2682. Find the Losers of the Circular Game

Problem Description

In this game involving `n` friends sitting in a circle, the play involves passing a ball around in a sequential and expanding pattern. The central rule is that with each turn, the distance in steps between the passer and the receiver increases linearly by `k`. That is, during the first turn, the ball is passed `k` steps away; on the second turn, it's `2 * k` steps away, then `3 * k` steps away on the third turn, and so on.

The key point is that the counting is cyclic and wraps around the circle. Once the count reaches the last friend, it continues from the first friend again. This cycle continues until a friend receives the ball for the second time, which signals the end of the game. The friends who never got the ball even once are the losers of the game.

The objective of this problem is to determine which friends lose the game, i.e., never receive the ball. This list of losers should be returned in ascending order of their numbered positions.

Intuition

To approach the solution, we can simulate the process of the game since the game's rules are straightforward. If we follow the ball pass-by-pass, we can record who gets the ball and when the game ends.

Considering we are simulating the game's process, we have to track each friend who has received the ball. We can use an array, `vis`, of the same length as the number of friends (`n`). Each position in the array corresponds to a friend, where a value of `True` at an index means the friend at that position has received the ball, and `False` means they have not.

We keep two variables: `i`, which keeps the position of the current ball-holder, and `p`, which keeps the count of passes made thus far. Starting at position `0` (the `1st` friend), with each turn, we mark the current position as visited (`True`), then move `i` by `p * k` steps forward (using modulo `%` operator to wrap around the circle), and increment `p` by `1` for the next pass. This continues until we return to a previously visited position, which means a friend got the ball for the second time, and the game ends.

After the game ends, we know who has had at least one turn with the ball. The remaining unvisited positions in the `vis` array correspond to friends who never got the ball—these are our losers. We output their corresponding numbers (incrementing by 1 since friend numbering is 1-based), filtered in ascending order.

The presented solution encapsulates this simulation in the `circularGameLosers` method and produces the result efficiently.

Solution Approach

The solution uses an elementary algorithm that is a direct implementation of the rules of the game. It simulates the passing of the ball amongst friends in a circle. The primary data structure is a list in Python (`vis`) used to keep track of which friends have received the ball.

Here is a step-by-step walkthrough of the algorithm:

1. Initialize the `vis` list with `False` values since no one has received the ball initially.

   ```
   vis = [False] * n
   ```

2. Create two variables: `i` to represent the current position (starting from `0`, which corresponds to the `1st` friend) and `p` to represent the `p`th pass (starting from `1`).

   ```
   i, p = 0, 1
   ```

3. Use a `while` loop to keep passing the ball until a friend receives it a second time, which is indicated when we hit a `True` in the `vis` list at position `i`.

4. Inside the loop, mark the current position `i` as visited (`True`) in `vis`.

   ```
   vis[i] = True
   ```

5. Calculate the next position `i` to pass the ball to using the formula `(i + p * k) % n`, which accounts for the cyclic nature of passing the ball around the circle.

   ```
   i = (i + p * k) % n
   ```

6. Increment `p` by `1` to reflect the rules of the game for the next turn (`p` increments linearly each turn).

7. When the loop ends (upon a second receipt), iterate through the `vis` list to collect the indices (friend numbers) where the value is still `False`, which means these friends did not receive the ball even once.

8. For each unvisited index `i`, return `i + 1` because friend numbering is 1-based, not 0-based. Produce the final output list using a list comprehension.

   ```
   return [i + 1 for i in range(n) if not vis[i]]
   ```

The algorithm is a straightforward simulation and manages to keep time complexity at O(n) since each friend is visited at most once (every friend gets the ball at most once before the game ends). The use of modulo `%` for cyclic counting and a list comprehension for filtering out the losers contribute to the solution's conciseness and efficiency.

Example Walkthrough

Let's imagine a game with `n = 5` friends sitting in a circle, and we'll use `k = 2` to set the passing sequence. So, in this setup, the ball will be passed in increasing steps of 2 each turn.

1. Let's start by creating the `vis` array to track who has received the ball, initializing all to `False`.

   ```
   vis = [False, False, False, False, False]
   ```

2. We'll start the pass from the `1st` friend (indexed as `0`) and initiate our pass count `p` at `1`.

   ```
   i, p = 0, 1  # Starting from the 1st friend
   ```

3. As per the rules, the `1st` friend passes the ball `2` steps away since `p * k = 1 * 2 = 2`. Hence, the ball goes to the `3rd` friend (position `i = (0 + 1*2) % 5 = 2`).

   ```
   vis[0] = True  # Mark the 1st friend as having received the ball
   i = (0 + 1*2) % 5  # Ball goes to the 3rd friend
   p += 1  # Increment pass counter for the next turn
   ```

   Now, `vis = [True, False, False, False, False]`

4. Now on the second turn, `p` has incremented to `2`, so the `3rd` friend will pass it `4` steps ahead, which loops back around and lands on the `2nd` friend (position `i = (2 + 2*2) % 5 = 1`).

   ```
   vis[2] = True  # Mark the 3rd friend as having received the ball
   i = (2 + 2 * 2) % 5  # Ball goes to the 2nd friend
   p += 1
   ```

   Now, `vis = [True, False, True, False, False]`

5. On the third turn, we have `p = 3`, the ball is passed to `(1 + 3*2) % 5 = 0`, which is back to the `1st` friend. But since the `1st` friend has already received the ball once (`vis[0] = True`), the game ends here.

6. At this point, we can see from our `vis` array who hasn't received the ball:

   ```
   vis = [True, False, True, False, False]
   ```

   The unvisited positions correspond to the `2nd`, `4th`, and `5th` friends.

7. We then return the list of losers (those who never received the ball), incrementing the index by one for the correct numbering:

   ```
   losers = [i + 1 for i in range(5) if not vis[i]  # [2, 4, 5]
   ```

In this example, the friends who lose this game are the `2nd`, `4th`, and `5th` friends.

Python Solution

```python
from typing import List

class Solution:
    def circularGameLosers(self, n: int, k: int) -> List[int]:
        # Create a list to keep track of visited positions
        visited = [False] * n

        # Initialize index and step count
        current_index, step = 0, 1

        # Loop and mark visited positions
        while not visited[current_index]:
            visited[current_index] = True

            # Calculate the next index by moving 'k' steps forward
            # 'step' increases to simulate the change in the total number of players
            current_index = (current_index + step * k) % n
            step += 1

        # Return a list of non-visited indices (losers)
        # Note: compensate for 0-indexed list by adding 1 to each index
        return [index + 1 for index in range(n) if not visited[index]]
```

Java Solution

```java
class Solution {

    // Function that determines the positions that lose in the circular game
    public int[] circularGameLosers(int n, int k) {
        // Create an array to mark visited (eliminated) positions
        boolean[] visited = new boolean[n];
        int count = 0; // Count the number of visited (eliminated) positions
        // Loop through the array, marking off eliminated positions
        for (int index = 0, step = 1; !visited[index]; step++) {
            visited[index] = true; // Mark the current position as visited
            ++count; // Increase the count of visited positions
            // Calculate the next index based on the current index, step number and k
            // Use modulo n to wrap around the circle
            index = (index + step * k) % n;
        }

        // Initialize an array to store the positions that did not lose (were not visited)
        int[] losers = new int[n - count];
        // Fill the array with the positions of those who did not lose
        for (int i = 0, j = 0; i < n; ++i) {
            if (!visited[i]) {
                losers[j++] = i + 1; // Store the position (1-indexed) in the losers array
            }
        }

        return losers; // Return the array with the positions that lost in the game
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    // Function simulates a circular game and returns a vector of losers' positions (1-indexed)
    std::vector<int> circularGameLosers(int n, int k) {
        std::vector<bool> visited(n, false); // Vector to keep track of visited positions

        // Starting from the first position, mark visited positions
        for (int currentPosition = 0, stepMultiplier = 1;
             !visited[currentPosition];
             ++stepMultiplier) {
            visited[currentPosition] = true;

            // Compute the next position considering the steps taken
            currentPosition = (currentPosition + stepMultiplier * k) % n;
        }

        std::vector<int> losers; // Initialize a vector to hold the losers
        // Add unvisited positions (losers) to the vector. Positions are incremented by
        // 1 because the problem is likely using 1-indexed positions.
        for (int i = 0; i < n; ++i) {
            if (!visited[i]) {
                losers.push_back(i + 1);
            }
        }

        return losers; // Return the vector of losers
    }
};
```

Typescript Solution

```typescript
function circularGameLosers(numPlayers: number, skipCount: number): number[] {
    // Create an array to keep track of the players who have been eliminated.
    const isEliminated = new Array(numPlayers).fill(false);

    // An array to hold the losers of the game, i.e., the eliminated players.
    const losers: number[] = [];

    // Loop to eliminate players until there is a last player standing.
    // 'currentIndex' represents the current player in the loop, 'pass' increments each round to mimic the circular nature.
    for (let currentIndex = 0, pass = 1; !isEliminated[currentIndex]; pass++) {
        // Mark the current player as eliminated.
        isEliminated[currentIndex] = true;

        // Calculate the next player to be eliminated, wrapping around if necessary.
        currentIndex = (currentIndex + pass * skipCount) % numPlayers;
    }

    // Collect the indexes of the players who were not eliminated.
    for (let index = 0; index < isEliminated.length; index++) {
        if (!isEliminated[index]) {
            // Players are numbered starting from 1, hence adding 1 to the index.
            losers.push(index + 1);
        }
    }

    // Return the list of losers.
    return losers;
}
```

Time and Space Complexity

Time Complexity

The given code generates a sequence of numbers to simulate a circular game where players are eliminated in rounds based on the number `k`. The while loop runs until it finds a previously visited index, which signifies the end of one complete cycle.

The worst-case time complexity occurs when the loop runs for all `n` players before any player is visited twice, so it will run for `n` iterations. Inside the loop, the main operations are:

1. Setting the `visit[]` value to `True`,
2. Calculating the next index `i` using arithmetic operations, and
3. Incrementing the value `p`.

The above operations are O(1) for each iteration.

Therefore, considering all the operations inside the loop, the worst-case time complexity of this code is `O(n)`.

Space Complexity

The space complexity is determined by the amount of memory used in relation to the input size. The space used in the code comes from:

1. The `vis` list, which is initialized to the size of the number of players `n`. This list takes up `O(n)` space.
2. Variables `i`, and `p`, which take constant space, thus `O(1)`.

There are no additional data structures that grow with the input size, thus the overall space complexity of the code is `O(n)`.