even case; `n` becomes `2`. Now `ans = 4`.
8. Finally, `n` is `2`, which is even again, and after one more shift, `n` becomes `1`. Now `ans = 5`.
9. Since `n` is now equal to 1, we break out of the loop.
10. The value of `ans` is `5`, representing the minimum number of steps needed to transform `n` from `15` to `1`.

The solution successfully applies the steps from the Reference Solution Approach, selecting operations that gradually reduce `n` to 1 with optimal efficiency, ending with an answer of 5 steps for this example.

Python Solution

```python
class Solution:
    def integer_replacement(self, n: int) -> int:
        # Initialize a counter for the number of steps taken
        step_count = 0

        # Continue processing until the integer becomes 1
        while n != 1:
            # If n is even, shift it right by 1 (equivalent to dividing by 2)
            if (n & 1) == 0:
                n >>= 1
            # If n is one less than a multiple of 4 (except when n is 3)
            # increment n (e.g., for 7 -> 8 is better than 7 -> 6)
            elif n != 3 and (n & 3) == 3:
                n += 1
            # In all other cases (n is odd and not handled by the above condition), decrement n
            else:
                n -= 1
            # Increment the step count after each operation
            step_count += 1

        # Return the total number of steps taken
        return step_count

```

Java Solution

```java
class Solution {
    public int integerReplacement(int n) {
        int steps = 0; // Counter for the number of steps taken to transform 'n' to 1

        while (n != 1) {
            if ((n & 1) == 0) { // If 'n' is even
                n >>>= 1; // Right shift (unsigned) to divide 'n' by 2
            } else if (n != 3 && (n & 3) == 3) { // If 'n' is not 3 and the last two bits are 11
                n++; // Increment 'n' since it leads to more 0s when it's divided by 2 subsequently
            } else {
                n--; // Decrement 'n' if it's odd and doesn't match the previous case
            }
            steps++; // Increment the step count after each operation
        }
        return steps; // Return the total number of steps once 'n' is reduced to 1
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to determine the minimum number of operations to transform
    // a given integer n to 1 by either decrementing by 1, incrementing by 1,
    // or halving it when even.
    int integerReplacement(int n) {
        int operationsCount = 0; // variable to count the number of operations
        long longNumber = n; // use a long to handle overflow

        // Continue until longNumber becomes 1
        while (longNumber != 1) {
            if ((longNumber & 1) == 0) {
                // If longNumber is even, halve it
                longNumber >>= 1;
            } else if (longNumber != 3 && (longNumber & 3) == 3) {
                // If longNumber is not 3 and ends with binary '11', increment it
                longNumber++;
            } else {
                // If longNumber is odd and not covered by the above case, decrement it
                longNumber--;
            }
            // Increment the operation count after each operation
            operationsCount++;
        }

        // Return the total count of operations performed
        return operationsCount;
    }
};

```

Typescript Solution

```typescript
// Global variable to count the number of operations
let operationsCount: number = 0;

// Function to determine the minimum number of operations to transform
// a given integer n to 1 by either decrementing by 1, incrementing by 1,
// or halving it when even.
function integerReplacement(n: number): number {
    // Initialize count and use a 'BigInt' for 'n' to handle overflow
    operationsCount = 0;
    let longNumber: bigint = BigInt(n);

    // Continue processing until longNumber becomes 1
    while (longNumber !== BigInt(1)) {
        if ((longNumber & BigInt(1)) === BigInt(0)) {
            // If longNumber is even, right shift equals dividing by 2
            longNumber >>= BigInt(1);
        } else if (longNumber !== BigInt(3) && (longNumber & BigInt(3)) === BigInt(3)) {
            // If longNumber ends with binary '11' (is odd) and is not 3, increment
            longNumber++;
        } else {
            // If longNumber is odd and not covered by the above case, decrement
            longNumber--;
        }
        // Increment the operation count after each operation
        operationsCount++;
    }

    // Return the total count of operations performed
    return operationsCount;
}

// Example usage:
// To call the function, simply invoke it with an integer argument
// let result: number = integerReplacement(1234);

```

Time and Space Complexity

The given code implements an algorithm to determine the minimum number of operations to reduce a number `n` to 1 by either dividing it by 2 if it's even or subtracting 1 or adding 1 if it's odd, which converges to division by 2 when possible.

Time Complexity:

The time complexity of the algorithm is determined by the number of operations needed to reduce `n` to 1. In the worst case scenario, for each bit in the binary representation of `n`, the loop might be executed twice (once for subtraction/addition to make it even, and once for the division by 2). However, the addition operation in `n += 1` can lead to the removal of multiple trailing '1's in binary, which means it's possible to skip several steps at once. Therefore, the time complexity is `O(log n)` in the average case, and in the worst case, it's slightly more than `O(log n)` due to the possible addition steps that can reduce the number of 1's in the binary representation.

Space Complexity:

The space complexity of the code is `O(1)`, as the algorithm uses a fixed amount of space - the variable `ans` to keep count of the operations, and `n` is modified in place without using additional memory that scales with the input size.