Problem 2335: Minimum Amount of Time to Fill Cups

Problem Information

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

Problem Description

You have a water dispenser that can dispense cold, warm, and hot water. Every second, you can either fill up

2

cups with

different

types of water, or

1

cup of any type of water.

You are given a

0-indexed

integer array

amount

of length

3

where

amount[0]

,

amount[1]

, and

amount[2]

denote the number of cold, warm, and hot water cups you need to fill respectively. Return

the

minimum

number of seconds needed to fill up all the cups

.

Example 1:

Input:

amount = [1,4,2]

Output:

4

Explanation:

One way to fill up the cups is: Second 1: Fill up a cold cup and a warm cup. Second 2: Fill up a warm cup and a hot cup. Second 3: Fill up a warm cup and a hot cup. Second 4: Fill up a warm cup. It can be proven that 4 is the minimum number of seconds needed.

Example 2:

Input:

amount = [5,4,4]

Output:

7

Explanation:

One way to fill up the cups is: Second 1: Fill up a cold cup, and a hot cup. Second 2: Fill up a cold cup, and a warm cup. Second 3: Fill up a cold cup, and a warm cup. Second 4: Fill up a warm cup, and a hot cup. Second 5: Fill up a cold cup, and a hot cup. Second 6: Fill up a cold cup, and a warm cup. Second 7: Fill up a hot cup.

Example 3:

Input:

amount = [5,0,0]

Output:

5

Explanation:

Every second, we fill up a cold cup.

Constraints:

amount.length == 3

0 <= amount[i] <= 100

Code Snippets

C++:

```
class Solution {
public:
int fillCups(vector<int>& amount) {

}
};
```

Java:

```
class Solution {
public int fillCups(int[] amount) {

}
}
```

Python3:

```
class Solution:
def fillCups(self, amount: List[int]) -> int:
```

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function fillCups(amount: number[]): number {

};
```

C#:

```
public class Solution {
public int FillCups(int[] amount) {

}
}
```

C:

```
int fillCups(int* amount, int amountSize) {

}
```

Go:

```
func fillCups(amount []int) int {

}
```

Kotlin:

```
class Solution {
fun fillCups(amount: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func fillCups(_ amount: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn fill_cups(amount: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} amount
# @return {Integer}
def fill_cups(amount)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $amount
* @return Integer
*/
function fillCups($amount) {

}
}
```

Dart:

```
class Solution {
int fillCups(List<int> amount) {

}
}
```

Scala:

```
object Solution {
def fillCups(amount: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec fill_cups(amount :: [integer]) :: integer
def fill_cups(amount) do

end
end
```

Erlang:

```
-spec fill_cups(Amount :: [integer()]) -> integer().
fill_cups(Amount) ->
.
```

Racket:

```
(define/contract (fill-cups amount)
(-> (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Minimum Amount of Time to Fill Cups
* Difficulty: Easy
* Tags: array, greedy, sort, 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 fillCups(vector<int>& amount) {

}
};
```

Java Solution:

```
/**
* Problem: Minimum Amount of Time to Fill Cups
* Difficulty: Easy
* Tags: array, greedy, sort, 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 fillCups(int[] amount) {

}
}
```

Python3 Solution:

```
"""
Problem: Minimum Amount of Time to Fill Cups
Difficulty: Easy
Tags: array, greedy, sort, 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 fillCups(self, amount: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

```
"""
```

JavaScript Solution:

```
/**
* Problem: Minimum Amount of Time to Fill Cups
* Difficulty: Easy
* Tags: array, greedy, sort, 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[]} amount
* @return {number}
*/
var fillCups = function(amount) {

};
```

TypeScript Solution:

```
/**
* Problem: Minimum Amount of Time to Fill Cups
* Difficulty: Easy
* Tags: array, greedy, sort, 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 fillCups(amount: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Minimum Amount of Time to Fill Cups
* Difficulty: Easy
* Tags: array, greedy, sort, 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 FillCups(int[] amount) {

}
}
```

C Solution:

```
/*
* Problem: Minimum Amount of Time to Fill Cups
* Difficulty: Easy
* Tags: array, greedy, sort, 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 fillCups(int* amount, int amountSize) {

}
```

Go Solution:

```
// Problem: Minimum Amount of Time to Fill Cups
// Difficulty: Easy
// Tags: array, greedy, sort, 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 fillCups(amount []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun fillCups(amount: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func fillCups(_ amount: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Minimum Amount of Time to Fill Cups
// Difficulty: Easy
// Tags: array, greedy, sort, 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 fill_cups(amount: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} amount
# @return {Integer}
def fill_cups(amount)
```

```
end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $amount
* @return Integer
*/
function fillCups($amount) {

}
}
```

Dart Solution:

```
class Solution {
int fillCups(List<int> amount) {

}
}
```

Scala Solution:

```
object Solution {
def fillCups(amount: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec fill_cups(amount :: [integer]) :: integer
def fill_cups(amount) do

end
end
```

Erlang Solution:

```
-spec fill_cups(Amount :: [integer()]) -> integer().
fill_cups(Amount) ->
.
```

Racket Solution:

```
(define/contract (fill-cups amount)
(-> (listof exact-integer?) exact-integer?)
)
```