2526. Find Consecutive Integers from a Data Stream

Counting

Hash Table

Problem Description

Design

Queue

Medium

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: Initialization: The DataStream class should be initialized with two integers, value and k. There is no stream yet, so we start

Data Stream

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

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

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

- Intuition

stream.

Here's the thought process for arriving at the solution: Keep track of the count of consecutive integers equal to value as they are added to the stream (self.cnt).

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.

input stream, and that each addition is processed in constant time.

- If the new integer is not equal to value, reset the counter to zero. After adding an integer and updating the counter, check if the count of consecutive value is at least k. This can be done by

The counter must be reset to zero if the incoming number (num) is not equal to value.

false. This algorithmic approach ensures that we are using only a constant amount of additional space regardless of the size of the

Return true if self.cnt is greater than or equal to k, signifying that the last 'k' integers are equal to value; otherwise, return

Solution Approach

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

• 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:

• 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.

self.val.

new window.

constructor.

comparing self.cnt with k.

• An additional instance variable self.cnt is initialized to zero. This variable keeps track of the number of consecutive integers that match

the need to maintain the whole stream, hence optimizing both space and time complexity.

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

 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.

return self.cnt >= self.k.

operations on the data structure:

Addition & checking (in the consec method):

• The method takes a single integer num as its input.

- 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
- 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
- **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

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

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. Add the integer 5 to the stream by calling consec(5).

Again, 5 is equal to self.val, and now self.cnt is incremented to 2.

Python

class DataStream:

• There are still fewer than k elements matching self.val, so it returns false. 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.

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

 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.

• The second call increments self.cnt to 2.

consec addition operation in constant time.

def __init__(self, value: int, k: int):

self.current_value = value

def consec(self, num: int) -> bool:

surpassed the threshold k

Example of how to use the DataStream class:

public DataStream(int value, int k) {

count = 0; // Initialize the count to 0.

this.value = value;

this.k = k;

let consecutiveCount: number;

currentValue = value;

consecutiveCount = 0;

threshold = k;

/**

/**

return self.consecutive_count >= self.threshold_k

// cnt tracks the current consecutive count of the value.

// k represents the threshold for consecutive appearances.

// val stores the value to track for consecutive appearances.

* @param value The value to track for consecutive appearance.

self.threshold_k = k

For the first call, since 5 equals self.val, self.cnt is now 1.

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.

Add the integer 5 to the stream by calling consec(5) again.

To sum up, the DataStream class managed the stream efficiently. It only kept track of the count of consecutive integers that were

The method returns false because the last three values are not all 5.

Next, add three consecutive 5 integers by calling consec(5) thrice.

Solution Implementation

Initialize with a fixed value and consecutive count threshold k

self.consecutive_count = 0 # Counter for consecutive occurrences

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

* It allows checking if the value has appeared consecutively at least 'k' times after each new number is observed.

The threshold for consecutive appearances required to return true.

* Constructor to initialize the DataStream with a specific value to track and the threshold of consecutive appearances.

```
/**
* The DataStream class provides a way to track consecutive appearances of a specific value in a stream of integers.
```

private int count;

private int value;

private int k;

* @param k

/**

*/

class DataStream {

obj = DataStream(value, k)

result = obj.consec(num)

Java

*/

```
/**
```

```
* 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.
                 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++
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:
                            // Count of how many consecutive times 'currentValue' has appeared.
   int consecutiveCount;
   int currentValue;
                            // The value we are tracking for consecutive appearances.
                            // The threshold for how many consecutive appearances are needed.
   int threshold;
/**
* Your DataStream object will be instantiated and called as such:
* DataStream* obj = new DataStream(value, k);
 * bool param_1 = obj->consec(num);
 */
TypeScript
// These variables replace private class properties.
let currentValue: number;
let threshold: number;
```

```
* 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);
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)
Time and Space Complexity
Time Complexity
```

* Initializes the data stream with an initial value and a threshold for consecutive numbers.

* @param {number} value - The initial value of the data stream.

function initializeDataStream(value: number, k: number): void {

* @param $\{number\}$ k - The threshold number of consecutive values.

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.