346. Moving Average from Data Stream

Problem Description

In the given problem, we are required to calculate the moving average of a stream of integers with a specified window size. The moving average is simply the average of the most recent `size` numbers in the stream; as a new number comes in, the oldest number in the window gets removed.

To solve this, we create a class `MovingAverage` that accomplishes two tasks:

1. It initializes the window with a specific size, and to track the numbers within the window, it stores them in an array.
2. It calculates the next average each time a new value is added to the window. If the window is full, it replaces the oldest number with the new value. If the window has not yet filled to its specified size, it simply adds the new value.

The result is that at any given time, we can obtain the moving average of the last `size` elements, efficiently and without having to sum the entire set of values each time.

Intuition

The intuition behind the solution is to maintain a dynamic sum of the integers within the current window so that we don't have to sum all the integers each time we want to compute the average; this would be inefficient, especially for large window sizes.

The method is to:

1. Store the numbers in a fixed-size array, where the size is equal to the given window size.
2. Keep track of a running sum, `s`, of the numbers in the current window.
3. Use a counter, `cnt`, to determine which element in our array is the oldest and should be replaced with the new incoming value.

Each time a new value comes in:

- We compute the index for the oldest value by taking `cnt` modulo the window size.
- We update the running sum `s` by subtracting the value that's being replaced and adding the new value.
- We then update the value at the computed index with the new value.
- Increment the count `cnt`.

When calculating the moving average, we divide `s` by the window size. However, until the window fills up for the first time, our window size is effectively the number of elements we have received so far. So we take the minimum of the counter `cnt` and the window size to know by what we should divide our running sum.

This approach ensures an efficient and scalable way to calculate the moving average as the stream progresses, with a time complexity of O(1) for each call to `next`.

Solution Approach

The `MovingAverage` class provides an elegant solution for calculating the moving average of a stream of integers using a fixed-size sliding window. The code snippet implements this using a circular array pattern, a running sum, and a counter.

Here's a step-by-step explanation of the implementation:

- As part of the class initialization (`__init__` method), an array `arr` is created with a length equal to the size of the window, initially filled with zeros. This array will serve as the circular buffer. Additionally, variables `s` (running sum) and `cnt` (counter) are initialized. Variable `s` will keep track of the sum of the integers in the current window, while `cnt` will be used to count the elements that have been processed.
- The `next` method takes an integer `val` as input, which represents the next number in the stream. It calculates the index where the new value should be inserted using the modulo operation `idx = self.cnt % len(self.arr)`. This ensures that once the array is filled, we overwrite the oldest value, thereby maintaining the size of the sliding window.
- The running sum is updated by subtracting the value at `arr[idx]` (which is the oldest value in the window or a zero initially) and adding the new val to it: `self.s += val - self.arr[idx]`.
- The value of `val` is then stored in the array at index `idx`: `self.arr[idx] = val`.
- Before calculating the average, `cnt` is incremented by one: `self.cnt += 1`.
- To compute the moving average, the running sum `s` is divided by the minimum of `cnt` and the window size `len(self.arr)`. Using `min(self.cnt, len(self.arr))` ensures that, before the window has been completely filled for the first time, we only divide by the actual number of elements that have been added, which will be less than the maximum window size.

The code efficiently implements this approach, ensuring that regardless of the stream length or window size, the time complexity of each call to `next(val)` remains constant, O(1). This is due to the fixed array size and the simple arithmetic operations involved in updating the running sum and calculating the average.

The data structures used here are:

- An array (`self.arr`) to store the most recent values up to the window size.
- Variables (`self.s` and `self.cnt`) to store the running sum and the count.

The main algorithmic pattern used is a circular array to maintain the sliding window efficiently.

Example Walkthrough

To illustrate the solution approach, let's use a small example where the window size for the moving average is set to 3, and we receive the following stream of integers: [5, 10, 15, 20].

Let's walk through the process:

1. Initialize the `MovingAverage` class with a window size of 3.
2. The array `arr` is initialized with `[0, 0, 0]`, `s` (running sum) is set to `0`, and `cnt` (counter) is set to `0`.

Now, we start adding numbers to the stream:

- First number is 5:
  - `cnt % 3 = 0 % 3 = 0`. We update `arr` to `[5, 0, 0]`.
  - Update running sum `s` (we subtract `arr[0]` (which is `0`) and add `5`, so `s` becomes `5`.
  - Now, the average is `5 / 1 = 5` as we have only one number (5) in our window.
- Second number is 10:
  - `cnt % 3 = 1 % 3 = 1`. We update `arr` to `[5, 10, 0]`.
  - Update running sum `s`: we subtract `arr[1]` (which is `0`) and add `10`, so `s` becomes `15`.
  - The average is `15 / 2 = 7.5` because we now have two numbers (5 and 10) in our window.
- Third number is 15:
  - `cnt % 3 = 2 % 3 = 2`. We update `arr` to `[5, 10, 15]`.
  - Update running sum `s`: we subtract `arr[2]` (which is `0`) and add `15`, so `s` becomes `30`.
  - The average is `30 / 3 = 10` as we have three numbers in our window.
- Fourth number is 20:
  - `cnt % 3 = 3 % 3 = 0` (this index wraps around since the window size is reached).
  - We update `arr`: before it was `[5, 10, 15]` and now the oldest value (5) will be replaced by 20, so `arr` becomes `[20, 10, 15]`.
  - Update running sum `s`: we subtract `arr[0]` (which was `5`) and add `20`, so `s` becomes `30 + 20 - 5 = 45`.
  - The average is `45 / 3 = 15` because the window is full, and we only consider the last three numbers (20, 10, and 15).

This example clearly demonstrates how the `MovingAverage` class efficiently updates and computes the new moving average as each number in the stream is added, by maintaining a sliding window of the most recent values.

Python Solution

```python
class MovingAverage:
    def __init__(self, size: int):
        """
        Initialize a MovingAverage object with the specified size.

        :param size: The size of the moving window for which the average
                     should be calculated.
        """
        self.queue = [0] * size  # Initialize a fixed-size list to store the values.
        self.window_sum = 0  # Sum of elements currently within the moving window.
        self.count = 0  # Total count of values processed, used to calculate the index.

    def next(self, val: int) -> float:
        """
        Calculate the moving average by adding a new value and
        removing the oldest one from the sum if necessary.

        :param val: The new value to be added in the moving average calculation.
        :return: The current moving average after adding the new value.
        """
        # Calculate the index of the oldest value based on the count of elements processed modulo the window size.
        index = self.count % len(self.queue)

        # Update the sum by adding the new value and subtracting the value that is being replaced.
        self.window_sum += val - self.queue[index]

        # Replace the oldest value in the queue with the new value.
        self.queue[index] = val

        # Increment the count of values processed.
        self.count += 1

        # Calculate and return the moving average.
        # Use min to get the correct count of elements if the window is not fully filled.
        return self.window_sum / min(self.count, len(self.queue))

# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)
```

Java Solution

```java
class MovingAverage {
    private int[] window; // Array to hold the values for the moving average calculation
    private int sum;      // Sum of the elements currently in the window
    private int count;    // The number of elements that have been inserted

    /**
     * Constructor to initialize the MovingAverage with a specific size.
     *
     * @param size The size of the window for the moving average
     */
    public MovingAverage(int size) {
        window = new int[size];
    }

    /**
     * Inserts a new value into the MovingAverage and returns the current average.
     *
     * @param val The new value to be added to the moving average
     * @return The current moving average after inserting the new value
     */
    public double next(int val) {
        int index = count % window.length; // Find the index to insert the new value
        sum -= window[index];              // Subtract the value that will be replaced from the sum
        sum += val;                        // Add the new value to the sum
        window[index] = val;               // Replace the old value with the new value in the window
        count++;                           // Increment the count of total values inserted

        // Calculate and return the moving average.
        // The denominator is the smaller of the count of values inserted and the window size.
        return (double) sum / Math.min(count, window.length);
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

// Class to calculate the moving average of the last N numbers in a stream.
class MovingAverage {
public:
    // Constructor initializes the circular buffer 'values_' with the given 'size' and resets the sum 'sum_' to 0.
    MovingAverage(int size) : size_(size), sum_(0), count_(0), values_(size, 0) {}

    // Function to add a new 'value' to the stream and compute the updated moving average.
    double next(int value) {
        // Determine the index in the circular buffer to update
        int index = count_ % size_;

        // Update sum: add the new value and subtract the value that is being replaced in the buffer
        sum_ += value - values_[index];

        // Assign the new value to the appropriate position in the circular buffer
        values_[index] = value;

        // Increase the count of the total number of values processed
        ++count_;

        // Calculate the average based on the minimum of 'count_' and 'size_'
        // This accounts for the case where fewer than 'size_' values have been processed
        return static_cast<double>(sum_) / std::min(count_, size_);
    }

private:
    int size_;                  // Size of the window for the moving average
    int sum_;                   // Sum of the last 'size_' values
    int count_;                 // Total number of values that have been processed
    std::vector<int> values_;   // Circular buffer to hold the last 'size_' values
};

// Usage example (not part of the MovingAverage class)
int main() {
    MovingAverage obj = MovingAverage(size);
    double avg1 = obj->next(val1); // Adds val1 to the stream and calculates moving average
    double avg2 = obj->next(val2); // Adds val2 to the stream and calculates moving average
    // ...
    delete obj; // Clean up the allocated object
    return 0;
}
*/
```

Typescript Solution

```typescript
// The size of the window for the moving average.
let size: number;

// The sum of the last 'size' values.
let sum: number = 0;

// The total number of values that have been processed.
let count: number = 0;

// Array to hold the last 'size' values, implementing circular buffer functionality.
let values: number[];

// Initialize the variables with a given size for the moving average calculation.
function initializeMovingAverage(windowSize: number): void {
    size = windowSize;
    sum = 0;
    count = 0;
    values = new Array(windowSize).fill(0);
}

// Add a new value to the stream and compute the updated moving average.
function next(value: number): number {
    // Determine the index in the circular buffer to update.
    let index = count % size;

    // Update sum: add the new value and subtract the value that is being replaced in the buffer.
    sum += value - values[index];

    // Assign the new value to the appropriate position in the circular buffer.
    values[index] = value;

    // Increase the count of the total number of values processed.
    count++;

    // Calculate the average based on the minimum of 'count' and 'size'.
    // This accounts for the case where fewer than 'size' values have been processed.
    return sum / Math.min(count, size);
}

// Usage example (Note: 'initializeMovingAverage' must be called before 'next')
/*
initializeMovingAverage(size);
let avg1 = next(value1); // Adds value1 to the stream and calculates moving average
let avg2 = next(value2); // Adds value2 to the stream and calculates moving average
// ...
*/
```

Time and Space Complexity

The given Python class `MovingAverage` implements a moving average with a fixed-size window. Its complexity characteristics are:

Time Complexity

Initialization (`__init__`): The constructor initializes an array with a length equal to the size, which takes $O(size)$ time as it must allocate space for `size` integers and fill them with zeros. It also initializes two integer variables, `self.s` and `self.cnt`, which is done in constant time, $O(1)$. Hence, the time complexity for initialization is $O(size)$.

Next (`next`): Each `next` operation involves a modulo operation to find the index, an addition, and a subtraction, all of which are $O(1)$ operations. Updating the sum `self.s` by adding the new `val` and subtracting the value overwritten in the array takes constant time. Since the array `self.arr` is fixed in size, no extra time for resizing or shifting elements is required. As such, the `next` method has a time complexity of $O(1)$.

Space Complexity

Space Complexity: The class maintains a fixed-size array, `self.arr`, to store the last `size` elements, which consumes $O(size)$ space. The other variables, `self.s` and `self.cnt`, consume a constant space, $O(1)$. Thus, the total space complexity for the `MovingAverage` class is $O(size)$, as this is the dominant term.