Problem 2526: Find Consecutive Integers from a Data Stream

Problem Information

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

Problem Description

For a stream of integers, implement a data structure that checks if the last

k

integers parsed in the stream are

equal

to

value

.

Implement the

DataStream

class:

DataStream(int value, int k)

Initializes the object with an empty integer stream and the two integers

value

and

k

.

boolean consec(int num)

Adds

num

to the stream of integers. Returns

true

if the last

k

integers are equal to

value

, and

false

otherwise. If there are less than

k

integers, the condition does not hold true, so returns

false

.

Example 1:

Input

["DataStream", "consec", "consec", "consec", "consec"] [[4, 3], [4], [4], [4], [3]]

Output

[null, false, false, true, false]

Explanation

DataStream dataStream = new DataStream(4, 3); //value = 4, k = 3 dataStream.consec(4); // Only 1 integer is parsed, so returns False. dataStream.consec(4); // Only 2 integers are parsed. // Since 2 is less than k, returns False. dataStream.consec(4); // The 3 integers parsed are all equal to value, so returns True. dataStream.consec(3); // The last k integers parsed in the stream are [4,4,3]. // Since 3 is not equal to value, it returns False.

Constraints:

1 <= value, num <= 10

9

1 <= k <= 10

5

At most

10

5

calls will be made to

consec

.

Code Snippets

C++:

```
class DataStream {
public:
DataStream(int value, int k) {

}

bool consec(int num) {

}
};

/**
* Your DataStream object will be instantiated and called as such:
* DataStream* obj = new DataStream(value, k);
* bool param_1 = obj->consec(num);
*/
```

Java:

```
class DataStream {

public DataStream(int value, int k) {

}

public boolean consec(int num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* DataStream obj = new DataStream(value, k);
* boolean param_1 = obj.consec(num);
*/
```

Python3:

```
class DataStream:

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


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




# Your DataStream object will be instantiated and called as such:
# obj = DataStream(value, k)
# param_1 = obj.consec(num)
```

Python:

```
class DataStream(object):

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


def consec(self, num):
"""
:type num: int
:rtype: bool
"""




# Your DataStream object will be instantiated and called as such:
# obj = DataStream(value, k)
# param_1 = obj.consec(num)
```

JavaScript:

```
/**
* @param {number} value
* @param {number} k
*/
var DataStream = function(value, k) {
```

```
};

/**
* @param {number} num
* @return {boolean}
*/
DataStream.prototype.consec = function(num) {

};

/**
* Your DataStream object will be instantiated and called as such:
* var obj = new DataStream(value, k)
* var param_1 = obj.consec(num)
*/
```

TypeScript:

```
class DataStream {
constructor(value: number, k: number) {

}

consec(num: number): boolean {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* var obj = new DataStream(value, k)
* var param_1 = obj.consec(num)
*/
```

C#:

```
public class DataStream {

public DataStream(int value, int k) {

}
```

```
public bool Consec(int num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* DataStream obj = new DataStream(value, k);
* bool param_1 = obj.Consec(num);
*/
```

C:

```
typedef struct {

} DataStream;


DataStream* dataStreamCreate(int value, int k) {

}

bool dataStreamConsec(DataStream* obj, int num) {

}

void dataStreamFree(DataStream* obj) {

}

/**
* Your DataStream struct will be instantiated and called as such:
* DataStream* obj = dataStreamCreate(value, k);
* bool param_1 = dataStreamConsec(obj, num);

* dataStreamFree(obj);
*/
```

Go:

```
type DataStream struct {

}


func Constructor(value int, k int) DataStream {

}


func (this *DataStream) Consec(num int) bool {

}


/**
* Your DataStream object will be instantiated and called as such:
* obj := Constructor(value, k);
* param_1 := obj.Consec(num);
*/
```

Kotlin:

```
class DataStream(value: Int, k: Int) {

fun consec(num: Int): Boolean {

}

}

/**
* Your DataStream object will be instantiated and called as such:
* var obj = DataStream(value, k)
* var param_1 = obj.consec(num)
*/
```

Swift:

```

class DataStream {
```

```
init(_ value: Int, _ k: Int) {

}

func consec(_ num: Int) -> Bool {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* let obj = DataStream(value, k)
* let ret_1: Bool = obj.consec(num)
*/
```

Rust:

```
struct DataStream {

}


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

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

}

fn consec(&self, num: i32) -> bool {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* let obj = DataStream::new(value, k);
* let ret_1: bool = obj.consec(num);
```

```
*/
```

Ruby:

```
class DataStream

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

end


=begin
:type num: Integer
:rtype: Boolean
=end
def consec(num)

end


end

# Your DataStream object will be instantiated and called as such:
# obj = DataStream.new(value, k)
# param_1 = obj.consec(num)
```

PHP:

```
class DataStream {
/**
* @param Integer $value
* @param Integer $k
*/
function __construct($value, $k) {

}

/**
```

```
* @param Integer $num
* @return Boolean
*/
function consec($num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* $obj = DataStream($value, $k);
* $ret_1 = $obj->consec($num);
*/
```

Dart:

```
class DataStream {

DataStream(int value, int k) {

}

bool consec(int num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* DataStream obj = DataStream(value, k);
* bool param1 = obj.consec(num);
*/
```

Scala:

```
class DataStream(_value: Int, _k: Int) {

def consec(num: Int): Boolean = {

}

}
```

```
/**
* Your DataStream object will be instantiated and called as such:
* val obj = new DataStream(value, k)
* val param_1 = obj.consec(num)
*/
```

Elixir:

```
defmodule DataStream do
@spec init_(value :: integer, k :: integer) :: any
def init_(value, k) do

end

@spec consec(num :: integer) :: boolean
def consec(num) do

end
end

# Your functions will be called as such:
# DataStream.init_(value, k)
# param_1 = DataStream.consec(num)

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

Erlang:

```
-spec data_stream_init_(Value :: integer(), K :: integer()) -> any().
data_stream_init_(Value, K) ->
.

-spec data_stream_consec(Num :: integer()) -> boolean().
data_stream_consec(Num) ->
.


%% Your functions will be called as such:
%% data_stream_init_(Value, K),
%% Param_1 = data_stream_consec(Num),
```

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

Racket:

```
(define data-stream%
(class object%
(super-new)

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

; consec : exact-integer? -> boolean?
(define/public (consec num)
)))

;; Your data-stream% object will be instantiated and called as such:
;; (define obj (new data-stream% [value value] [k k]))
;; (define param_1 (send obj consec num))
```

Solutions

C++ Solution:

```
/*
* Problem: Find Consecutive Integers from a Data Stream
* Difficulty: Medium
* Tags: hash, queue
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class DataStream {
public:
```

```
DataStream(int value, int k) {

}

bool consec(int num) {

}
};

/**
* Your DataStream object will be instantiated and called as such:
* DataStream* obj = new DataStream(value, k);
* bool param_1 = obj->consec(num);
*/
```

Java Solution:

```
/**
* Problem: Find Consecutive Integers from a Data Stream
* Difficulty: Medium
* Tags: hash, queue
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class DataStream {

public DataStream(int value, int k) {

}

public boolean consec(int num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* DataStream obj = new DataStream(value, k);
```

```
* boolean param_1 = obj.consec(num);
*/
```

Python3 Solution:

```
"""
Problem: Find Consecutive Integers from a Data Stream
Difficulty: Medium
Tags: hash, queue

Approach: Use hash map for O(1) lookups
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(n) for hash map
"""

class DataStream:

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


def consec(self, num: int) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class DataStream(object):

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


def consec(self, num):
"""
:type num: int
:rtype: bool
"""
```

```
# Your DataStream object will be instantiated and called as such:
# obj = DataStream(value, k)
# param_1 = obj.consec(num)
```

JavaScript Solution:

```
/**
* Problem: Find Consecutive Integers from a Data Stream
* Difficulty: Medium
* Tags: hash, queue
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

/**
* @param {number} value
* @param {number} k
*/
var DataStream = function(value, k) {

};

/**
* @param {number} num
* @return {boolean}
*/
DataStream.prototype.consec = function(num) {

};

/**
* Your DataStream object will be instantiated and called as such:
* var obj = new DataStream(value, k)
* var param_1 = obj.consec(num)
*/
```

TypeScript Solution:

```
/**
* Problem: Find Consecutive Integers from a Data Stream
* Difficulty: Medium
* Tags: hash, queue
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class DataStream {
constructor(value: number, k: number) {

}

consec(num: number): boolean {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* var obj = new DataStream(value, k)
* var param_1 = obj.consec(num)
*/
```

C# Solution:

```
/*
* Problem: Find Consecutive Integers from a Data Stream
* Difficulty: Medium
* Tags: hash, queue
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

public class DataStream {

public DataStream(int value, int k) {
```

```
}

public bool Consec(int num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* DataStream obj = new DataStream(value, k);
* bool param_1 = obj.Consec(num);
*/
```

C Solution:

```
/*
* Problem: Find Consecutive Integers from a Data Stream
* Difficulty: Medium
* Tags: hash, queue
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/




typedef struct {

} DataStream;


DataStream* dataStreamCreate(int value, int k) {

}

bool dataStreamConsec(DataStream* obj, int num) {

}
```

```
void dataStreamFree(DataStream* obj) {

}

/**
* Your DataStream struct will be instantiated and called as such:
* DataStream* obj = dataStreamCreate(value, k);
* bool param_1 = dataStreamConsec(obj, num);

* dataStreamFree(obj);
*/
```

Go Solution:

```
// Problem: Find Consecutive Integers from a Data Stream
// Difficulty: Medium
// Tags: hash, queue
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

type DataStream struct {

}


func Constructor(value int, k int) DataStream {

}


func (this *DataStream) Consec(num int) bool {

}


/**
* Your DataStream object will be instantiated and called as such:
* obj := Constructor(value, k);
* param_1 := obj.Consec(num);
```

```
*/
```

Kotlin Solution:

```
class DataStream(value: Int, k: Int) {

fun consec(num: Int): Boolean {

}

}

/**
* Your DataStream object will be instantiated and called as such:
* var obj = DataStream(value, k)
* var param_1 = obj.consec(num)
*/
```

Swift Solution:

```
class DataStream {

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

}

func consec(_ num: Int) -> Bool {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* let obj = DataStream(value, k)
* let ret_1: Bool = obj.consec(num)
*/
```

Rust Solution:

```
// Problem: Find Consecutive Integers from a Data Stream
// Difficulty: Medium
// Tags: hash, queue
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

struct DataStream {

}



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

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

}

fn consec(&self, num: i32) -> bool {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* let obj = DataStream::new(value, k);
* let ret_1: bool = obj.consec(num);
*/
```

Ruby Solution:

```
class DataStream

=begin
:type value: Integer
:type k: Integer
=end
```

```
def initialize(value, k)

end


=begin
:type num: Integer
:rtype: Boolean
=end
def consec(num)

end



end

# Your DataStream object will be instantiated and called as such:
# obj = DataStream.new(value, k)
# param_1 = obj.consec(num)
```

PHP Solution:

```
class DataStream {
/**
* @param Integer $value
* @param Integer $k
*/
function __construct($value, $k) {

}

/**
* @param Integer $num
* @return Boolean
*/
function consec($num) {

}
}

/**
```

```
* Your DataStream object will be instantiated and called as such:
* $obj = DataStream($value, $k);
* $ret_1 = $obj->consec($num);
*/
```

Dart Solution:

```
class DataStream {

DataStream(int value, int k) {

}

bool consec(int num) {

}
}

/**
* Your DataStream object will be instantiated and called as such:
* DataStream obj = DataStream(value, k);
* bool param1 = obj.consec(num);
*/
```

Scala Solution:

```
class DataStream(_value: Int, _k: Int) {

def consec(num: Int): Boolean = {

}

}

/**
* Your DataStream object will be instantiated and called as such:
* val obj = new DataStream(value, k)
* val param_1 = obj.consec(num)
*/
```

Elixir Solution:

```
defmodule DataStream do
@spec init_(value :: integer, k :: integer) :: any
def init_(value, k) do

end

@spec consec(num :: integer) :: boolean
def consec(num) do

end
end

# Your functions will be called as such:
# DataStream.init_(value, k)
# param_1 = DataStream.consec(num)

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

Erlang Solution:

```
-spec data_stream_init_(Value :: integer(), K :: integer()) -> any().
data_stream_init_(Value, K) ->
.

-spec data_stream_consec(Num :: integer()) -> boolean().
data_stream_consec(Num) ->
.


%% Your functions will be called as such:
%% data_stream_init_(Value, K),
%% Param_1 = data_stream_consec(Num),

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

Racket Solution:

```
(define data-stream%
(class object%
(super-new)
```

```
; value : exact-integer?
; k : exact-integer?
(init-field
value
k)

; consec : exact-integer? -> boolean?
(define/public (consec num)
)))

;; Your data-stream% object will be instantiated and called as such:
;; (define obj (new data-stream% [value value] [k k]))
;; (define param_1 (send obj consec num))
```