Problem 2276: Count Integers in Intervals

Problem Information

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

Problem Description

Given an

empty

set of intervals, implement a data structure that can:

Add

an interval to the set of intervals.

Count

the number of integers that are present in

at least one

interval.

Implement the

CountIntervals

class:

CountIntervals()

Initializes the object with an empty set of intervals.

void add(int left, int right)

Adds the interval

[left, right]

to the set of intervals.

int count()

Returns the number of integers that are present in

at least one

interval.

Note

that an interval

[left, right]

denotes all the integers

x

where

left <= x <= right

.

Example 1:

Input

["CountIntervals", "add", "add", "count", "add", "count"] [[], [2, 3], [7, 10], [], [5, 8], []]

Output

[null, null, null, 6, null, 8]

Explanation

CountIntervals countIntervals = new CountIntervals(); // initialize the object with an empty set of intervals. countIntervals.add(2, 3); // add [2, 3] to the set of intervals. countIntervals.add(7, 10); // add [7, 10] to the set of intervals. countIntervals.count(); // return 6 // the integers 2 and 3 are present in the interval [2, 3]. // the integers 7, 8, 9, and 10 are present in the interval [7, 10]. countIntervals.add(5, 8); // add [5, 8] to the set of intervals. countIntervals.count(); // return 8 // the integers 2 and 3 are present in the interval [2, 3]. // the integers 5 and 6 are present in the interval [5, 8]. // the integers 7 and 8 are present in the intervals [5, 8] and [7, 10]. // the integers 9 and 10 are present in the interval [7, 10].

Constraints:

1 <= left <= right <= 10

9

At most

10

5

calls

in total

will be made to

add

and

count

.

At least

one

call will be made to

count

.

Code Snippets

C++:

```
class CountIntervals {
public:
CountIntervals() {

}

void add(int left, int right) {

}

int count() {

}
};

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals* obj = new CountIntervals();
* obj->add(left,right);
* int param_2 = obj->count();
*/
```

Java:

```
class CountIntervals {

public CountIntervals() {

}

public void add(int left, int right) {

}

public int count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals obj = new CountIntervals();
* obj.add(left,right);
* int param_2 = obj.count();
*/
```

Python3:

```
class CountIntervals:

def __init__(self):


def add(self, left: int, right: int) -> None:


def count(self) -> int:



# Your CountIntervals object will be instantiated and called as such:
# obj = CountIntervals()
# obj.add(left,right)
# param_2 = obj.count()
```

Python:

```
class CountIntervals(object):

def __init__(self):


def add(self, left, right):
"""
:type left: int
:type right: int
:rtype: None
"""


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



# Your CountIntervals object will be instantiated and called as such:
# obj = CountIntervals()
# obj.add(left,right)
# param_2 = obj.count()
```

JavaScript:

```

var CountIntervals = function() {

};

/**
* @param {number} left
* @param {number} right
* @return {void}
*/
CountIntervals.prototype.add = function(left, right) {

};

/**
* @return {number}
```

```
*/
CountIntervals.prototype.count = function() {

};

/**
* Your CountIntervals object will be instantiated and called as such:
* var obj = new CountIntervals()
* obj.add(left,right)
* var param_2 = obj.count()
*/
```

TypeScript:

```
class CountIntervals {
constructor() {

}

add(left: number, right: number): void {

}

count(): number {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* var obj = new CountIntervals()
* obj.add(left,right)
* var param_2 = obj.count()
*/
```

C#:

```
public class CountIntervals {

public CountIntervals() {

}
public void Add(int left, int right) {

}

public int Count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals obj = new CountIntervals();
* obj.Add(left,right);
* int param_2 = obj.Count();
*/
```

C:

```
typedef struct {

} CountIntervals;


CountIntervals* countIntervalsCreate() {

}

void countIntervalsAdd(CountIntervals* obj, int left, int right) {

}

int countIntervalsCount(CountIntervals* obj) {

}

void countIntervalsFree(CountIntervals* obj) {

}
```

```
/**
* Your CountIntervals struct will be instantiated and called as such:
* CountIntervals* obj = countIntervalsCreate();
* countIntervalsAdd(obj, left, right);

* int param_2 = countIntervalsCount(obj);

* countIntervalsFree(obj);
*/
```

Go:

```
type CountIntervals struct {

}


func Constructor() CountIntervals {

}


func (this *CountIntervals) Add(left int, right int) {

}


func (this *CountIntervals) Count() int {

}


/**
* Your CountIntervals object will be instantiated and called as such:
* obj := Constructor();
* obj.Add(left,right);
* param_2 := obj.Count();
*/
```

Kotlin:

```
class CountIntervals() {

fun add(left: Int, right: Int) {

}

fun count(): Int {

}

}

/**
* Your CountIntervals object will be instantiated and called as such:
* var obj = CountIntervals()
* obj.add(left,right)
* var param_2 = obj.count()
*/
```

Swift:

```

class CountIntervals {

init() {

}

func add(_ left: Int, _ right: Int) {

}

func count() -> Int {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* let obj = CountIntervals()
* obj.add(left, right)
* let ret_2: Int = obj.count()
*/
```

Rust:

```
struct CountIntervals {

}



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

fn new() -> Self {

}

fn add(&self, left: i32, right: i32) {

}

fn count(&self) -> i32 {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* let obj = CountIntervals::new();
* obj.add(left, right);
* let ret_2: i32 = obj.count();
*/
```

Ruby:

```
class CountIntervals
def initialize()

end



=begin
```

```
:type left: Integer
:type right: Integer
:rtype: Void
=end
def add(left, right)

end



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

end



end

# Your CountIntervals object will be instantiated and called as such:
# obj = CountIntervals.new()
# obj.add(left, right)
# param_2 = obj.count()
```

PHP:

```
class CountIntervals {
/**
*/
function __construct() {

}

/**
* @param Integer $left
* @param Integer $right
* @return NULL
*/
function add($left, $right) {

}
/**
* @return Integer
*/
function count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* $obj = CountIntervals();
* $obj->add($left, $right);
* $ret_2 = $obj->count();
*/
```

Dart:

```
class CountIntervals {

CountIntervals() {

}

void add(int left, int right) {

}

int count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals obj = CountIntervals();
* obj.add(left,right);
* int param2 = obj.count();
*/
```

Scala:

```
class CountIntervals() {

def add(left: Int, right: Int): Unit = {

}

def count(): Int = {

}

}

/**
* Your CountIntervals object will be instantiated and called as such:
* val obj = new CountIntervals()
* obj.add(left,right)
* val param_2 = obj.count()
*/
```

Elixir:

```
defmodule CountIntervals do
@spec init_() :: any
def init_() do

end

@spec add(left :: integer, right :: integer) :: any
def add(left, right) do

end

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

end
end

# Your functions will be called as such:
# CountIntervals.init_()
# CountIntervals.add(left, right)
# param_2 = CountIntervals.count()
# CountIntervals.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang:

```
-spec count_intervals_init_() -> any().
count_intervals_init_() ->
.

-spec count_intervals_add(Left :: integer(), Right :: integer()) -> any().
count_intervals_add(Left, Right) ->
.

-spec count_intervals_count() -> integer().
count_intervals_count() ->
.


%% Your functions will be called as such:
%% count_intervals_init_(),
%% count_intervals_add(Left, Right),
%% Param_2 = count_intervals_count(),

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

Racket:

```
(define count-intervals%
(class object%
(super-new)

(init-field)

; add : exact-integer? exact-integer? -> void?
(define/public (add left right)
)
; count : -> exact-integer?
(define/public (count)
)))

;; Your count-intervals% object will be instantiated and called as such:
```

```
;; (define obj (new count-intervals%))
;; (send obj add left right)
;; (define param_2 (send obj count))
```

Solutions

C++ Solution:

```
/*
* Problem: Count Integers in Intervals
* Difficulty: Hard
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

class CountIntervals {
public:
CountIntervals() {

}

void add(int left, int right) {

}

int count() {

}
};

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals* obj = new CountIntervals();
* obj->add(left,right);
* int param_2 = obj->count();
*/
```

Java Solution:

```
/**
* Problem: Count Integers in Intervals
* Difficulty: Hard
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

class CountIntervals {

public CountIntervals() {

}

public void add(int left, int right) {

}

public int count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals obj = new CountIntervals();
* obj.add(left,right);
* int param_2 = obj.count();
*/
```

Python3 Solution:

```
"""
Problem: Count Integers in Intervals
Difficulty: Hard
Tags: tree

Approach: DFS or BFS traversal
Time Complexity: O(n) where n is number of nodes
```

```
Space Complexity: O(h) for recursion stack where h is height
"""

class CountIntervals:

def __init__(self):


def add(self, left: int, right: int) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class CountIntervals(object):

def __init__(self):


def add(self, left, right):
"""
:type left: int
:type right: int
:rtype: None
"""


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



# Your CountIntervals object will be instantiated and called as such:
# obj = CountIntervals()
# obj.add(left,right)
# param_2 = obj.count()
```

JavaScript Solution:

```
/**
* Problem: Count Integers in Intervals
* Difficulty: Hard
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/


var CountIntervals = function() {

};

/**
* @param {number} left
* @param {number} right
* @return {void}
*/
CountIntervals.prototype.add = function(left, right) {

};

/**
* @return {number}
*/
CountIntervals.prototype.count = function() {

};

/**
* Your CountIntervals object will be instantiated and called as such:
* var obj = new CountIntervals()
* obj.add(left,right)
* var param_2 = obj.count()
*/
```

TypeScript Solution:

```
/**
* Problem: Count Integers in Intervals
* Difficulty: Hard
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

class CountIntervals {
constructor() {

}

add(left: number, right: number): void {

}

count(): number {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* var obj = new CountIntervals()
* obj.add(left,right)
* var param_2 = obj.count()
*/
```

C# Solution:

```
/*
* Problem: Count Integers in Intervals
* Difficulty: Hard
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/
```

```
public class CountIntervals {

public CountIntervals() {

}

public void Add(int left, int right) {

}

public int Count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals obj = new CountIntervals();
* obj.Add(left,right);
* int param_2 = obj.Count();
*/
```

C Solution:

```
/*
* Problem: Count Integers in Intervals
* Difficulty: Hard
* Tags: tree
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/




typedef struct {

} CountIntervals;
```

```
CountIntervals* countIntervalsCreate() {

}

void countIntervalsAdd(CountIntervals* obj, int left, int right) {

}

int countIntervalsCount(CountIntervals* obj) {

}

void countIntervalsFree(CountIntervals* obj) {

}

/**
* Your CountIntervals struct will be instantiated and called as such:
* CountIntervals* obj = countIntervalsCreate();
* countIntervalsAdd(obj, left, right);

* int param_2 = countIntervalsCount(obj);

* countIntervalsFree(obj);
*/
```

Go Solution:

```
// Problem: Count Integers in Intervals
// Difficulty: Hard
// Tags: tree
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

type CountIntervals struct {


}
```

```
func Constructor() CountIntervals {

}


func (this *CountIntervals) Add(left int, right int) {

}


func (this *CountIntervals) Count() int {

}


/**
* Your CountIntervals object will be instantiated and called as such:
* obj := Constructor();
* obj.Add(left,right);
* param_2 := obj.Count();
*/
```

Kotlin Solution:

```
class CountIntervals() {

fun add(left: Int, right: Int) {

}

fun count(): Int {

}

}

/**
* Your CountIntervals object will be instantiated and called as such:
* var obj = CountIntervals()
* obj.add(left,right)
* var param_2 = obj.count()
*/
```

Swift Solution:

```
class CountIntervals {

init() {

}

func add(_ left: Int, _ right: Int) {

}

func count() -> Int {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* let obj = CountIntervals()
* obj.add(left, right)
* let ret_2: Int = obj.count()
*/
```

Rust Solution:

```
// Problem: Count Integers in Intervals
// Difficulty: Hard
// Tags: tree
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

struct CountIntervals {


}
```

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

fn new() -> Self {

}

fn add(&self, left: i32, right: i32) {

}

fn count(&self) -> i32 {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* let obj = CountIntervals::new();
* obj.add(left, right);
* let ret_2: i32 = obj.count();
*/
```

Ruby Solution:

```
class CountIntervals
def initialize()

end



=begin
:type left: Integer
:type right: Integer
:rtype: Void
=end
```

```
def add(left, right)

end



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

end



end

# Your CountIntervals object will be instantiated and called as such:
# obj = CountIntervals.new()
# obj.add(left, right)
# param_2 = obj.count()
```

PHP Solution:

```
class CountIntervals {
/**
*/
function __construct() {

}

/**
* @param Integer $left
* @param Integer $right
* @return NULL
*/
function add($left, $right) {

}

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

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* $obj = CountIntervals();
* $obj->add($left, $right);
* $ret_2 = $obj->count();
*/
```

Dart Solution:

```
class CountIntervals {

CountIntervals() {

}

void add(int left, int right) {

}

int count() {

}
}

/**
* Your CountIntervals object will be instantiated and called as such:
* CountIntervals obj = CountIntervals();
* obj.add(left,right);
* int param2 = obj.count();
*/
```

Scala Solution:

```
class CountIntervals() {

def add(left: Int, right: Int): Unit = {
}

def count(): Int = {

}

}

/**
* Your CountIntervals object will be instantiated and called as such:
* val obj = new CountIntervals()
* obj.add(left,right)
* val param_2 = obj.count()
*/
```

Elixir Solution:

```
defmodule CountIntervals do
@spec init_() :: any
def init_() do

end

@spec add(left :: integer, right :: integer) :: any
def add(left, right) do

end

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

end
end

# Your functions will be called as such:
# CountIntervals.init_()
# CountIntervals.add(left, right)
# param_2 = CountIntervals.count()

# CountIntervals.init_ will be called before every test case, in which you
```

```
can do some necessary initializations.
```

Erlang Solution:

```
-spec count_intervals_init_() -> any().
count_intervals_init_() ->
.

-spec count_intervals_add(Left :: integer(), Right :: integer()) -> any().
count_intervals_add(Left, Right) ->
.

-spec count_intervals_count() -> integer().
count_intervals_count() ->
.



%% Your functions will be called as such:
%% count_intervals_init_(),
%% count_intervals_add(Left, Right),
%% Param_2 = count_intervals_count(),

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

Racket Solution:

```
(define count-intervals%
(class object%
(super-new)

(init-field)

; add : exact-integer? exact-integer? -> void?
(define/public (add left right)
)
; count : -> exact-integer?
(define/public (count)
)))

;; Your count-intervals% object will be instantiated and called as such:
;; (define obj (new count-intervals%))
;; (send obj add left right)
;; (define param_2 (send obj count))
```