3021. Alice and Bob Playing Flower Game

Medium Math

Problem Description

Alice and Bob are playing a strategy game involving picking flowers arranged in a circle. The number of flowers between them in a clockwise direction is represented by `x`, and in an anti-clockwise direction by `y`. The game has specific rules:

1. Alice will take the first turn.
2. On each turn, the player can pick a flower from either the clockwise or anti-clockwise side.
3. If there are no flowers left, the player who took the last turn wins by capturing their opponent.

We need to find the total number of unique starting configurations (`(x, y)` pairs) that allow Alice to win the game guaranteeing these constraints:

- `x` is within the range `[1, n]`.
- `y` is within the range `[1, m]`.

The crux of the problem lies in finding combinations where Alice has a winning strategy based on the rules set forth.

Intuition

The key to solving this problem is recognizing the importance of the parity (odd or even) of the sum of `x` and `y`. Since Alice goes first, for her to win, she needs to ensure the final move is hers. This can only be guaranteed if `x + y` is odd since players alternate turns. In this case, if the sum is even, Bob will take the last turn and win. But if it's odd, Alice will.

Given this, we can split the possible ranges of `x` and `y` into odd and even counts:

- When `x` is odd, `y` must be even for the sum to be odd. The possible values for `x` would be half of `n`, but rounded up, because in a range of `n`, we're looking for the number of odd numbers (`(n + 1) // 2`). For `m` being the range for `y`, the formula to find even counts is `m // 2`.
- Conversely, if `x` is even, `y` must be odd to satisfy the odd sum condition. Here, the possible values for `x` would be half of `n`, but rounded down this time (`n // 2`), for even counts. For `y`, the similar logic of finding odd counts leads to `(m + 1) // 2`.

Calculating both these scenarios gives us all the possible combinations where Alice will win:

- Count of (odd `x`, even `y`) pairs plus
- Count of (even `x`, odd `y`) pairs

The product of these counts for each scenario gives us the total winning configurations for Alice, which is the solution to the problem.

Solution Approach

The solution to this game problem uses a straightforward mathematical approach to count the number of valid `(x, y)` pairs where Alice can win.

From our intuition, we know that Alice can only win if the total number of flowers (`x + y`) is odd. To ensure this, we need to count the number of odd possibilities for `x` and pair each with an even possibility for `y`, and vice versa.

Here's how the implementation works, using the provided solution code as an example:

1. Calculate the number of odd possibilities for `x`, which amounts to half of the range of `n` rounded up: `a1 = (n + 1) // 2`. The `//` operator in Python performs integer division, which discards any fractional part, and adding 1 before division accounts for rounding up.
2. Similarly, calculate the number of even possibilities for `y`, which is half of the range of `m` rounded down: `b1 = (m + 1) // 2`.
3. Next, calculate the number of even possibilities for `x`: `a2 = n // 2`.
4. And the number of odd possibilities for `y`: `b2 = m // 2`.
5. Finally, the total number of valid `(x, y)` pairs is the sum of the product of odds of `x` with evens of `y` and the product of evens of `x` with odds of `y`: `return a1 * b2 + a2 * b1`.

This approach avoids the need for loops or complex data structures. It simply uses arithmetic operations to evaluate the necessary counts to satisfy the game's winning condition for Alice. By scaling the counts of odd and even possibilities against each other, the solution effectively maps out all permutations that lead to Alice's victory.

Note the clever use of integer division and rounding to quickly determine the counts of odd and even numbers within the specified ranges. This technique reduces a potentially complex combinatorial problem to a few arithmetic operations, showcasing the power of mathematical analysis in algorithm design.

Example Walkthrough

Let's assume we have `n = 3` and `m = 4`. This means the range of `x` (clockwise) can be `[1, 2, 3]`, and the range of `y` (anti-clockwise) can be `[1, 2, 3, 4]`. We will illustrate how to find the total number of unique starting configurations `(x, y)` pairs that allow Alice to win the game under these conditions.

1. **Odd `x`, Even `y` Pairings:**
   - The total number of odd values `x` can take in the range `[1, 3]` is 2 (which are `1` and `3`). We calculate this using `a1 = (n + 1) // 2`, so `a1 = (3 + 1) // 2`, which equals 2.
   - The range of `n` is `[1, 2, 3, 4]`. The total number of even values that `y` can take is 2 (which are `2` and `4`). We calculate this using `b1 = m // 2`, so `b1 = 4 // 2`, which equals 2.
   - Pairing odd `x` values with even `y` values gives us four configurations: `(1, 2)`, `(1, 4)`, `(3, 2)`, and `(3, 4)`.
2. **Even `x`, Odd `y` Pairings:**
   - The range of `n` is `[1, 2, 3]`. The total number of even values `x` can take is 1 (which is `2`) calculated by `a2 = n // 2`, so `a2 = 3 // 2`, which equals `1`.
   - For `y`, the total number of odd values in the range `[1, 2, 3, 4]` is 2 (which are `1` and `3`). We calculate this using `b2 = (m + 1) // 2`, so `b2 = (4 + 1) // 2`, which equals 2.
   - Pairing even `x` values with odd `y` values gives us two configurations: `(2, 1)` and `(2, 3)`.

Adding the counts together:

- We have `a1 * b1` configurations from the first scenario, which translates to `2 * 2 = 4`.
- We also have `a2 * b2` configurations from the second scenario, which translates to `1 * 2 = 2`.

So, we add `4 + 2` to get a total of `6` unique starting configurations where Alice can win.

Therefore, for `n = 3` and `m = 4`, there are `6` unique game setups `(x, y)` where Alice has a guaranteed winning strategy.

Solution Implementation

Python

```python
class Solution:
    def flowerGame(self, n: int, m: int) -> int:
        # Calculate half of the garden, rounded up
        half_up_n = (n + 1) // 2
        half_up_m = (m + 1) // 2

        # Calculate half of the garden, rounded down
        half_down_n = n // 2
        half_down_m = m // 2

        # Calculate the result based on the game's logic, which seems to involve
        # multiplying the halves of the two dimensions, with one dimension always rounded up
        # and the other rounded down, then adding both results
        result = half_up_n * half_down_m + half_down_n * half_up_m

        return result
```

Java

```java
class Solution {
    // Method to compute the result of the flower game
    public long flowerGame(int petals, int players) {
        // a1 represents half the petals, rounded up
        long upperHalfPetals = (petals + 1) / 2;
        // b1 represents half the players, rounded up
        long upperHalfPlayers = (players + 1) / 2;
        // a2 represents half the petals, rounded down
        long lowerHalfPetals = petals / 2;
        // b2 represents half the players, rounded down
        long lowerHalfPlayers = players / 2;

        // Return the sum of cross-products of upper and lower halves of petals and players
        // This probably follows some game rule logic based on the distribution of petals and players
        return upperHalfPetals * lowerHalfPlayers + lowerHalfPetals * upperHalfPlayers;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to play the flower game
    // Parameters:
    // n - number of flowers along the length of the garden
    // m - number of flowers along the width of the garden
    // Returns the total number of distinct pairs that can be picked
    long long flowerGame(int n, int m) {
        // Calculate half of the length of the garden rounded up
        long long length_rounded_up = (n + 1) / 2;
        // Calculate half of the width of the garden rounded up
        long long width_rounded_up = (m + 1) / 2;

        // Calculate half of the length of the garden rounded down
        long long length_rounded_down = n / 2;
        // Calculate half of the width of the garden rounded down
        long long width_rounded_down = m / 2;

        // Total number of distinct pairs is calculated by
        // Adding the product of half the length (rounded up)
        // and half the width (rounded down)
        // with the product of half the length (rounded down)
        // and half the width (rounded up)
        // This is because a flower can be paired with any flower not on its row or column
        // We round up and down to cover both even and odd numbers of n and m
        long long total_pairs = length_rounded_up * width_rounded_down + length_rounded_down * width_rounded_up;

        // Return the total number of pairs
        return total_pairs;
    }
};
```

TypeScript

```typescript
// Defines a function that calculates a value based on the input parameters
// related to a hypothetical 'flower game'.
// @param playerCount - an integer representing the number of players in the game.
// @param flowerCount - an integer representing the number of flowers in the game.
// @returns an integer calculated based on a specific formula.
function flowerGame(playerCount: number, flowerCount: number): number {
    // Calculate the midpoints for the players and flowers, rounding down for even numbers
    // and rounding up for odd numbers, which is effectively a ceiling division by 2.
    const midpointPlayersCeil = Math.ceil(playerCount / 2);
    const midpointFlowersCeil = Math.ceil(flowerCount / 2);

    // Calculate the midpoints for the players and flowers, rounding down.
    const midpointPlayersFloor = Math.floor(playerCount / 2);
    const midpointFlowersFloor = Math.floor(flowerCount / 2);

    // Returns the result of the game's specific formula calculation.
    // The formula involves multiplying the opposing midpoints (ceiling midpoint of one group
    // with the floor midpoint of the other group) and adding the two products together.
    return (midpointPlayersCeil * midpointFlowersFloor) + (midpointPlayersFloor * midpointFlowersCeil);
}
```

```python
class Solution:
    def flowerGame(self, n: int, m: int) -> int:
        # Calculate half of the garden, rounded up
        half_up_n = (n + 1) // 2
        half_up_m = (m + 1) // 2

        # Calculate half of the garden, rounded down
        half_down_n = n // 2
        half_down_m = m // 2

        # Calculate the result based on the game's logic, which seems to involve
        # multiplying the halves of the two dimensions, with one dimension always rounded up
        # and the other rounded down, then adding both results
        result = half_up_n * half_down_m + half_down_n * half_up_m

        return result
```

Time and Space Complexity

The time complexity of the code is `O(1)` because all operations (`+`, `//`, `*`) are constant time operations that do not depend on the size of the input. The calculations are done in a fixed number of steps regardless of the values of `n` and `m`.

The space complexity is also `O(1)` as space usage does not scale with input size; only a fixed number of integer variables (`a1`, `b1`, `a2`, `b2`) are used, which occupy a constant amount of space.