Problem 403: Frog Jump

Problem Information

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

Problem Description

A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of

stones

positions (in units) in sorted

ascending order

, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be

1

unit.

If the frog's last jump was

k

units, its next jump must be either

k - 1

,

k

, or

k + 1

units. The frog can only jump in the forward direction.

Example 1:

Input:

stones = [0,1,3,5,6,8,12,17]

Output:

true

Explanation:

The frog can jump to the last stone by jumping 1 unit to the 2nd stone, then 2 units to the 3rd stone, then 2 units to the 4th stone, then 3 units to the 6th stone, 4 units to the 7th stone, and 5 units to the 8th stone.

Example 2:

Input:

stones = [0,1,2,3,4,8,9,11]

Output:

false

Explanation:

There is no way to jump to the last stone as the gap between the 5th and 6th stone is too large.

Constraints:

2 <= stones.length <= 2000

0 <= stones[i] <= 2

31

- 1

stones[0] == 0

stones

is sorted in a strictly increasing order.

Code Snippets

C++:

```
class Solution {
public:
bool canCross(vector<int>& stones) {

}
};
```

Java:

```
class Solution {
public boolean canCross(int[] stones) {

}
}
```

Python3:

```
class Solution:
def canCross(self, stones: List[int]) -> bool:
```

Python:

```
class Solution(object):
def canCross(self, stones):
"""
:type stones: List[int]
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[]} stones
* @return {boolean}
*/
var canCross = function(stones) {

};
```

TypeScript:

```
function canCross(stones: number[]): boolean {

};
```

C#:

```
public class Solution {
public bool CanCross(int[] stones) {

}
}
```

C:

```
bool canCross(int* stones, int stonesSize) {

}
```

Go:

```
func canCross(stones []int) bool {

}
```

Kotlin:

```
class Solution {
fun canCross(stones: IntArray): Boolean {

}
}
```

Swift:

```
class Solution {
func canCross(_ stones: [Int]) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_cross(stones: Vec<i32>) -> bool {

}
}
```

Ruby:

```
# @param {Integer[]} stones
# @return {Boolean}
def can_cross(stones)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $stones
* @return Boolean

*/
function canCross($stones) {

}
}
```

Dart:

```
class Solution {
bool canCross(List<int> stones) {

}
}
```

Scala:

```
object Solution {
def canCross(stones: Array[Int]): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec can_cross(stones :: [integer]) :: boolean
def can_cross(stones) do

end
end
```

Erlang:

```
-spec can_cross(Stones :: [integer()]) -> boolean().
can_cross(Stones) ->
.
```

Racket:

```
(define/contract (can-cross stones)
(-> (listof exact-integer?) boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Frog Jump
* Difficulty: Hard
* Tags: array, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
bool canCross(vector<int>& stones) {

}
};
```

Java Solution:

```
/**
* Problem: Frog Jump
* Difficulty: Hard
* Tags: array, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public boolean canCross(int[] stones) {

}
}
```

Python3 Solution:

```
"""
Problem: Frog Jump
Difficulty: Hard
Tags: array, dp, sort

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def canCross(self, stones: List[int]) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canCross(self, stones):
"""
:type stones: List[int]
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Frog Jump
* Difficulty: Hard
* Tags: array, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} stones
* @return {boolean}
*/
var canCross = function(stones) {
};
```

TypeScript Solution:

```
/**
* Problem: Frog Jump
* Difficulty: Hard
* Tags: array, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function canCross(stones: number[]): boolean {

};
```

C# Solution:

```
/*
* Problem: Frog Jump
* Difficulty: Hard
* Tags: array, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public bool CanCross(int[] stones) {

}
}
```

C Solution:

```
/*
* Problem: Frog Jump
* Difficulty: Hard
* Tags: array, dp, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

bool canCross(int* stones, int stonesSize) {

}
```

Go Solution:

```go
// Problem: Frog Jump
// Difficulty: Hard
// Tags: array, dp, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func canCross(stones []int) bool {

}
```

Kotlin Solution:

```kotlin
class Solution {
fun canCross(stones: IntArray): Boolean {

}
}
```

Swift Solution:

```swift
class Solution {
func canCross(_ stones: [Int]) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Frog Jump
// Difficulty: Hard
// Tags: array, dp, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn can_cross(stones: Vec<i32>) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[]} stones
# @return {Boolean}
def can_cross(stones)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $stones
* @return Boolean
*/
function canCross($stones) {

}
}
```

Dart Solution:

```
class Solution {
bool canCross(List<int> stones) {

}
}
```

Scala Solution:

```
object Solution {
def canCross(stones: Array[Int]): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_cross(stones :: [integer]) :: boolean
def can_cross(stones) do

end
end
```

Erlang Solution:

```
-spec can_cross(Stones :: [integer()]) -> boolean().
can_cross(Stones) ->
.
```

Racket Solution:

```
(define/contract (can-cross stones)
(-> (listof exact-integer?) boolean?)
)
```