<https://leetcode.ca/all/346.html>

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Example:

MovingAverage m = new MovingAverage(3);

m.next(1) = 1

m.next(10) = (1 + 10) / 2

m.next(3) = (1 + 10 + 3) / 3

m.next(5) = (10 + 3 + 5) / 3

**Attempt 1: 2024-01-05**

**Solution 1: Queue (10min)**

import java.util.\*;

public class MovingAverage {

int maxSize = 0;

Queue<Integer> q;

double presum = 0.0;

public MovingAverage(int size) {

this.maxSize = size;

this.q = new LinkedList<>();

}

public double next(int val) {

if(maxSize == q.size()) {

presum -= q.remove();

}

q.offer(val);

presum += val;

return presum / q.size();

}

public static void main(String[] args) {

MovingAverage m = new MovingAverage(3);

double a = m.next(1);

double b = m.next(10);

double c = m.next(3);

double d = m.next(5);

System.out.println(a);

System.out.println(b);

System.out.println(c);

System.out.println(d);

}

}

Time Complexity: O(1)

Space Complexity: O(N)

**Refer to**

<https://grandyang.com/leetcode/346/>

这道题定义了一个MovingAverage类，里面可以存固定个数字，然后我们每次读入一个数字，如果加上这个数字后总个数大于限制的个数，那么我们移除最早进入的数字，然后返回更新后的平均数，这种先进先出的特性最适合使用队列queue来做，而且我们还需要一个double型的变量sum来记录当前所有数字之和，这样有新数字进入后，如果没有超出限制个数，则sum加上这个数字，如果超出了，那么sum先减去最早的数字，再加上这个数字，然后返回sum除以queue的个数即可：

class MovingAverage {

public:

MovingAverage(int size) {

this->size = size;

sum = 0;

}

double next(int val) {

if (q.size() >= size) {

sum -= q.front(); q.pop();

}

q.push(val);

sum += val;

return sum / q.size();

}

private:

queue<int> q;

int size;

double sum;

};

**Solution 2: Array to simulate Queue with index replacement (10min)**

import java.util.\*;

public class Solution {

int count = 0;

int[] arr;

double presum = 0.0;

public Solution(int size) {

this.arr = new int[size];

}

public double next(int val) {

int index = count % arr.length;

// No need if condition because if count < arr.length(size)

// the arr[index] just = 0

//if(count >= arr.length) {

presum -= arr[index];

//}

arr[index] = val;

presum += val;

count++;

return count < arr.length ? presum / count : presum / arr.length;

}

public static void main(String[] args) {

Solution so = new Solution(3);

double a = so.next(1);

double b = so.next(10);

double c = so.next(3);

double d = so.next(5);

System.out.println(a);

System.out.println(b);

System.out.println(c);

System.out.println(d);

}

}

**Refer to**

<https://algo.monster/liteproblems/346>

**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:

It initializes the window with a specific size, and to track the numbers within the window, it stores them in an array.

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:

Store the numbers in a fixed-size array, where the size is equal to the given window size.

Keep track of a running sum, s, of the numbers in the current window.

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:

Initialize the MovingAverage class with a window size of 3.

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.

**Java Solution**

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);

    }

}

**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.