Problem 1046: Last Stone Weight

Problem Information

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

Problem Description

You are given an array of integers

stones

where

stones[i]

is the weight of the

i

th

stone.

We are playing a game with the stones. On each turn, we choose the

heaviest two stones

and smash them together. Suppose the heaviest two stones have weights

x

and

y

with

x <= y

. The result of this smash is:

If

x == y

, both stones are destroyed, and

If

x != y

, the stone of weight

x

is destroyed, and the stone of weight

y

has new weight

y - x

.

At the end of the game, there is

at most one

stone left.

Return

the weight of the last remaining stone

. If there are no stones left, return

0

.

Example 1:

Input:

stones = [2,7,4,1,8,1]

Output:

1

Explanation:

We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then, we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then, we combine 2 and 1 to get 1 so the array converts to [1,1,1] then, we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of the last stone.

Example 2:

Input:

stones = [1]

Output:

1

Constraints:

1 <= stones.length <= 30

1 <= stones[i] <= 1000

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

```
/**
* @param {number[]} stones
* @return {number}
*/
```

```
var lastStoneWeight = function(stones) {

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

```
}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (last-stone-weight stones)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Last Stone Weight
* Difficulty: Easy
* Tags: array, queue, heap
*
* 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 lastStoneWeight(vector<int>& stones) {

}
};
```

Java Solution:

```
/**
* Problem: Last Stone Weight
* Difficulty: Easy
* Tags: array, queue, heap
*
* 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 lastStoneWeight(int[] stones) {

}
}
```

Python3 Solution:

```
"""
Problem: Last Stone Weight
Difficulty: Easy
Tags: array, queue, heap

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 lastStoneWeight(self, stones: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Last Stone Weight
* Difficulty: Easy
* Tags: array, queue, heap
*
* 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 lastStoneWeight = function(stones) {

};
```

TypeScript Solution:

```
/**
* Problem: Last Stone Weight
* Difficulty: Easy
* Tags: array, queue, heap
*
* 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 lastStoneWeight(stones: number[]): number {
```

```
};
```

C# Solution:

```
/*
* Problem: Last Stone Weight
* Difficulty: Easy
* Tags: array, queue, heap
*
* 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 LastStoneWeight(int[] stones) {

}
}
```

C Solution:

```
/*
* Problem: Last Stone Weight
* Difficulty: Easy
* Tags: array, queue, heap
*
* 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 lastStoneWeight(int* stones, int stonesSize) {

}
```

Go Solution:

```
// Problem: Last Stone Weight
// Difficulty: Easy
```

```
// Tags: array, queue, heap
//
// 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 lastStoneWeight(stones []int) int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Last Stone Weight
// Difficulty: Easy
// Tags: array, queue, heap
//
// 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 last_stone_weight(stones: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

```
end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (last-stone-weight stones)
(-> (listof exact-integer?) exact-integer?)
)
```