Problem 2498: Frog Jump II

Problem Information

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

Problem Description

You are given a

0-indexed

integer array

stones

sorted in

strictly increasing order

representing the positions of stones in a river.

A frog, initially on the first stone, wants to travel to the last stone and then return to the first stone. However, it can jump to any stone

at most once

.

The

length

of a jump is the absolute difference between the position of the stone the frog is currently on and the position of the stone to which the frog jumps.

More formally, if the frog is at

stones[i]

and is jumping to

stones[j]

, the length of the jump is

|stones[i] - stones[j]|

.

The

cost

of a path is the

maximum length of a jump

among all jumps in the path.

Return

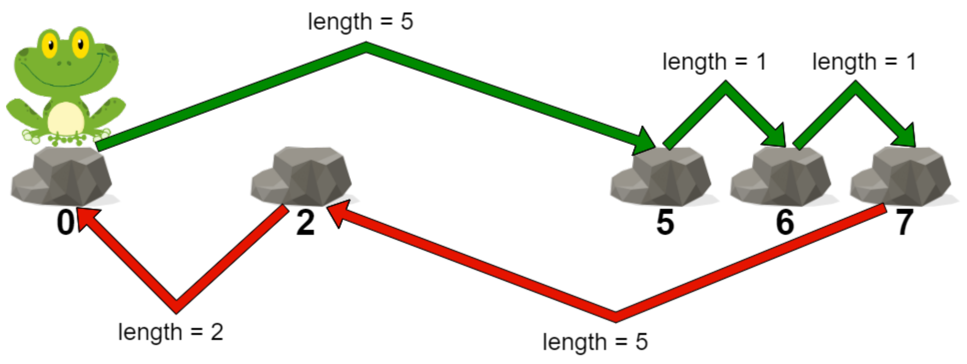
the

minimum

cost of a path for the frog

.

Example 1:



Input:

stones = [0,2,5,6,7]

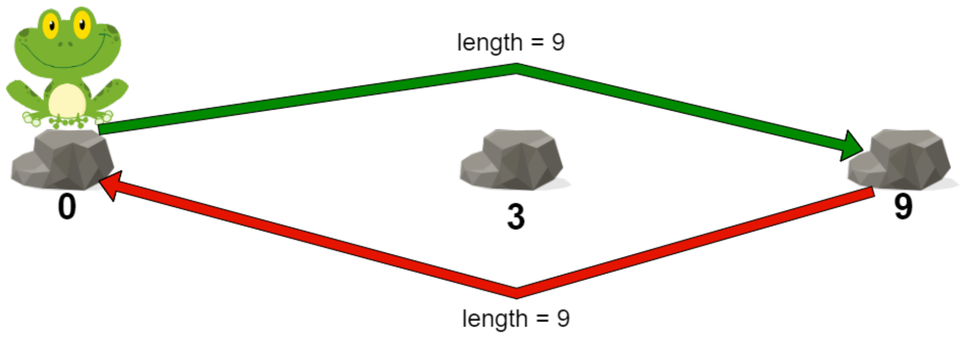
Output:

5

Explanation:

The above figure represents one of the optimal paths the frog can take. The cost of this path is 5, which is the maximum length of a jump. Since it is not possible to achieve a cost of less than 5, we return it.

Example 2:



Input:

stones = [0,3,9]

Output:

9

Explanation:

The frog can jump directly to the last stone and come back to the first stone. In this case, the length of each jump will be 9. The cost for the path will be max(9, 9) = 9. It can be shown that this is the minimum achievable cost.

Constraints:

2 <= stones.length <= 10

5

0 <= stones[i] <= 10

9

stones[0] == 0

stones

is sorted in a strictly increasing order.

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int maxJump(int[] stones) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function maxJump(stones: number[]): number {

};
```

C#:

```
public class Solution {
public int MaxJump(int[] stones) {

}
}
```

C:

```
int maxJump(int* stones, int stonesSize) {

}
```

Go:

```
func maxJump(stones []int) int {

}
```

Kotlin:

```
class Solution {
fun maxJump(stones: IntArray): Int {

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

```
class Solution {
int maxJump(List<int> stones) {

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (max-jump stones)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Frog Jump II
* Difficulty: Medium
* Tags: array, greedy, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
};
```

Java Solution:

```
/**
* Problem: Frog Jump II
* Difficulty: Medium
* Tags: array, greedy, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public int maxJump(int[] stones) {

}
}
```

Python3 Solution:

```
"""
Problem: Frog Jump II
Difficulty: Medium
Tags: array, greedy, sort, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Frog Jump II
* Difficulty: Medium
* Tags: array, greedy, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
```

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

};
```

TypeScript Solution:

```
/**
* Problem: Frog Jump II
* Difficulty: Medium
* Tags: array, greedy, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

function maxJump(stones: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Frog Jump II
* Difficulty: Medium
* Tags: array, greedy, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int MaxJump(int[] stones) {

}
}
```

C Solution:

```
/*
* Problem: Frog Jump II
* Difficulty: Medium
* Tags: array, greedy, sort, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int maxJump(int* stones, int stonesSize) {

}
```

Go Solution:

```
// Problem: Frog Jump II
// Difficulty: Medium
// Tags: array, greedy, sort, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

func maxJump(stones []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun maxJump(stones: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func maxJump(_ stones: [Int]) -> Int {
}
}
```

Rust Solution:

```
// Problem: Frog Jump II
// Difficulty: Medium
// Tags: array, greedy, sort, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

```
class Solution {
int maxJump(List<int> stones) {

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (max-jump stones)
(-> (listof exact-integer?) exact-integer?)
)
```