1523. Count Odd Numbers in an Interval Range

Easy Math

Leetcode Link

Problem Description

In this problem, you are given two non-negative integers `low` and `high`. The task is to find and return the count of odd numbers that are between `low` and `high` (inclusive). In other words, you need to determine the number of odd numbers in the integer range starting from `low` to `high`, including both `low` and `high` if they happen to be odd.

Intuition

The intuition behind the solution lies in understanding how odd and even numbers are distributed in a range of integers. Consider any two consecutive numbers, one will always be even, and the other will be odd. Thus, in any range of integers, the count of odd and even numbers should be roughly equal.

However, the exact count may vary depending on the parity of `low` and `high` and whether the range includes an odd or even number of integers. The trick to solving this efficiently is to use a formula that captures this observation without having to iterate over each number in the range.

Here's how the solution is derived:

- The number of integers in the range `low` to `high` (inclusive) can be given by `high - low + 1`.
- For a range starting from 1 to a number `n`, the number of odd numbers can be calculated by `(n + 1) // 2`. This is because the sequence of odd and even numbers is regular, and half of the numbers up to `n` will be odd (plus one for odd `n`).
- If `low` and `high` are both odd or both even, the ranges from 1 to `low - 1` and 1 to `high` contain the same number of odd and even numbers. The difference of these two will give us the count of odd numbers between `low` and `high`.
- If `low` is odd and `high` is even (or vice versa), then we have to adjust our calculation to ensure we count the extra odd or even number at the start of the range.

So, the formula `((high + 1) >> 1) - (low >> 1)` does exactly this by effectively calculating the count of odd numbers from 1 to `high` and subtracting the count of odd numbers from 1 to `low - 1`. The `>> 1` is a bit-shift operation that divides the number by 2, which is the same as using `// 2` for integers but may be faster in certain contexts.

This results in a simple, efficient solution that exploits the pattern in which odd and even numbers occur within ranges.

Solution Approach

The solution uses an efficient mathematical approach rather than brute-forcing through the range to count odd numbers. The key observations made about the distribution of odd and even numbers in a range are used to derive a simple formula. Here is a step-by-step breakdown of the implementation:

1. The expression `(high + 1) >> 1` calculates the number of odd numbers from the range `1` to `high`. The term `high + 1` adds one more count to facilitate the correct division by 2 (since we are interested in odd numbers). The bit-shift operation `>> 1` is effectively dividing the value by `2`.

2. Similarly, the expression `(low >> 1)` calculates the number of odd numbers in the range from `1` to `low - 1`. This works because `low` itself is not included and the bit-shift by one position to the right divides `low` by `2`. If `low` is odd, subtracting one before division gives the correct count of odd numbers below `low`.

3. By subtracting the count of odd numbers below `low` from the count of odd numbers up to and including `high`, what remains is the count of odd numbers in the inclusive range between `low` and `high`.

The implementation relies on no other algorithms or data structures. No loops or recursive calls are used. It is a direct application of an arithmetic operation and a bit-shift, which are both O(1) time complexity operations. The efficiency of the solution is a result of the formula which uses the inherent pattern of alternating odd and even numbers.

Example Walkthrough

Let's work through a small example to illustrate the solution approach. Consider the range of integers from `low = 3` to `high = 7`. We want to count how many odd numbers there are between 3 and 7, inclusive.

Here are the numbers in the range: 3, 4, 5, 6, 7. The odd numbers are 3, 5, and 7. There are 3 odd numbers in this range.

Now let's use the solution approach to get the same result:

1. We calculate the number of odd numbers up to `high` with the expression `(high + 1) >> 1`. For `high = 7`, this gives us `(7 + 1) >> 1`, which is `8 >> 1`. Bit-shifting `8` one position to the right is the same as dividing `8` by `2`, which gives us `4`. So there are 4 odd numbers from 1 to 7 inclusive.

2. We calculate the number of odd numbers before `low` with the expression `(low >> 1)`. For `low = 3`, this gives us `3 >> 1`, which is `1` when rounded down (as the bit-shifting operation inherently does for integers). So, there is 1 odd number in the range 1 to 2 (since `low` is not included).

3. We find the count of odd numbers between `low` and `high` by subtracting the count of odd numbers below `low` from the count of odd numbers up to `high`. In this case, we subtract `1` from `4` to get `3`, which is the count of odd numbers in the range of 3 to 7 inclusive.

As evidenced by the example, the formula `((high + 1) >> 1) - (low >> 1)` correctly calculates the number of odd numbers within a given range without the need for iterating over all elements in the range, which is much more efficient for larger ranges.

Using this approach with our example range from `low = 3` to `high = 7`, we've demonstrated that there are indeed 3 odd numbers, matching our earlier manual count.

Python Solution

```python
class Solution:
    def count_odds(self, low: int, high: int) -> int:
        # Calculate the number of odds between two numbers low and high (inclusive).
        # The formula ((high + 1) >> 1) calculates how many odd numbers are in
        # the range from 0 to high inclusive by dividing the endpoint by 2 and
        # rounding up, if necessary.
        # Similarly, (low >> 1) calculates how many odd numbers there are from
        # 0 up to but not including low.
        # The difference between these two quantities gives the count of odd numbers
        # in the range [low, high].
        return ((high + 1) >> 1) - (low >> 1)

```

Java Solution

```java
class Solution {
    // Function to count the number of odd numbers between low and high (inclusive)
    public int countOdds(int low, int high) {
        // ((high + 1) >> 1) calculates the number of odd numbers from 1 to high
        // (low >> 1) calculates the number of odd numbers from 1 to (low - 1)
        // Subtracting the two gives the number of odd numbers between low and high
        return ((high + 1) >> 1) - (low >> 1);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to count the number of odd numbers within a given range.
    int countOdds(int low, int high) {
        // Calculate the number of odd numbers before 'high' and 'low'.
        // Use bitwise right shift to divide by 2. Adding 1 to 'high' handles the case when 'high' is odd.
        int oddsBeforeHigh = (high + 1) >> 1;
        int oddsBeforeLow = low >> 1;

        // The number of odd numbers within the range is the difference
        // of the two calculated values.
        int countOfOdds = oddsBeforeHigh - oddsBeforeLow;

        // Return the count of odd numbers in the range.
        return countOfOdds;
    }
};

```

Typescript Solution

```typescript
// This function counts the number of odd numbers in the range [low, high].
// It leverages bitwise operations to efficiently perform the calculation.
function countOdds(low: number, high: number): number {
    // Increment the upper bound (`high`) by 1 and then shift it one bit to the right.
    // This effectively divides `high + 1` by 2, rounding down to the nearest integer.
    // This calculation returns the number of odd numbers from 0 up to `high`.
    const oddCountUpToHigh: number = (high + 1) >> 1;

    // Shift `low` one bit to the right to divide it by 2, rounding down to the nearest integer.
    // This calculation returns the number of odd numbers from 0 up to `low - 1`.
    const oddCountUpToLow: number = low >> 1;

    // Subtract the number of odds up to `low - 1` from the number of odds up to `high`
    // to get the count of odd numbers in the range [low, high].
    return oddCountUpToHigh - oddCountUpToLow;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(1)`. This is because the operations performed to calculate the number of odd numbers within the range — specifically, the additions, subtractions, and bit shifts — are constant time operations, regardless of the input size.

The space complexity of the code is also `O(1)`. There are a fixed number of variables used, which does not scale with the input size, thus only a constant amount of memory is used.