Problem 346: Moving Average from Data Stream

Problem Information

Difficulty: Easy
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

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

Implement the

MovingAverage

class:

MovingAverage(int size)

Initializes the object with the size of the window

size

.

double next(int val)

Returns the moving average of the last

size

values of the stream.

Example 1:

Input

["MovingAverage", "next", "next", "next", "next"] [[3], [1], [10], [3], [5]]

Output

[null, 1.0, 5.5, 4.66667, 6.0]

Explanation

MovingAverage movingAverage = new MovingAverage(3); movingAverage.next(1); // return 1.0 = 1 / 1 movingAverage.next(10); // return 5.5 = (1 + 10) / 2 movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3 movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3

Constraints:

1 <= size <= 1000

-10

5

<= val <= 10

5

At most

10

4

calls will be made to

next

.

Code Snippets

C++:

```
class MovingAverage {
public:
MovingAverage(int size) {

}

double next(int val) {

}
};

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage* obj = new MovingAverage(size);
* double param_1 = obj->next(val);
*/
```

Java:

```
class MovingAverage {

public MovingAverage(int size) {

}

public double next(int val) {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage obj = new MovingAverage(size);
* double param_1 = obj.next(val);
*/
```

Python3:

```
class MovingAverage:

def __init__(self, size: int):


def next(self, val: int) -> float:




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

Python:

```
class MovingAverage(object):

def __init__(self, size):
"""
:type size: int
"""


def next(self, val):
"""
:type val: int
:rtype: float
"""




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

JavaScript:

```
/**
* @param {number} size
*/
var MovingAverage = function(size) {

};
```

```
/**
* @param {number} val
* @return {number}
*/
MovingAverage.prototype.next = function(val) {

};

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

TypeScript:

```
class MovingAverage {
constructor(size: number) {

}

next(val: number): number {

}
}

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

C#:

```
public class MovingAverage {

public MovingAverage(int size) {

}

public double Next(int val) {
```

```
}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage obj = new MovingAverage(size);
* double param_1 = obj.Next(val);
*/
```

C:

```
typedef struct {

} MovingAverage;


MovingAverage* movingAverageCreate(int size) {

}

double movingAverageNext(MovingAverage* obj, int val) {

}

void movingAverageFree(MovingAverage* obj) {

}

/**
* Your MovingAverage struct will be instantiated and called as such:
* MovingAverage* obj = movingAverageCreate(size);
* double param_1 = movingAverageNext(obj, val);

* movingAverageFree(obj);
*/
```

Go:

```
type MovingAverage struct {

}


func Constructor(size int) MovingAverage {

}


func (this *MovingAverage) Next(val int) float64 {

}


/**
* Your MovingAverage object will be instantiated and called as such:
* obj := Constructor(size);
* param_1 := obj.Next(val);
*/
```

Kotlin:

```
class MovingAverage(size: Int) {

fun next(`val`: Int): Double {

}

}

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

Swift:

```

class MovingAverage {

init(_ size: Int) {
```

```
}

func next(_ val: Int) -> Double {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* let obj = MovingAverage(size)
* let ret_1: Double = obj.next(val)
*/
```

Rust:

```
struct MovingAverage {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl MovingAverage {

fn new(size: i32) -> Self {

}

fn next(&self, val: i32) -> f64 {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* let obj = MovingAverage::new(size);
* let ret_1: f64 = obj.next(val);
*/
```

Ruby:

```
class MovingAverage

=begin
:type size: Integer
=end
def initialize(size)

end


=begin
:type val: Integer
:rtype: Float
=end
def next(val)

end



end

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

PHP:

```
class MovingAverage {
/**
* @param Integer $size
*/
function __construct($size) {

}

/**
* @param Integer $val
* @return Float
*/
function next($val) {
```

```
}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* $obj = MovingAverage($size);
* $ret_1 = $obj->next($val);
*/
```

Dart:

```
class MovingAverage {

MovingAverage(int size) {

}

double next(int val) {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage obj = MovingAverage(size);
* double param1 = obj.next(val);
*/
```

Scala:

```
class MovingAverage(_size: Int) {

def next(`val`: Int): Double = {

}

}

/**
* Your MovingAverage object will be instantiated and called as such:
* val obj = new MovingAverage(size)
```

```
* val param_1 = obj.next(`val`)
*/
```

Elixir:

```
defmodule MovingAverage do
@spec init_(size :: integer) :: any
def init_(size) do

end

@spec next(val :: integer) :: float
def next(val) do

end
end

# Your functions will be called as such:
# MovingAverage.init_(size)
# param_1 = MovingAverage.next(val)

# MovingAverage.init_ will be called before every test case, in which you can
do some necessary initializations.
```

Erlang:

```
-spec moving_average_init_(Size :: integer()) -> any().
moving_average_init_(Size) ->
.

-spec moving_average_next(Val :: integer()) -> float().
moving_average_next(Val) ->
.


%% Your functions will be called as such:
%% moving_average_init_(Size),
%% Param_1 = moving_average_next(Val),

%% moving_average_init_ will be called before every test case, in which you
can do some necessary initializations.
```

Racket:

```
(define moving-average%
(class object%
(super-new)

; size : exact-integer?
(init-field
size)

; next : exact-integer? -> flonum?
(define/public (next val)
)))

;; Your moving-average% object will be instantiated and called as such:
;; (define obj (new moving-average% [size size]))
;; (define param_1 (send obj next val))
```

Solutions

C++ Solution:

```
/*
* Problem: Moving Average from Data Stream
* Difficulty: Easy
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class MovingAverage {
public:
MovingAverage(int size) {

}

double next(int val) {

}
```

```
};

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage* obj = new MovingAverage(size);
* double param_1 = obj->next(val);
*/
```

Java Solution:

```
/**
* Problem: Moving Average from Data Stream
* Difficulty: Easy
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class MovingAverage {

public MovingAverage(int size) {

}

public double next(int val) {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage obj = new MovingAverage(size);
* double param_1 = obj.next(val);
*/
```

Python3 Solution:

```
"""
Problem: Moving Average from Data Stream
```

```
Difficulty: Easy
Tags: array, queue

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class MovingAverage:

def __init__(self, size: int):


def next(self, val: int) -> float:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class MovingAverage(object):

def __init__(self, size):
"""
:type size: int
"""


def next(self, val):
"""
:type val: int
:rtype: float
"""




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

JavaScript Solution:

```
/**
* Problem: Moving Average from Data Stream
* Difficulty: Easy
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number} size
*/
var MovingAverage = function(size) {

};

/**
* @param {number} val
* @return {number}
*/
MovingAverage.prototype.next = function(val) {

};

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

TypeScript Solution:

```
/**
* Problem: Moving Average from Data Stream
* Difficulty: Easy
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/
```

```
class MovingAverage {
constructor(size: number) {

}

next(val: number): number {

}
}

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

C# Solution:

```
/*
* Problem: Moving Average from Data Stream
* Difficulty: Easy
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class MovingAverage {

public MovingAverage(int size) {

}

public double Next(int val) {

}
}

/**
```

```
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage obj = new MovingAverage(size);
* double param_1 = obj.Next(val);
*/
```

C Solution:

```
/*
* Problem: Moving Average from Data Stream
* Difficulty: Easy
* Tags: array, queue
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/




typedef struct {

} MovingAverage;


MovingAverage* movingAverageCreate(int size) {

}

double movingAverageNext(MovingAverage* obj, int val) {

}

void movingAverageFree(MovingAverage* obj) {

}

/**
* Your MovingAverage struct will be instantiated and called as such:
* MovingAverage* obj = movingAverageCreate(size);
* double param_1 = movingAverageNext(obj, val);
```

```
* movingAverageFree(obj);
*/
```

Go Solution:

```
// Problem: Moving Average from Data Stream
// Difficulty: Easy
// Tags: array, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

type MovingAverage struct {

}


func Constructor(size int) MovingAverage {

}


func (this *MovingAverage) Next(val int) float64 {

}


/**
* Your MovingAverage object will be instantiated and called as such:
* obj := Constructor(size);
* param_1 := obj.Next(val);
*/
```

Kotlin Solution:

```
class MovingAverage(size: Int) {

fun next(`val`: Int): Double {
```

```
}

}

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

Swift Solution:

```
class MovingAverage {

init(_ size: Int) {

}

func next(_ val: Int) -> Double {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* let obj = MovingAverage(size)
* let ret_1: Double = obj.next(val)
*/
```

Rust Solution:

```
// Problem: Moving Average from Data Stream
// Difficulty: Easy
// Tags: array, queue
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

struct MovingAverage {
```

```
}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl MovingAverage {

fn new(size: i32) -> Self {

}

fn next(&self, val: i32) -> f64 {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* let obj = MovingAverage::new(size);
* let ret_1: f64 = obj.next(val);
*/
```

Ruby Solution:

```
class MovingAverage

=begin
:type size: Integer
=end
def initialize(size)

end


=begin
:type val: Integer
:rtype: Float
=end
```

```
def next(val)

end


end

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

PHP Solution:

```
class MovingAverage {
/**
* @param Integer $size
*/
function __construct($size) {

}

/**
* @param Integer $val
* @return Float
*/
function next($val) {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* $obj = MovingAverage($size);
* $ret_1 = $obj->next($val);
*/
```

Dart Solution:

```
class MovingAverage {

MovingAverage(int size) {
```

```
}

double next(int val) {

}
}

/**
* Your MovingAverage object will be instantiated and called as such:
* MovingAverage obj = MovingAverage(size);
* double param1 = obj.next(val);
*/
```

Scala Solution:

```
class MovingAverage(_size: Int) {

def next(`val`: Int): Double = {

}

}

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

Elixir Solution:

```
defmodule MovingAverage do
@spec init_(size :: integer) :: any
def init_(size) do

end

@spec next(val :: integer) :: float
def next(val) do
```

```
end
end

# Your functions will be called as such:
# MovingAverage.init_(size)
# param_1 = MovingAverage.next(val)

# MovingAverage.init_ will be called before every test case, in which you can
do some necessary initializations.
```

Erlang Solution:

```
-spec moving_average_init_(Size :: integer()) -> any().
moving_average_init_(Size) ->
.

-spec moving_average_next(Val :: integer()) -> float().
moving_average_next(Val) ->
.


%% Your functions will be called as such:
%% moving_average_init_(Size),
%% Param_1 = moving_average_next(Val),

%% moving_average_init_ will be called before every test case, in which you
can do some necessary initializations.
```

Racket Solution:

```
(define moving-average%
(class object%
(super-new)

; size : exact-integer?
(init-field
size)

; next : exact-integer? -> flonum?
(define/public (next val)
)))
```

```
;; Your moving-average% object will be instantiated and called as such:
;; (define obj (new moving-average% [size size]))
;; (define param_1 (send obj next val))
```