1535. Find the Winner of an Array Game

Medium Array Simulation

Problem Description

In this problem, you are given an array `arr` consisting of distinct integers and an additional integer `k`. The problem simulates a game played with the elements of the array which involves repeated comparisons between the first two elements (`arr[0]` and `arr[1]`). In each round of the game, if `arr[0]` is greater than `arr[1]`, then `arr[0]` wins the round, remains at the first position, and the loser (`arr[1]`) is moved to the end of the array. Conversely, if `arr[1]` is greater, it takes the first position, and the previous `arr[0]` is moved to the end. Rounds continue until one of the integers wins `k` consecutive rounds. The task is to determine which integer will win the game under these rules.

It is important to note that the game will have a winner and that each integer in the array is unique.

Intuition

The intuition behind the solution is quite straightforward with a keen observation of the game's rules. Since the integers are distinct, there will be a 'strongest' integer in the array that will eventually defeat all others in comparison. As soon as this 'strongest' integer wins for the first time, it will keep winning against all the other integers. Thus, we can conclude that this 'strongest' integer only needs to win `k` times in a row or beat all other integers once to become the winner of the game.

Understanding this core principle, we can iterate through the array and maintain a count of consecutive wins for the current maximum integer we've found (`mx`). If the current maximum integer wins against the next challenger (`x`), we increment the win counter (`cnt`). If the challenger wins, it becomes the new maximum integer and the win counter resets to 1. This process continues until either the counter reaches `k` or we have checked all integers in the array. At that point, the current maximum (`mx`) is the winner of the game.

Solution Approach

The implementation of the solution leverages a simple iterative approach, which falls under the category of simulation algorithms. Simulation involves mimicking the operation of a real-world process or system over time.

The algorithm makes use of two variables, `mx` to keep track of the current maximum integer (the one that would potentially win the game) and `cnt` to count the number of consecutive rounds the current maximum integer has won. Initially, `mx` is set to the first element in the array (`arr[0]`), and `cnt` is set to zero as no rounds have been played yet.

The solution iterates over the elements of the array starting from the second element (`arr[1]`). For each element `x` in the array, the algorithm does the following:

- Check if the current maximum integer `mx` is smaller than the current element `x`.
  - If so, this means `x` wins this round, so `mx` is updated to `x`, and `cnt` is reset to 1 since `x` has now won 1 consecutive round.
  - Otherwise, `mx` has won the current round against `x`, so we increment `cnt` to reflect the additional consecutive win.

After each comparison, the algorithm checks if the win counter `cnt` has reached `k`. If it has, it means the current maximum integer `mx` has won `k` consecutive rounds, and we can break out of the loop.

The loop continues until either:

- The win counter `cnt` has reached `k` or;
- All elements in the array have been compared with `mx`.

Once the loop has finished (either by reaching the end of the array or by `cnt` reaching `k`), `mx` holds the value of the winning integer, which is then returned.

This solution works efficiently because it leverages the fact that the largest element in the array will inevitably end up at position `0` and stay there until the game is finished. As soon as it gets to position `0`, the maximum number of comparisons it will have to go through is `k` or the number of remaining elements in the array, whichever is smaller.

Here is the solution code enclosed within code ticks for clarity:

```
class Solution:
    def getWinner(self, arr: List[int], k: int) -> int:
        mx = arr[0]
        cnt = 0
        for x in arr[1:]:
            if mx < x:
                mx = x
                cnt = 1
            else:
                cnt += 1
            if cnt == k:
                break
        return mx
```

Through this code, we can see the application of algorithmic thinking, making use of efficient iteration and simple conditional checks to arrive at the solution. No additional data structures are needed, as we can solve the problem in a straight pass through the array, and the code runs in linear time relative to the size of the array.

Example Walkthrough

Let's consider a small example where the array `arr` is `[2, 1, 3, 5, 4]` and the additional integer `k` is 2. According to the rules, we need to find out which integer will win `k` or 2 consecutive rounds.

Initially, we set `mx = arr[0]`, which is `2`, and `cnt = 0`. Then we start iterating over the array from `arr[1]`.

1. Compare `mx` (2) with `arr[1]` (1):
   - `mx` is greater than `arr[1]`, so `mx` remains the same and `cnt` is incremented to 1.
   - The updated values are `mx = 2`, `cnt = 1`.
2. Next, we compare `mx` (2) with `arr[2]` (3):
   - `mx` is smaller than `arr[2]`, so `mx` becomes 3 and `cnt` is reset to 1.
   - The updated values are `mx = 3`, `cnt = 1`.
3. We then compare `mx` (3) with `arr[3]` (5):
   - Again, `mx` is smaller, so `mx` becomes 5, and `cnt` is reset to 1.
   - The updated values are `mx = 5`, `cnt = 1`.
4. Finally, we compare `mx` (5) with `arr[4]` (4):
   - `mx` is greater, so `mx` stays as 5, and `cnt` is incremented to 2.
   - The updated values are `mx = 5`, `cnt = 2`.

Since `cnt` has now reached `k` (which is 2), we stop here. The integer 5 has won 2 consecutive rounds, satisfying our condition for winning the game. Therefore, 5 will be returned as the winner of the game.

Through this example, we can observe that as soon as an integer wins against its immediate challenger, the counter is updated, and if the integer continues to win consecutively or the counter reaches `k`, this integer is considered the game winner. The code simulates the exact steps described above to extract the winner effectively and efficiently.

Solution Implementation

Python

```
from typing import List

class Solution:
    def getWinner(self, arr: List[int], k: int) -> int:
        # Initialize the maximum value with the first element in the array
        max_value = arr[0]
        # Initialize the counter for consecutive wins
        consecutive_wins = 0

        # Iterate over the array starting from the second element
        for value in arr[1:]:
            # If current value is greater than the maximum value found so far,
            # update the maximum value and reset consecutive wins counter
            if max_value < value:
                max_value = value
                consecutive_wins = 1
            else:
                # If current value is not greater, increment the consecutive wins counter
                consecutive_wins += 1

            # If the consecutive wins match the k value, break out of the loop
            if consecutive_wins == k:
                break

        # Return the maximum value which is the winner
        return max_value
```

Java

```
class Solution {

    public int getWinner(int[] arr, int k) {
        // Current maximum element found in the array.
        int currentMax = arr[0];

        // Initialize the count of consecutive wins for the current maximum element.
        int count = 0;

        // Loop through the array starting from the second element.
        for (int i = 1; i < arr.length; ++i) {
            // If the current maximum is less than the current element of the array,
            // update the current maximum to this new larger value
            // and reset the count of consecutive wins to 1.
            if (currentMax < arr[i]) {
                currentMax = arr[i];
                count = 1; // Reset count for the new maximum element.
            } else {
                // Otherwise, increment the count of consecutive wins for the current maximum.
                ++count;
            }

            // If the count of consecutive wins equals k, stop the loop as we found the winner.
            if (count == k) {
                break;
            }
        }

        // Return the element that has won k times in a row or is the largest in the array.
        return currentMax;
    }
}
```

C++

```
#include <vector>
using namespace std;

class Solution {
public:
    // Function to determine the winner of the game according to the given rules
    int getWinner(vector<int>& arr, int k) {
        int maxElement = arr[0]; // Initialize the maximum element found so far
        int winCount = 0; // Counter for the number of consecutive wins of 'maxElement'

        // Iterate through the array
        for (int i = 1; i < arr.size(); ++i) {
            // Check if the current element is greater than the maxElement found so far
            if (maxElement < arr[i]) {
                maxElement = arr[i]; // Update maxElement to the new maximum
                winCount = 1; // Reset win counter since we have a new maxElement
            } else {
                // If maxElement is still greater, increase its winCount
                ++winCount;
            }
            // If the winCount reaches k, current maxElement is the winner, break the loop
            if (winCount == k) {
                break;
            }
        }

        // Return the element that has won k times in a row or is the maximum element found
        return maxElement;
    }
};
```

TypeScript

```
function getWinner(arr: number[], k: number): number {
    // Initialize the maximum number found so far to the first element of the array
    let maxNumber = arr[0];
    // Initialize a counter to track the number of consecutive wins
    let winCount = 0;

    // Loop through the array starting from the second element
    for (const current of arr.slice(1)) {
        // If the current number is greater than the maxNumber, update maxNumber
        // and reset the winCount since we have a new "leader"
        if (maxNumber < current) {
            maxNumber = current;
            winCount = 1;
        } else {
            // If the current number is not greater, increment the winCount
            ++winCount;
        }

        // If the winCount has reached the required number of consecutive wins, break the loop
        if (winCount === k) {
            break;
        }
    }

    // Return the number which reached the required k consecutive wins (or if none did, the maxNumber)
    return maxNumber;
}
```

```
from typing import List

class Solution:
    def getWinner(self, arr: List[int], k: int) -> int:
        # Initialize the maximum value with the first element in the array
        max_value = arr[0]
        # Initialize the counter for consecutive wins
        consecutive_wins = 0

        # Iterate over the array starting from the second element
        for value in arr[1:]:
            # If current value is greater than the maximum value found so far,
            # update the maximum value and reset consecutive wins counter
            if max_value < value:
                max_value = value
                consecutive_wins = 1
            else:
                # If current value is not greater, increment the consecutive wins counter
                consecutive_wins += 1

            # If the consecutive wins match the k value, break out of the loop
            if consecutive_wins == k:
                break

        # Return the maximum value which is the winner
        return max_value
```

Time and Space Complexity

The given Python code implements a function to determine the winner in a game played with an array and a number `k`. The winner is the first integer that wins `k` consecutive rounds against all the subsequent integers it faces in the list.

Time Complexity:

The time complexity of the function is `O(n)`, where `n` is the length of the input list `arr`. This is because the code iterates through the list once, comparing each element with the current maximum (`mx`) until either an element has won `k` times consecutively or we have reached the end of the list.

The worst-case scenario occurs when `k` is greater than the length of the list, which would result in a complete single iteration through the list. Each comparison takes `O(1)` time, so going through the list is `O(n)`.

Space Complexity:

The space complexity of the function is `O(1)`. Only a constant amount of extra space is used, which includes variables for the current maximum (`mx`), the current count (`cnt`), and any other temporary variables used within the loop (like `x`). This does not scale with the input size, so it remains constant.