Problem 1825: Finding MK Average

Problem Information

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

Problem Description

You are given two integers,

m

and

k

, and a stream of integers. You are tasked to implement a data structure that calculates the

MKAverage

for the stream.

The

MKAverage

can be calculated using these steps:

If the number of the elements in the stream is less than

m

you should consider the

MKAverage

to be

-1

. Otherwise, copy the last

m

elements of the stream to a separate container.

Remove the smallest

k

elements and the largest

k

elements from the container.

Calculate the average value for the rest of the elements

rounded down to the nearest integer

.

Implement the

MKAverage

class:

MKAverage(int m, int k)

Initializes the

MKAverage

object with an empty stream and the two integers

m

and

k

.

void addElement(int num)

Inserts a new element

num

into the stream.

int calculateMKAverage()

Calculates and returns the

MKAverage

for the current stream

rounded down to the nearest integer

.

Example 1:

Input

["MKAverage", "addElement", "addElement", "calculateMKAverage", "addElement", "calculateMKAverage", "addElement", "addElement", "addElement", "calculateMKAverage"]
[[3, 1], [3], [1], [], [10], [], [5], [5], [5], []]

Output

[null, null, null, -1, null, 3, null, null, null, 5]

Explanation

MKAverage obj = new MKAverage(3, 1); obj.addElement(3); // current elements are [3] obj.addElement(1); // current elements are [3,1] obj.calculateMKAverage(); // return -1, because m = 3 and only 2 elements exist. obj.addElement(10); // current elements are [3,1,10] obj.calculateMKAverage(); // The last 3 elements are [3,1,10]. // After removing smallest and largest 1 element the container will be [3]. // The average of [3] equals 3/1 = 3, return 3 obj.addElement(5); // current elements are [3,1,10,5] obj.addElement(5); // current elements are [3,1,10,5,5] obj.addElement(5); // current elements are [3,1,10,5,5,5] obj.calculateMKAverage(); // The last 3 elements are [5,5,5]. // After removing smallest and largest 1 element the container will be [5]. // The average of [5] equals 5/1 = 5, return 5

Constraints:

3 <= m <= 10

5

1 < k*2 < m

1 <= num <= 10

5

At most

10

5

calls will be made to

addElement

and

calculateMKAverage

.

Code Snippets

C++:

```
class MKAverage {
public:
MKAverage(int m, int k) {

}

void addElement(int num) {

}

int calculateMKAverage() {

}
};

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage* obj = new MKAverage(m, k);
* obj->addElement(num);
* int param_2 = obj->calculateMKAverage();
*/
```

Java:

```
class MKAverage {

public MKAverage(int m, int k) {

}

public void addElement(int num) {

}

public int calculateMKAverage() {
```

```
}
}

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage obj = new MKAverage(m, k);
* obj.addElement(num);
* int param_2 = obj.calculateMKAverage();
*/
```

Python3:

```
class MKAverage:

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


def addElement(self, num: int) -> None:


def calculateMKAverage(self) -> int:



# Your MKAverage object will be instantiated and called as such:
# obj = MKAverage(m, k)
# obj.addElement(num)
# param_2 = obj.calculateMKAverage()
```

Python:

```
class MKAverage(object):

def __init__(self, m, k):
"""
:type m: int
:type k: int
"""


def addElement(self, num):
```

```
        """
        :type num: int
        :rtype: None
        """


    def calculateMKAverage(self):
        """
        :rtype: int
        """



# Your MKAverage object will be instantiated and called as such:
# obj = MKAverage(m, k)
# obj.addElement(num)
# param_2 = obj.calculateMKAverage()
```

JavaScript:

```
/**
* @param {number} m
* @param {number} k
*/
var MKAverage = function(m, k) {

};

/**
* @param {number} num
* @return {void}
*/
MKAverage.prototype.addElement = function(num) {

};

/**
* @return {number}
*/
MKAverage.prototype.calculateMKAverage = function() {

};
```

```
/**
* Your MKAverage object will be instantiated and called as such:
* var obj = new MKAverage(m, k)
* obj.addElement(num)
* var param_2 = obj.calculateMKAverage()
*/
```

TypeScript:

```
class MKAverage {
constructor(m: number, k: number) {

}

addElement(num: number): void {

}

calculateMKAverage(): number {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* var obj = new MKAverage(m, k)
* obj.addElement(num)
* var param_2 = obj.calculateMKAverage()
*/
```

C#:

```
public class MKAverage {

public MKAverage(int m, int k) {

}

public void AddElement(int num) {

}
```

```
public int CalculateMKAverage() {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage obj = new MKAverage(m, k);
* obj.AddElement(num);
* int param_2 = obj.CalculateMKAverage();
*/
```

C:

```
typedef struct {

} MKAverage;


MKAverage* mKAverageCreate(int m, int k) {

}

void mKAverageAddElement(MKAverage* obj, int num) {

}

int mKAverageCalculateMKAverage(MKAverage* obj) {

}

void mKAverageFree(MKAverage* obj) {

}

/**
* Your MKAverage struct will be instantiated and called as such:
* MKAverage* obj = mKAverageCreate(m, k);
```

```
* mKAverageAddElement(obj, num);

* int param_2 = mKAverageCalculateMKAverage(obj);

* mKAverageFree(obj);
*/
```

Go:

```
type MKAverage struct {

}


func Constructor(m int, k int) MKAverage {

}


func (this *MKAverage) AddElement(num int) {

}


func (this *MKAverage) CalculateMKAverage() int {

}


/**
* Your MKAverage object will be instantiated and called as such:
* obj := Constructor(m, k);
* obj.AddElement(num);
* param_2 := obj.CalculateMKAverage();
*/
```

Kotlin:

```
class MKAverage(m: Int, k: Int) {

fun addElement(num: Int) {
```

```
}

fun calculateMKAverage(): Int {

}

}

/**
* Your MKAverage object will be instantiated and called as such:
* var obj = MKAverage(m, k)
* obj.addElement(num)
* var param_2 = obj.calculateMKAverage()
*/
```

Swift:

```
class MKAverage {

init(_ m: Int, _ k: Int) {

}

func addElement(_ num: Int) {

}

func calculateMKAverage() -> Int {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* let obj = MKAverage(m, k)
* obj.addElement(num)
* let ret_2: Int = obj.calculateMKAverage()
*/
```

Rust:

```
struct MKAverage {

}


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

fn new(m: i32, k: i32) -> Self {

}

fn add_element(&self, num: i32) {

}

fn calculate_mk_average(&self) -> i32 {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* let obj = MKAverage::new(m, k);
* obj.add_element(num);
* let ret_2: i32 = obj.calculate_mk_average();
*/
```

Ruby:

```
class MKAverage

=begin
:type m: Integer
:type k: Integer
=end
def initialize(m, k)

end
```

```
=begin
:type num: Integer
:rtype: Void
=end
def add_element(num)

end


=begin
:rtype: Integer
=end
def calculate_mk_average()

end



end

# Your MKAverage object will be instantiated and called as such:
# obj = MKAverage.new(m, k)
# obj.add_element(num)
# param_2 = obj.calculate_mk_average()
```

PHP:

```
class MKAverage {
/**
* @param Integer $m
* @param Integer $k
*/
function __construct($m, $k) {

}

/**
* @param Integer $num
* @return NULL
*/
function addElement($num) {
```

```
}

/**
* @return Integer
*/
function calculateMKAverage() {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* $obj = MKAverage($m, $k);
* $obj->addElement($num);
* $ret_2 = $obj->calculateMKAverage();
*/
```

Dart:

```
class MKAverage {

MKAverage(int m, int k) {

}

void addElement(int num) {

}

int calculateMKAverage() {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage obj = MKAverage(m, k);
* obj.addElement(num);
* int param2 = obj.calculateMKAverage();
*/
```

Scala:

```
class MKAverage(_m: Int, _k: Int) {

def addElement(num: Int): Unit = {

}

def calculateMKAverage(): Int = {

}

}

/**
* Your MKAverage object will be instantiated and called as such:
* val obj = new MKAverage(m, k)
* obj.addElement(num)
* val param_2 = obj.calculateMKAverage()
*/
```

Elixir:

```
defmodule MKAverage do
@spec init_(m :: integer, k :: integer) :: any
def init_(m, k) do

end

@spec add_element(num :: integer) :: any
def add_element(num) do

end

@spec calculate_mk_average() :: integer
def calculate_mk_average() do

end
end

# Your functions will be called as such:
# MKAverage.init_(m, k)
# MKAverage.add_element(num)
```

```
# param_2 = MKAverage.calculate_mk_average()

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

Erlang:

```
-spec mk_average_init_(M :: integer(), K :: integer()) -> any().
mk_average_init_(M, K) ->
.

-spec mk_average_add_element(Num :: integer()) -> any().
mk_average_add_element(Num) ->
.

-spec mk_average_calculate_mk_average() -> integer().
mk_average_calculate_mk_average() ->
.


%% Your functions will be called as such:
%% mk_average_init_(M, K),
%% mk_average_add_element(Num),
%% Param_2 = mk_average_calculate_mk_average(),

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

Racket:

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

; m : exact-integer?
; k : exact-integer?
(init-field
m
k)

; add-element : exact-integer? -> void?
(define/public (add-element num)
```

```
)
; calculate-mk-average : -> exact-integer?
(define/public (calculate-mk-average)
)))

;; Your mk-average% object will be instantiated and called as such:
;; (define obj (new mk-average% [m m] [k k]))
;; (send obj add-element num)
;; (define param_2 (send obj calculate-mk-average))
```

Solutions

C++ Solution:

```
/*
* Problem: Finding MK Average
* Difficulty: Hard
* Tags: queue, heap
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class MKAverage {
public:
MKAverage(int m, int k) {

}

void addElement(int num) {

}

int calculateMKAverage() {

}
};

/**
```

```
* Your MKAverage object will be instantiated and called as such:
* MKAverage* obj = new MKAverage(m, k);
* obj->addElement(num);
* int param_2 = obj->calculateMKAverage();
*/
```

Java Solution:

```
/**
* Problem: Finding MK Average
* Difficulty: Hard
* Tags: queue, heap
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class MKAverage {

public MKAverage(int m, int k) {

}

public void addElement(int num) {

}

public int calculateMKAverage() {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage obj = new MKAverage(m, k);
* obj.addElement(num);
* int param_2 = obj.calculateMKAverage();
*/
```

Python3 Solution:

```
"""
Problem: Finding MK Average
Difficulty: Hard
Tags: queue, heap

Approach: Optimized algorithm based on problem constraints
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(1) to O(n) depending on approach
"""

class MKAverage:

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



def addElement(self, num: int) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class MKAverage(object):

def __init__(self, m, k):
"""
:type m: int
:type k: int
"""



def addElement(self, num):
"""
:type num: int
:rtype: None
"""



def calculateMKAverage(self):
"""
:rtype: int
"""
```

```
# Your MKAverage object will be instantiated and called as such:
# obj = MKAverage(m, k)
# obj.addElement(num)
# param_2 = obj.calculateMKAverage()
```

JavaScript Solution:

```
/**
* Problem: Finding MK Average
* Difficulty: Hard
* Tags: queue, heap
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number} m
* @param {number} k
*/
var MKAverage = function(m, k) {

};

/**
* @param {number} num
* @return {void}
*/
MKAverage.prototype.addElement = function(num) {

};

/**
* @return {number}
*/
MKAverage.prototype.calculateMKAverage = function() {

};
```

```
/**
* Your MKAverage object will be instantiated and called as such:
* var obj = new MKAverage(m, k)
* obj.addElement(num)
* var param_2 = obj.calculateMKAverage()
*/
```

TypeScript Solution:

```
/**
* Problem: Finding MK Average
* Difficulty: Hard
* Tags: queue, heap
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

class MKAverage {
constructor(m: number, k: number) {

}

addElement(num: number): void {

}

calculateMKAverage(): number {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* var obj = new MKAverage(m, k)
* obj.addElement(num)
* var param_2 = obj.calculateMKAverage()
*/
```

C# Solution:

```
/*
* Problem: Finding MK Average
* Difficulty: Hard
* Tags: queue, heap
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(1) to O(n) depending on approach
*/

public class MKAverage {

public MKAverage(int m, int k) {

}

public void AddElement(int num) {

}

public int CalculateMKAverage() {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage obj = new MKAverage(m, k);
* obj.AddElement(num);
* int param_2 = obj.CalculateMKAverage();
*/
```

C Solution:

```
/*
* Problem: Finding MK Average
* Difficulty: Hard
* Tags: queue, heap
*
* Approach: Optimized algorithm based on problem constraints
* Time Complexity: O(n) to O(n^2) depending on approach
```

```
* Space Complexity: O(1) to O(n) depending on approach
*/




typedef struct {

} MKAverage;


MKAverage* mKAverageCreate(int m, int k) {

}

void mKAverageAddElement(MKAverage* obj, int num) {

}

int mKAverageCalculateMKAverage(MKAverage* obj) {

}

void mKAverageFree(MKAverage* obj) {

}

/**
* Your MKAverage struct will be instantiated and called as such:
* MKAverage* obj = mKAverageCreate(m, k);
* mKAverageAddElement(obj, num);

* int param_2 = mKAverageCalculateMKAverage(obj);

* mKAverageFree(obj);
*/
```

Go Solution:

```
// Problem: Finding MK Average
// Difficulty: Hard
```

```
// Tags: queue, heap
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

type MKAverage struct {

}


func Constructor(m int, k int) MKAverage {

}


func (this *MKAverage) AddElement(num int) {

}


func (this *MKAverage) CalculateMKAverage() int {

}


/**
* Your MKAverage object will be instantiated and called as such:
* obj := Constructor(m, k);
* obj.AddElement(num);
* param_2 := obj.CalculateMKAverage();
*/
```

Kotlin Solution:

```
class MKAverage(m: Int, k: Int) {

fun addElement(num: Int) {

}
```

```
fun calculateMKAverage(): Int {

}

}

/**
* Your MKAverage object will be instantiated and called as such:
* var obj = MKAverage(m, k)
* obj.addElement(num)
* var param_2 = obj.calculateMKAverage()
*/
```

Swift Solution:

```
class MKAverage {

init(_ m: Int, _ k: Int) {

}

func addElement(_ num: Int) {

}

func calculateMKAverage() -> Int {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* let obj = MKAverage(m, k)
* obj.addElement(num)
* let ret_2: Int = obj.calculateMKAverage()
*/
```

Rust Solution:

```
// Problem: Finding MK Average
// Difficulty: Hard
// Tags: queue, heap
//
// Approach: Optimized algorithm based on problem constraints
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(1) to O(n) depending on approach

struct MKAverage {

}



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

fn new(m: i32, k: i32) -> Self {

}

fn add_element(&self, num: i32) {

}

fn calculate_mk_average(&self) -> i32 {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* let obj = MKAverage::new(m, k);
* obj.add_element(num);
* let ret_2: i32 = obj.calculate_mk_average();
*/
```

Ruby Solution:

```
class MKAverage

=begin
:type m: Integer
:type k: Integer
=end
def initialize(m, k)

end


=begin
:type num: Integer
:rtype: Void
=end
def add_element(num)

end


=begin
:rtype: Integer
=end
def calculate_mk_average()

end



end

# Your MKAverage object will be instantiated and called as such:
# obj = MKAverage.new(m, k)
# obj.add_element(num)
# param_2 = obj.calculate_mk_average()
```

PHP Solution:

```
class MKAverage {
/**
* @param Integer $m
* @param Integer $k
*/
```

```
function __construct($m, $k) {

}

/**
* @param Integer $num
* @return NULL
*/
function addElement($num) {

}

/**
* @return Integer
*/
function calculateMKAverage() {

}
}

/**
* Your MKAverage object will be instantiated and called as such:
* $obj = MKAverage($m, $k);
* $obj->addElement($num);
* $ret_2 = $obj->calculateMKAverage();
*/
```

Dart Solution:

```
class MKAverage {

MKAverage(int m, int k) {

}

void addElement(int num) {

}

int calculateMKAverage() {
```

```
}
}

/**
* Your MKAverage object will be instantiated and called as such:
* MKAverage obj = MKAverage(m, k);
* obj.addElement(num);
* int param2 = obj.calculateMKAverage();
*/
```

Scala Solution:

```
class MKAverage(_m: Int, _k: Int) {

def addElement(num: Int): Unit = {

}

def calculateMKAverage(): Int = {

}

}

/**
* Your MKAverage object will be instantiated and called as such:
* val obj = new MKAverage(m, k)
* obj.addElement(num)
* val param_2 = obj.calculateMKAverage()
*/
```

Elixir Solution:

```
defmodule MKAverage do
@spec init_(m :: integer, k :: integer) :: any
def init_(m, k) do

end

@spec add_element(num :: integer) :: any
def add_element(num) do
```

```
end

@spec calculate_mk_average() :: integer
def calculate_mk_average() do

end
end

# Your functions will be called as such:
# MKAverage.init_(m, k)
# MKAverage.add_element(num)
# param_2 = MKAverage.calculate_mk_average()

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

Erlang Solution:

```
-spec mk_average_init_(M :: integer(), K :: integer()) -> any().
mk_average_init_(M, K) ->
.

-spec mk_average_add_element(Num :: integer()) -> any().
mk_average_add_element(Num) ->
.

-spec mk_average_calculate_mk_average() -> integer().
mk_average_calculate_mk_average() ->
.


%% Your functions will be called as such:
%% mk_average_init_(M, K),
%% mk_average_add_element(Num),
%% Param_2 = mk_average_calculate_mk_average(),

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

Racket Solution:

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

; m : exact-integer?
; k : exact-integer?
(init-field
m
k)

; add-element : exact-integer? -> void?
(define/public (add-element num)
)
; calculate-mk-average : -> exact-integer?
(define/public (calculate-mk-average)
)))

;; Your mk-average% object will be instantiated and called as such:
;; (define obj (new mk-average% [m m] [k k]))
;; (send obj add-element num)
;; (define param_2 (send obj calculate-mk-average))
```