Problem 3360: Stone Removal Game

Problem Information

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

Problem Description

Alice and Bob are playing a game where they take turns removing stones from a pile, with

Alice going first

.

Alice starts by removing

exactly

10 stones on her first turn.

For each subsequent turn, each player removes

exactly

1 fewer

stone

than the previous opponent.

The player who cannot make a move loses the game.

Given a positive integer

n

, return

true

if Alice wins the game and

false

otherwise.

Example 1:

Input:

n = 12

Output:

true

Explanation:

Alice removes 10 stones on her first turn, leaving 2 stones for Bob.

Bob cannot remove 9 stones, so Alice wins.

Example 2:

Input:

n = 1

Output:

false

Explanation:

Alice cannot remove 10 stones, so Alice loses.

Constraints:

1 <= n <= 50

Code Snippets

C++:

```
class Solution {
public:
bool canAliceWin(int n) {

}
};
```

Java:

```
class Solution {
public boolean canAliceWin(int n) {

}
}
```

Python3:

```
class Solution:
def canAliceWin(self, n: int) -> bool:
```

Python:

```
class Solution(object):
def canAliceWin(self, n):
"""
:type n: int
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number} n
* @return {boolean}
*/
var canAliceWin = function(n) {

};
```

TypeScript:

```
function canAliceWin(n: number): boolean {

};
```

C#:

```
public class Solution {
public bool CanAliceWin(int n) {

}
}
```

C:

```
bool canAliceWin(int n) {

}
```

Go:

```
func canAliceWin(n int) bool {

}
```

Kotlin:

```
class Solution {
fun canAliceWin(n: Int): Boolean {

}
}
```

Swift:

```
class Solution {
func canAliceWin(_ n: Int) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_alice_win(n: i32) -> bool {

}
}
```

Ruby:

```
# @param {Integer} n
# @return {Boolean}
def can_alice_win(n)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @return Boolean
*/
function canAliceWin($n) {

}
}
```

Dart:

```
class Solution {
bool canAliceWin(int n) {

}
}
```

Scala:

```
object Solution {
def canAliceWin(n: Int): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec can_alice_win(n :: integer) :: boolean
def can_alice_win(n) do

end
end
```

Erlang:

```
-spec can_alice_win(N :: integer()) -> boolean().
can_alice_win(N) ->
.
```

Racket:

```
(define/contract (can-alice-win n)
(-> exact-integer? boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Stone Removal Game
* Difficulty: Easy
* Tags: math
*
* 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 Solution {
public:
bool canAliceWin(int n) {

}
};
```

Java Solution:

```
/**
* Problem: Stone Removal Game
* Difficulty: Easy
* Tags: math
*
* 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 Solution {
public boolean canAliceWin(int n) {

}
}
```

Python3 Solution:

```
"""
Problem: Stone Removal Game
Difficulty: Easy
Tags: math

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 Solution:
def canAliceWin(self, n: int) -> bool:
# TODO: Implement optimized solution
```

```
pass
```

Python Solution:

```
class Solution(object):
def canAliceWin(self, n):
"""
:type n: int
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Stone Removal Game
* Difficulty: Easy
* Tags: math
*
* 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} n
* @return {boolean}
*/
var canAliceWin = function(n) {

};
```

TypeScript Solution:

```
/**
* Problem: Stone Removal Game
* Difficulty: Easy
* Tags: math
*
* 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
```

```
*/

function canAliceWin(n: number): boolean {

};
```

C# Solution:

```
/*
* Problem: Stone Removal Game
* Difficulty: Easy
* Tags: math
*
* 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 Solution {
public bool CanAliceWin(int n) {

}
}
```

C Solution:

```
/*
* Problem: Stone Removal Game
* Difficulty: Easy
* Tags: math
*
* 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
*/

bool canAliceWin(int n) {

}
```

Go Solution:

```
// Problem: Stone Removal Game
// Difficulty: Easy
// Tags: math
//
// 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

func canAliceWin(n int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun canAliceWin(n: Int): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func canAliceWin(_ n: Int) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Stone Removal Game
// Difficulty: Easy
// Tags: math
//
// 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

impl Solution {
pub fn can_alice_win(n: i32) -> bool {

}
```

```
}
```

Ruby Solution:

```
# @param {Integer} n
# @return {Boolean}
def can_alice_win(n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @return Boolean
*/
function canAliceWin($n) {

}
}
```

Dart Solution:

```
class Solution {
bool canAliceWin(int n) {

}
}
```

Scala Solution:

```
object Solution {
def canAliceWin(n: Int): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_alice_win(n :: integer) :: boolean
def can_alice_win(n) do

end
end
```

Erlang Solution:

```
-spec can_alice_win(N :: integer()) -> boolean().
can_alice_win(N) ->
.
```

Racket Solution:

```
(define/contract (can-alice-win n)
(-> exact-integer? boolean?)
)
```