Problem 2105: Watering Plants II

Problem Information

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

Problem Description

Alice and Bob want to water

n

plants in their garden. The plants are arranged in a row and are labeled from

0

to

n - 1

from left to right where the

i

th

plant is located at

x = i

.

Each plant needs a specific amount of water. Alice and Bob have a watering can each,

initially full

. They water the plants in the following way:

Alice waters the plants in order from

left to right

, starting from the

0

th

plant. Bob waters the plants in order from

right to left

, starting from the

(n - 1)

th

plant. They begin watering the plants

simultaneously

.

It takes the same amount of time to water each plant regardless of how much water it needs.

Alice/Bob

must

water the plant if they have enough in their can to

fully

water it. Otherwise, they

first

refill their can (instantaneously) then water the plant.

In case both Alice and Bob reach the same plant, the one with

more

water currently in his/her watering can should water this plant. If they have the same amount of water, then Alice should water this plant.

Given a

0-indexed

integer array

plants

of

n

integers, where

plants[i]

is the amount of water the

i

th

plant needs, and two integers

capacityA

and

capacityB

representing the capacities of Alice's and Bob's watering cans respectively, return

the

number of times

they have to refill to water all the plants

.

Example 1:

Input:

plants = [2,2,3,3], capacityA = 5, capacityB = 5

Output:

1

Explanation:

- Initially, Alice and Bob have 5 units of water each in their watering cans. - Alice waters plant 0, Bob waters plant 3. - Alice and Bob now have 3 units and 2 units of water respectively. - Alice has enough water for plant 1, so she waters it. Bob does not have enough water for plant 2, so he refills his can then waters it. So, the total number of times they have to refill to water all the plants is 0 + 0 + 1 + 0 = 1.

Example 2:

Input:

plants = [2,2,3,3], capacityA = 3, capacityB = 4

Output:

2

Explanation:

- Initially, Alice and Bob have 3 units and 4 units of water in their watering cans respectively. - Alice waters plant 0, Bob waters plant 3. - Alice and Bob now have 1 unit of water each, and need to water plants 1 and 2 respectively. - Since neither of them have enough water for their current plants, they refill their cans and then water the plants. So, the total number of times they have to refill to water all the plants is 0 + 1 + 1 + 0 = 2.

Example 3:

Input:

plants = [5], capacityA = 10, capacityB = 8

Output:

0

Explanation:

- There is only one plant. - Alice's watering can has 10 units of water, whereas Bob's can has 8 units. Since Alice has more water in her can, she waters this plant. So, the total number of times they have to refill is 0.

Constraints:

n == plants.length

1 <= n <= 10

5

1 <= plants[i] <= 10

6

max(plants[i]) <= capacityA, capacityB <= 10

Code Snippets

C++:

```
class Solution {
public:
int minimumRefill(vector<int>& plants, int capacityA, int capacityB) {

}
};
```

Java:

```
class Solution {
public int minimumRefill(int[] plants, int capacityA, int capacityB) {

}
}
```

Python3:

```
class Solution:
def minimumRefill(self, plants: List[int], capacityA: int, capacityB: int) ->
int:
```

Python:

```
class Solution(object):
def minimumRefill(self, plants, capacityA, capacityB):
"""
:type plants: List[int]
:type capacityA: int
:type capacityB: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} plants
* @param {number} capacityA
* @param {number} capacityB
* @return {number}
*/
var minimumRefill = function(plants, capacityA, capacityB) {

};
```

TypeScript:

```
function minimumRefill(plants: number[], capacityA: number, capacityB:
number): number {

};
```

C#:

```
public class Solution {
public int MinimumRefill(int[] plants, int capacityA, int capacityB) {

}
}
```

C:

```
int minimumRefill(int* plants, int plantsSize, int capacityA, int capacityB)
{

}
```

Go:

```
func minimumRefill(plants []int, capacityA int, capacityB int) int {

}
```

Kotlin:

```
class Solution {
fun minimumRefill(plants: IntArray, capacityA: Int, capacityB: Int): Int {
```

```
}
}
```

Swift:

```
class Solution {
func minimumRefill(_ plants: [Int], _ capacityA: Int, _ capacityB: Int) ->
Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_refill(plants: Vec<i32>, capacity_a: i32, capacity_b: i32) ->
i32 {

}
}
```

Ruby:

```
# @param {Integer[]} plants
# @param {Integer} capacity_a
# @param {Integer} capacity_b
# @return {Integer}
def minimum_refill(plants, capacity_a, capacity_b)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $plants
* @param Integer $capacityA
* @param Integer $capacityB
* @return Integer
*/
function minimumRefill($plants, $capacityA, $capacityB) {
```

```
}
}
```

Dart:

```
class Solution {
int minimumRefill(List<int> plants, int capacityA, int capacityB) {

}
}
```

Scala:

```
object Solution {
def minimumRefill(plants: Array[Int], capacityA: Int, capacityB: Int): Int =
{

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_refill(plants :: [integer], capacity_a :: integer, capacity_b
:: integer) :: integer
def minimum_refill(plants, capacity_a, capacity_b) do

end
end
```

Erlang:

```
-spec minimum_refill(Plants :: [integer()], CapacityA :: integer(), CapacityB
:: integer()) -> integer().
minimum_refill(Plants, CapacityA, CapacityB) ->
.
```

Racket:

```
(define/contract (minimum-refill plants capacityA capacityB)
(-> (listof exact-integer?) exact-integer? exact-integer? exact-integer?)
```

```
)
```

Solutions

C++ Solution:

```
/*
* Problem: Watering Plants II
* Difficulty: Medium
* Tags: array
*
* 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 minimumRefill(vector<int>& plants, int capacityA, int capacityB) {

}
};
```

Java Solution:

```
/**
* Problem: Watering Plants II
* Difficulty: Medium
* Tags: array
*
* 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 minimumRefill(int[] plants, int capacityA, int capacityB) {

}
}
```

Python3 Solution:

```
"""
Problem: Watering Plants II
Difficulty: Medium
Tags: array

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 minimumRefill(self, plants: List[int], capacityA: int, capacityB: int) ->
int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumRefill(self, plants, capacityA, capacityB):
"""
:type plants: List[int]
:type capacityA: int
:type capacityB: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Watering Plants II
* Difficulty: Medium
* Tags: array
*
* 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[]} plants
* @param {number} capacityA
* @param {number} capacityB
* @return {number}
*/
var minimumRefill = function(plants, capacityA, capacityB) {

};
```

TypeScript Solution:

```
/**
* Problem: Watering Plants II
* Difficulty: Medium
* Tags: array
*
* 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 minimumRefill(plants: number[], capacityA: number, capacityB:
number): number {

};
```

C# Solution:

```
/*
* Problem: Watering Plants II
* Difficulty: Medium
* Tags: array
*
* 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 MinimumRefill(int[] plants, int capacityA, int capacityB) {
```

```
}
}
```

C Solution:

```
/*
* Problem: Watering Plants II
* Difficulty: Medium
* Tags: array
*
* 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 minimumRefill(int* plants, int plantsSize, int capacityA, int capacityB)
{

}
```

Go Solution:

```
// Problem: Watering Plants II
// Difficulty: Medium
// Tags: array
//
// 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 minimumRefill(plants []int, capacityA int, capacityB int) int {

}
```

Kotlin Solution:

```
class Solution {
fun minimumRefill(plants: IntArray, capacityA: Int, capacityB: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func minimumRefill(_ plants: [Int], _ capacityA: Int, _ capacityB: Int) ->
Int {

}
}
```

Rust Solution:

```
// Problem: Watering Plants II
// Difficulty: Medium
// Tags: array
//
// 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 minimum_refill(plants: Vec<i32>, capacity_a: i32, capacity_b: i32) ->
i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} plants
# @param {Integer} capacity_a
# @param {Integer} capacity_b
# @return {Integer}
def minimum_refill(plants, capacity_a, capacity_b)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $plants
```

```
* @param Integer $capacityA
* @param Integer $capacityB
* @return Integer
*/
function minimumRefill($plants, $capacityA, $capacityB) {

}
}
```

Dart Solution:

```
class Solution {
int minimumRefill(List<int> plants, int capacityA, int capacityB) {

}
}
```

Scala Solution:

```
object Solution {
def minimumRefill(plants: Array[Int], capacityA: Int, capacityB: Int): Int =
{

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_refill(plants :: [integer], capacity_a :: integer, capacity_b
:: integer) :: integer
def minimum_refill(plants, capacity_a, capacity_b) do

end
end
```

Erlang Solution:

```
-spec minimum_refill(Plants :: [integer()], CapacityA :: integer(), CapacityB
:: integer()) -> integer().
minimum_refill(Plants, CapacityA, CapacityB) ->
```

```
.
```

Racket Solution:

```
(define/contract (minimum-refill plants capacityA capacityB)
(-> (listof exact-integer?) exact-integer? exact-integer? exact-integer?)
)
```