1823. Find the Winner of the Circular Game

Medium Recursion Queue Array Math Simulation

Problem Description

In this game, `n` friends are sitting in a circle numbered from `1` to `n`. The game follows a set of rules where the friends count off from `1` to `k` in a clockwise direction, and each time the count reaches `k`, that person is removed from the circle. The counting process starts from the next person in the circle after the one who just got removed, and it continues until there's only one person left, who is declared the winner. The problem is to find out who will be the winner.

Intuition

The solution approach uses recursion to simulate the elimination process.

1. Base Case - If there is only one person (`n == 1`), they are the winner because there's nobody else to play the game with.
2. Recursive Case - When there are `n` people in the game, we can first find out who will be the winner among these `n` people without considering the counting and elimination process. To do this, we recursively solve the problem for `n-1` people to get the winner.
3. Since the game is circular, we need to handle the counting wrap around. This means if we reach the end of the circle while counting, we start from the beginning again. We accommodate this by using modulus operation (`% n`), which effectively resets the count when we reach `n` (the end of the circle).
4. After the recursive call, we adjust the position because we need to account for the shifted positions after each elimination. When the person at position `k` is eliminated, the next counting should start from `k+1`, but due to the removal, positions of everyone are shifted one place to the left. This gives the equation `ans = (k + self.findTheWinner(n - 1, k)) % n`.
5. The modulus might return `0`, but since the friends are numbered starting from `1`, if `ans == 0`, we should return `n` (because friend `n` would be in that position after the wrap around).

This recursive pattern continues until we reach the base case and then we unwind the recursion stack back to the original problem size (`n`), at each step applying the position adjustment. Eventually, we can find the winner of the game.

Solution Approach

The implementation uses a straightforward recursive function to simulate the elimination process in the game.

Here's a more detailed walkthrough of the code:

1. The function `findTheWinner` takes two arguments: `n` (number of friends) and `k` (the count at which friends are eliminated).
2. It first checks if `n` is equal to `1`, which is our base case. If this is true, it returns `1` since the last remaining friend is the winner by default.
3. The heart of the function is the recursive call. The function calls itself, but with `n - 1` instead of `n`. This simulates the game with one less friend, as one friend is supposed to get eliminated in every round.
4. The recursive call will eventually hit the base case and then return, unwinding the recursion. Every recursive step is calculating who would be the winner if the game started with one less player.

   The key here is to understand what `(k + self.findTheWinner(n - 1, k)) % n` is doing:

   - `self.findTheWinner(n - 1, k)`: This gives us the winner with one less player.
   - `k + self.findTheWinner(n - 1, k)`: This assumes that the elimination game starts right after the last winner. We add `k` because our counting includes the person we start with.
   - `(k + self.findTheWinner(n - 1, k)) % n`: We use the modular operator to wrap the counting around the circle since the circle is closed and we need to determine the actual position in the current round.
5. The variable `ans` holds the result of the modular arithmetic. If `ans` equals `0`, it means the position has wrapped around to the end of the circle, and hence the winner should be the nth friend. Otherwise, `ans` is already the correct position for the winner.
6. Finally, the function returns `ans`.

The algorithm does not use any additional data structures and simply relies on the call stack to perform the recursive calls. It follows the pattern of solving a smaller instance of the problem (with one less friend) and using that solution to build up to the full problem's solution.

This approach is efficient because it simplifies a seemingly complex circular problem into a series of linear steps using the properties of modular arithmetic to keep track of positions.

Example Walkthrough

Let's take a small example where `n = 5` friends are sitting in a circle and `k = 2`, which means every second person is eliminated. We will illustrate the solution approach using this example:

1. Imagine 5 friends sitting in a circle, numbered from 1 to 5. We start counting from friend 1 to friend 5 repeatedly.
2. In the first round, we start counting from friend 1. The counting goes `1 -> 2`. Since we are removing every second friend, friend 2 is eliminated.
3. Now we have friends 1, 3, 4, and 5 left. The next round starts from friend 3, and we count `3 -> 4`, so now friend 4 gets eliminated.
4. The circle is now down to friends 1, 3, and 5. The next round starts from friend 5 and the counting goes `5 -> 1`. This eliminates friend 1.
5. With friends 3 and 5 left, we start from friend 3 and count two places, arriving back at friend 3. Friend 3 is now eliminated.
6. We are left with friend 5, who is the winner of this game according to our rules.

Now let's walk through how our recursive algorithm would find this winner:

1. We call `findTheWinner(5, 2)`. Since `n` is not 1, the base case does not apply, and we proceed to the recursive case.
2. We make a recursive call `findTheWinner(5 - 1, 2)`, which is the same as `findTheWinner(4, 2)`. Think of this as "simulating" the game with one less friend (since friend 2 got eliminated).
3. Continuing recursively, we would call `findTheWinner(3, 2)`, `findTheWinner(2, 2)` and finally `findTheWinner(1, 2)`, which is our base case. Here `findTheWinner(1, 2)` would return 1 because only one friend is left.
4. Now, we unwind the recursion. When we called `findTheWinner(2, 2)`, based on our equation the result would be `(2 + 1) % 2` which equals 1. So, if the game had started with two people and we were eliminating every second person, friend 1 would win.
5. Back at `findTheWinner(3, 2)`, the result would be `(2 + 1) % 3` which equals 0. Since numbering starts at 1, we consider the last position, which is friend 3, as the winner when there are 3 participants.
6. Continuing this process, when we reach back to our original problem `findTheWinner(5, 2)`, we apply our equation `(2 + winnerFromPreviousStep) % 5`. Through our recursion, the combined rolls yield the same pattern that emerged when we manually simulated the game: friend 5 is the winner.
7. The function returns the value `5`, correctly identifying friend 5 as the winner of the game.

Through the recursive calls, the solution approach effectively tracks back from the base case to the original full circle, at each stage considering the winner if the circle had one less friend, and adjusting for elimination and wraparound using modular arithmetic.

Solution Implementation

Python

```python
class Solution:
    def findTheWinner(self, n: int, k: int) -> int:
        # Base case: If there is only one person, they are the winner.
        if n == 1:
            return 1

        # Recursive call to find the winner among (n-1) people.
        # The winner's position among n people is offset by k positions from
        # the position among (n-1) people.
        winner = (k + self.findTheWinner(n - 1, k)) % n

        # If modulo operation results in 0, the winner is at the nth position
        # because indexing starts from 1, not 0.
        # Otherwise, return the calculated winner position.
        return n if winner == 0 else winner
```

Java

```java
class Solution {
    // This method determines the winner in a game where n people
    // are sitting in a circle and elimination occurs at every kth person
    public int findTheWinner(int n, int k) {
        // Base case: if there is only one person, they are the winner
        if (n == 1) {
            return 1;
        }
        // Recursively find the winner for n - 1 people
        // and adjust the position with the offset k
        int winner = (findTheWinner(n - 1, k) + k) % n;

        // Java uses 0-based indexing but the problem assumes 1-based indexing.
        // Therefore, if the result is 0, we convert it to n (the last person),
        // as the modulo operation may result in zero.
        // Otherwise, we return the calculated position.
        return winner == 0 ? n : winner;
    }
}
```

C++

```cpp
class Solution {
public:
    // This function determines the winner in a game simulating the Josephus problem.
    // 'n' represents the number of players, and 'k' is the count after which a player is eliminated.
    int findTheWinner(int n, int k) {
        if (n == 1)
            return 1; // Base case: if there's only one player, they're the winner.

        // Recursive call to find out the winner among n-1 players.
        int winner = findTheWinner(n - 1, k);

        // Adjusting the winner index for the current round, considering 0-based indexing.
        int adjustedWinner = (winner + k) % n;

        // If the adjustedWinner is 0, it implies that the winner is the nth player.
        // Otherwise, return the index considering 1-based indexing.
        return adjustedWinner == 0 ? n : adjustedWinner;
    }
};
```

TypeScript

```typescript
// Node definition for a singly linked list node.
interface ListNode {
    value: number;
    next: ListNode | null;
}

// Function to create a new linked list node.
function createNode(value: number, next: ListNode | null = null): ListNode {
    return { value, next };
}

// Function to find the winner in the game.
function findTheWinner(n: number, k: number): number {
    // If k equals 1, the winner is the last person, because no one is eliminated.
    if (k === 1) {
        return n;
    }

    // Create a dummy node to help with circular list operations.
    let dummy: ListNode = createNode(0);
    let current = dummy;

    // Construct the circular linked list.
    for (let i = 1; i <= n; i++) {
        current.next = createNode(i);
        current = current.next;
    }
    // Complete the circular list by linking the last node back to the first node.
    current.next = dummy.next;

    // Start with the dummy node, which is right before the first node.
    current = dummy;
    let count = 0;

    // Loop until a single node is left in the circular linked list.
    // This node will point to itself.
    while (current.next !== current) {
        // Walk the list and increment the count.
        count++;
        // If the count reaches k, remove the kth node.
        if (count === k) {
            current.next = current.next.next;
            count = 0; // Reset count after elimination.
        } else {
            // Move to the next node.
            current = current.next;
        }
    }

    // Return the value of the surviving node.
    return current.value;
}

class Solution:
    def findTheWinner(self, n: int, k: int) -> int:
        # Base case: If there is only one person, they are the winner.
        if n == 1:
            return 1

        # Recursive call to find the winner among (n-1) people.
        # The winner's position among n people is offset by k positions from
        # the position among (n-1) people.
        winner = (k + self.findTheWinner(n - 1, k)) % n

        # If modulo operation results in 0, the winner is at the nth position
        # because indexing starts from 1, not 0.
        # Otherwise, return the calculated winner position.
        return n if winner == 0 else winner
```

Time and Space Complexity

The given Python code is a recursive implementation for finding a winner in a game where `n` people are sitting in a circle, and we eliminate every `k`-th person until only one person is left.

Time Complexity:

The time complexity of this recursive solution is `O(n)`. This is because the findTheWinner function is called recursively `n-1` times until `n` reaches `1`. In each recursive call, the computation performed is in constant time `O(1)`, so the total time complexity is a sum of `n-1` constant-time operations, which simplifies to `O(n)`.

Space Complexity:

The space complexity of this recursive solution is also `O(n)`. This is because the maximum depth of the recursion call stack will be `n-1`. Each call uses a constant amount of space, but since calls are not returned until the base case (n == 1) is met, it requires space proportional to the number of recursive calls, which is `n-1`. Thus, the space complexity simplifies to `O(n)`.