991. Broken Calculator

Medium Greedy Math

Problem Description

In this problem, we are given a "broken" calculator that can perform only two operations:

- 1. Multiply the current number displayed by 2.
- 2. Subtract 1 from the current number displayed.

called target using the fewest number of operations. We need to find out the minimum operations required to achieve this.

The calculator initially shows a number called startValue. The task is to transform this startValue into another given number

Intuition

The intuitive approach to solve this problem might be to start from startValue and try to reach target using a series of multiplications and subtractions. However, this can be very inefficient because the number of possibilities can explode, leading to a high time complexity solution.

Instead, we reverse our thinking process and start from target and try to reach start Value. We take advantage of the fact that the reverse operations (dividing by 2 and adding 1) are more restricted since we can divide by 2 only when the current number is

even. This gives us a direction in our decision-making process and reduces the number of choices at each step, making the problem much simpler. Here's the step-by-step intuition: 1. If target is greater than startValue, we can only reach it by performing the reverse operations because multiplying startValue may overshoot

- target.
- 2. If target is odd, the last operation performed must have been subtracting 1 (since we cannot divide an odd number by 2). So, we add 1 to target.
- 3. If target is even, the last operation could have been a division by 2, so we divide the target by 2. 4. We count each operation performed, and once target is less than or equal to startValue, we stop. 5. The remaining difference between startValue and target represents the number of times we'd need to subtract 1 from startValue to reach
- target.
- By following these steps, we can ensure that we use the minimum operations to transform start Value into target on the broken calculator.

Solution Approach

The Solution provided is a direct implementation of the thought process described in the Intuition section. It's a linear approach,

where the algorithm goes through a series of steps to transform the target back to the startValue. The crucial insight is that

working backward from the target value is more efficient than trying to approach the target starting from the startValue. This is

because multiplying can lead to rapidly overshooting the goal, but working backward constrains the choices. Here's a step-by-step walkthrough of the implementation: • A variable ans is initialized to 0 to count the number of operations needed. • While startValue is less than target, a loop continues to perform the reverse operations to bring target closer to startValue.

∘ Inside the loop, first, there is a check to see if target is odd using target & 1. This is a bitwise AND operation, which is equivalent to

checking if the last bit of target is 1. If it is odd (true), 1 is added to target to simulate the reverse of a subtract operation.

linear in terms of the number of operations needed.

efficiently reaches the optimal solution.

- If target is even (false), the target is right-shifted by 1 bit using target >>= 1, which is equivalent to dividing the target by 2. • The ans counter is then incremented for each operation performed, whether it's an addition or a division.
- reflect these operations. • Finally, ans is returned as the minimum number of operations needed to display target on the calculator from startValue. This approach is efficient because each iteration reduces target significantly (either by a factor of 2 or approaching the even number when odd), and there are no unneeded complexities in performing the operations, leading to a time complexity that is

• When target is less than or equal to startValue, the loop ends. The final difference between startValue and target indicates how many

subtractions would be necessary if we were working forwards from startValue. Thus, startValue - target is added to the ans counter to

By continuously halving the target, we ensure that we're using the most significant reduction at each step when possible, and only when target becomes odd, we perform an addition. This way, despite simulating an inefficient calculator, the algorithm

Example Walkthrough Let's walk through an example to illustrate the solution approach using the broken calculator problem.

Now we need to find the minimum operations to transform startValue into target. **Solution Steps**

1. Since target > startValue, we know we have to work backwards from target to startValue. 2. Initial ans is 0 because no operations have been performed yet.

Conclusion

Python

Example

• startValue = 5

• target = 8

- \circ We perform the reverse of multiplication by dividing by 2. So, target = target / 2 = 8 / 2 = 4.
- Again, we divide by 2. So, target = target / 2 = 4 / 2 = 2. Increment ans to 2.
- Third iteration: target is 2, which is even. Continue with division. So, target = target / 2 = 2 / 2 = 1.
- Increment ans to 3.

Now we enter the loop:

• Increment ans to 1.

• First iteration: target is 8, which is even.

• Second iteration: target is 4, which is even.

• Add 4 to ans, which is 3 from earlier. Now ans = 3 + 4 = 7.

Initialize the number of operations to 0

Increment the operations counter

Return the total number of operations performed

// Once we reach or go below startValue, add the difference

// Function to calculate the minimum number of operations required

// to reach from 'startValue' to 'target' by either multiplying by 2 or

// This is the reverse operation of multiplying by 2.

// Once we have a 'startValue' greater than or equal to the 'target',

Loop until the start value is greater than or equal to target

If target is even, divide it by 2 using right shift

If target is odd, increment it to make it even

// we need to perform ('startValue' - 'target') subtractions to reach the 'target'.

int operationCount = 0; // Variable to store the minimum number of operations.

// An odd number cannot be reached by doubling (which always results in an even number),

// If the target is even, perform a right bit shift operation equivalent to dividing by 2.

// Continue the process until startValue is at least as large as the target.

// If the target is an odd number, increment it to make it even.

// so we add 1 (which is the reverse operation of subtracting 1).

// Increase the operation count after each modification to the target.

// Since at this point only decrements are allowed

numOfOperations += startValue - target;

// Return the total number of operations

return numOfOperations;

// subtracting 1 in each operation.

while (startValue < target) {</pre>

if (target & 1) {

target++;

target >>= 1;

++operationCount;

else {

int brokenCalc(int startValue, int target) {

Loop until the start value is greater than or equal to target

If target is even, divide it by 2 using right shift

If target is odd, increment it to make it even

- The loop ends here because target = 1 is less than startValue = 5. We now add startValue target to ans.
- The minimum number of operations required for the broken calculator starting from 5 to reach 8 is 7. These operations are three divisions (halving the number three times) and four subtractions.

• startValue - target = 5 - 1 = 4. This means we need to perform 4 subtractions if we were moving forwards from startValue.

class Solution: def broken_calc(self, start_value: int, target: int) -> int:

operations = 0

else:

while start_value < target:</pre>

target += 1

target >>= 1

if target % 2:

operations += 1

return operations

Solution Implementation

```
# Add the difference between start value and the target to the operations
# This handles the case where we need to perform 'multiply by 2' operations
operations += start_value - target
```

Java

```
class Solution {
    /**
    * Calculates the minimum number of operations to transform
    * startValue to target by either multiplying by 2 or decrementing by 1.
     * @param startValue The starting value.
     * @param target
                       The target value.
                        The minimum number of operations required.
    * @return
    public int brokenCalc(int startValue, int target) {
        int numOfOperations = 0; // Initialize operation count
       // Work backwards from the target value until we reach or go below startValue
       while (startValue < target) {</pre>
            if ((target & 1) == 1) {
                // If target is odd, increment it (reverse of decrementing in forward direction)
                target++;
            } else {
                // If target is even, halve it (reverse of doubling in forward direction)
                target >>= 1; // Equivalent to target /= 2;
            numOfOperations++; // Increment the count of operations
```

C++

public:

class Solution {

```
// Once we have a startValue greater than or equal to the target,
       // we need to perform (startValue - target) subtractions to reach the target.
       operationCount += startValue - target;
       // Return the total number of operations required.
       return operationCount;
};
TypeScript
// Global variable to store the minimum number of operations.
let operationCount = 0;
// Function to calculate the minimum number of operations required
// to reach from 'startValue' to 'target' by either multiplying by 2 or
// subtracting 1 in each operation.
function brokenCalc(startValue: number, target: number): number {
    // Reset operation count at the start of the function call.
    operationCount = 0;
   // Continue the process until 'startValue' is at least as large as the 'target'.
    while (startValue < target) {</pre>
       // If the 'target' is an odd number, increment it to make it even.
       // An odd number cannot be reached by doubling (which always results
       // in an even number), so we add 1 (which is the reverse operation
       // of subtracting 1 in the problem context).
       if (target % 2 === 1) {
            target++;
       // If the 'target' is even, divide it by 2.
       // This is the reverse operation of multiplying by 2.
       else {
            target /= 2;
       // Increase the operation count after each modification to the 'target'.
       operationCount++;
```

```
# Increment the operations counter
    operations += 1
# Add the difference between start value and the target to the operations
# This handles the case where we need to perform 'multiply by 2' operations
```

Return the total number of operations performed

operations += start_value - target

operationCount += startValue - target;

return operationCount;

operations = 0

else:

return operations

Time and Space Complexity

while start_value < target:</pre>

target += 1

target >>= 1

if target % 2:

class Solution:

// Return the total number of operations required.

Initialize the number of operations to 0

def broken_calc(self, start_value: int, target: int) -> int:

Time Complexity The time complexity of the given code is O(log(target)). The while loop runs until startValue is greater than or equal to target.

At each iteration of the loop, if target is even, it is halved (which significantly decreases the target in logarithmic steps), or if it's odd, it is incremented by 1, which eventually makes it even for the next step. Since target is divided by 2 in potentially every other iteration, the loop runs in O(log(target)) time with respect to the target value.

Space Complexity

The space complexity of the code is 0(1). The solution does not use any additional storage that grows with the size of the input. It uses a fixed amount of space for the variables ans, startValue, and target irrespective of the input size.