2526. Find Consecutive Integers from a Data Stream

Medium Design Queue Hash Table Counting Data Stream

Leetcode Link

Problem Description

The problem provides a scenario where we have to manage a data structure for a stream of integers. The primary goal of this data structure is to check if the last 'k' integers received in the stream are all the same values as a specified 'value'. Two main operations need to be supported:

1. Initialization: The `DataStream` class should be initialized with two integers, `value` and `k`. There is no stream yet, so we start with an empty one.
2. Addition & Check: The class should provide a method `consec` which takes an integer `num` and adds it to the stream. The method should return `true` if the last 'k' integers in the stream are the same as `value`. If there are fewer than 'k' integers in the stream, or the last 'k' integers are not all equal to `value`, the method should return `false`.

The puzzle lies in efficiently managing the stream and checking the condition with each addition while considering the following:

- The stream is potentially endless, so storing all integers is impractical.
- The check needs to be performed only on the last 'k' elements.

Intuition

The intuition behind the solution is to maintain only the necessary information to determine if the last 'k' integers are equal to `value`. In this case, we avoid storing the entire stream, which is a crucial optimization given the infinite potential size of the stream.

Here's the thought process for arriving at the solution:

1. Keep track of the count of consecutive integers equal to `value` as they are added to the stream (`self.cnt`).
2. The counter must be reset to zero if the incoming number (`num`) is not equal to `value`.
3. Each time a new integer is added to the stream via `consec` method, there are two possibilities:
   - If the new integer is equal to `value`, increment the counter.
   - If the new integer is not equal to `value`, reset the counter to zero.
4. After adding an integer and updating the counter, check if the count of consecutive `value` is at least `k`. This can be done by comparing `self.cnt` with `k`.
5. Return `true` if `self.cnt` is greater than or equal to `k`, signifying that the last 'k' integers are equal to `value`; otherwise, return `false`.

This algorithmic approach ensures that we are using only a constant amount of additional space regardless of the size of the input stream, and that each addition is processed in constant time.

Solution Approach

The implementation uses a simple class `DataStream` with the following components:

1. Initialization (in the `__init__` method):
   - When a `DataStream` object is created, it is initialized with two instance variables:
     - `self.val`: Stores the target value we want to compare the integers in the stream against. It's set to the `value` parameter passed to the constructor.
     - `self.k`: Stores the count 'k' that determines the number of consecutive integers we need to check. It's set to the `k` parameter passed to the constructor.
   - An additional instance variable `self.cnt` is initialized to zero. This variable keeps track of the number of consecutive integers that match `self.val`.
2. Addition & checking (in the `consec` method):
   - The method takes a single integer `num` as its input.
   - If `num` is the same as `self.val`, increment `self.cnt` since we have another occurrence; `self.cnt += 1`.
   - If `num` is not the same as `self.val`, we no longer have a consecutive sequence; reset `self.cnt` to zero; `self.cnt = 0`.
   - Finally, return if `self.cnt` is at least `self.k`, indicating that the last 'k' integers added were all equal to `self.val`. This is a boolean statement: `return self.cnt >= self.k`.

The key here is the counter `self.cnt`; it is smartly used to keep a running total of consecutive integers equal to `self.val` without the need to maintain the whole stream, hence optimizing both space and time complexity.

This approach follows a common pattern known as the sliding window, although the window is implicit in this case since we are not actively maintaining a list of the last 'k' elements. Instead, we're keeping track of how many of those elements meet our criteria (being equal to `self.val`). If at any point a number does not match `self.val`, the counter is reset, signifying the start of a new window.

Example Walkthrough

Let's consider an example to illustrate the solution approach. We want to track whether the last `k` numbers added to the stream are equal to a certain value `value`. For this example, say that `value = 5` and `k = 3`. We'll perform the following sequence of operations on the data structure:

1. Initialize the `DataStream` object with `value = 5` and `k = 3`.

   At this point, `self.val = 5`, `self.k = 3`, and `self.cnt = 0`. The stream is empty.
2. Add the integer `5` to the stream by calling `consec(5)`.
   - Since `5` is equal to `self.val`, `self.cnt` becomes `1`.
   - There are not yet `k` elements in the stream, so the method returns `false`.
3. Add the integer `5` to the stream by calling `consec(5)` again.
   - Again, `5` is equal to `self.val`, and now `self.cnt` is incremented to `2`.
   - There are still fewer than `k` elements matching `self.val`, so it returns `false`.
4. Add the integer `3` to the stream by calling `consec(3)`.
   - Since `3` is not equal to `self.val`, `self.cnt` is reset to `0`.
   - The method returns `false` because the last three values are not all `5`.
5. Next, add three consecutive `5` integers by calling `consec(5)` thrice.
   - For the first call, since `5` equals `self.val`, `self.cnt` is now `1`.
   - The second call increments `self.cnt` to `2`.
   - With the third call, `self.cnt` will become `3`.
   - At this point, `self.cnt` is equal to `self.k` (both are `3`), which means the last three added integers are all `5`, so the method returns `true`.

To sum up, the `DataStream` class managed the stream efficiently. It only kept track of the count of consecutive integers that were equal to `self.val`, using `self.cnt`. The class did not store all the integers in the stream, thus saving space, and it performed each `consec` addition operation in constant time.

Python Solution

```python
class DataStream:
    def __init__(self, value: int, k: int):
        # Initialize with a fixed value and consecutive count threshold k
        self.current_value = value
        self.threshold_k = k
        self.consecutive_count = 0  # Counter for consecutive occurrences

    def consec(self, num: int) -> bool:
        # Resets the counter if the new number is not equal to the current value,
        # otherwise increments the counter.
        if num != self.current_value:
            self.consecutive_count = 0
        else:
            self.consecutive_count += 1

        # Return True if the count of consecutive numbers has reached or
        # surpassed the threshold k
        return self.consecutive_count >= self.threshold_k

# Example of how to use the DataStream class:
# obj = DataStream(value, k)
# result = obj.consec(num)
```

Java Solution

```java
/**
 * The DataStream class provides a way to track consecutive appearances of a specific value in a stream of integers.
 * It allows checking if the value has appeared consecutively at least 'k' times after each new number is observed.
 */
class DataStream {
    // cnt tracks the current consecutive count of the value.
    private int count;
    // val stores the value to track for consecutive appearances.
    private int value;
    // k represents the threshold for consecutive appearances.
    private int k;

    /**
     * Constructor to initialize the DataStream with a specific value to track and the threshold of consecutive appearances.
     *
     * @param value The value to track for consecutive appearance.
     * @param k     The threshold for consecutive appearances required to return true.
     */
    public DataStream(int value, int k) {
        this.value = value;
        this.k = k;
        count = 0; // Initialize the count to 0.
    }

    /**
     * Checks if the given number is equal to the tracked value and updates the consecutive count.
     * If the count matches or exceeds 'k', returns true; otherwise, resets count and returns false.
     *
     * @param num The next number in the data stream to compare against the tracked value.
     * @return   True if the tracked value has appeared at least 'k' times consecutively, otherwise false.
     */
    public boolean consec(int num) {
        // If num is equal to the value we're tracking, increment the count. Otherwise, reset the count to 0.
        count = (num == value) ? count + 1 : 0;
        // If the count is greater than or equal to k, return true, as we have seen the value 'k' times consecutively.
        return count >= k;
    }
}

/**
 * Usage:
 * DataStream obj = new DataStream(value, k);
 * boolean result = obj.consec(num);
 */
```

C++ Solution

```cpp
class DataStream {
public:
    // Constructor to initialize the DataStream object with a starting value and the threshold k.
    DataStream(int value, int k) : currentValue(value), threshold(k), consecutiveCount(0) {
        // The current consecutive count is initialized to 0.
    }

    // Function that checks if the current number extends the consecutive sequence of a specific value.
    // It returns true if the sequence has reached a length of k or more.
    bool consec(int num) {
        // If the current number is the same as the value we are tracking,
        // increment the consecutive count. Otherwise, reset the count to 0.
        consecutiveCount = (num == currentValue) ? consecutiveCount + 1 : 0;

        // Check if the consecutive count has reached the threshold 'k'.
        // If it has, return true. Otherwise, return false.
        return consecutiveCount >= threshold;
    }

private:
    int consecutiveCount;    // Count of how many consecutive times 'currentValue' has appeared.
    int currentValue;        // The value we are tracking for consecutive appearances.
    int threshold;           // The threshold for how many consecutive appearances are needed.
};

/**
 * Your DataStream object will be instantiated and called as such:
 * DataStream* obj = new DataStream(value, k);
 * bool param_1 = obj->consec(num);
 */
```

Typescript Solution

```typescript
// These variables replace private class properties.
let currentValue: number;
let threshold: number;
let consecutiveCount: number;

/**
 * Initializes the data stream with an initial value and a threshold for consecutive numbers.
 * @param {number} value - The initial value of the data stream.
 * @param {number} k - The threshold number of consecutive values.
 */
function initializeDataStream(value: number, k: number): void {
    currentValue = value;
    threshold = k;
    consecutiveCount = 0;
}

/**
 * Evaluates whether the given number has appeared consecutively at least 'k' times.
 * @param {number} num - The number to check against the current value in the data stream.
 * @returns {boolean} - True if 'num' has appeared consecutively at least 'k' times; otherwise, false.
 */
function consec(num: number): boolean {
    if (currentValue === num) {
        consecutiveCount += 1;
    } else {
        consecutiveCount = 0;
    }

    return consecutiveCount >= threshold;
}

// Example usage:
// initializeDataStream(value, k);
// const isConsecutive = consec(num);
```

Time and Space Complexity

Time Complexity

The `consec` method of the `DataStream` class has a time complexity of $O(1)$. This constant time complexity arises because within the `consec` method, all operations (including comparison, conditional operation, and arithmetic addition) are basic and execute in constant time, independent of the input size.

Space Complexity

The space complexity of the `DataStream` class is $O(1)$. There are a fixed number of instance variables (`val`, `k`, `cnt`) that do not scale with the size of the input. Hence, the amount of memory used does not increase as the size of the data stream increases.