Problem 3477: Fruits Into Baskets II

Problem Information

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

Problem Description

You are given two arrays of integers,

fruits

and

baskets

, each of length

n

, where

fruits[i]

represents the

quantity

of the

i

th

type of fruit, and

baskets[j]

represents the

capacity

of the

j

th

basket.

From left to right, place the fruits according to these rules:

Each fruit type must be placed in the

leftmost available basket

with a capacity

greater than or equal

to the quantity of that fruit type.

Each basket can hold

only one

type of fruit.

If a fruit type

cannot be placed

in any basket, it remains

unplaced

.

Return the number of fruit types that remain unplaced after all possible allocations are made.

Example 1:

Input:

fruits = [4,2,5], baskets = [3,5,4]

Output:

1

Explanation:

fruits[0] = 4

is placed in

baskets[1] = 5

.

fruits[1] = 2

is placed in

baskets[0] = 3

.

fruits[2] = 5

cannot be placed in

baskets[2] = 4

.

Since one fruit type remains unplaced, we return 1.

Example 2:

Input:

fruits = [3,6,1], baskets = [6,4,7]

Output:

0

Explanation:

fruits[0] = 3

is placed in

baskets[0] = 6

.

fruits[1] = 6

cannot be placed in

baskets[1] = 4

(insufficient capacity) but can be placed in the next available basket,

baskets[2] = 7

.

fruits[2] = 1

is placed in

baskets[1] = 4

.

Since all fruits are successfully placed, we return 0.

Constraints:

n == fruits.length == baskets.length

1 <= n <= 100

1 <= fruits[i], baskets[i] <= 1000

Code Snippets

C++:

```
class Solution {
public:
int numOfUnplacedFruits(vector<int>& fruits, vector<int>& baskets) {

}
};
```

Java:

```
class Solution {
public int numOfUnplacedFruits(int[] fruits, int[] baskets) {

}
}
```

Python3:

```
class Solution:
def numOfUnplacedFruits(self, fruits: List[int], baskets: List[int]) -> int:
```

Python:

```
class Solution(object):
def numOfUnplacedFruits(self, fruits, baskets):
"""
:type fruits: List[int]
:type baskets: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} fruits
* @param {number[]} baskets
* @return {number}
*/
var numOfUnplacedFruits = function(fruits, baskets) {

};
```

TypeScript:

```
function numOfUnplacedFruits(fruits: number[], baskets: number[]): number {

};
```

C#:

```
public class Solution {
public int NumOfUnplacedFruits(int[] fruits, int[] baskets) {

}
}
```

C:

```
int numOfUnplacedFruits(int* fruits, int fruitsSize, int* baskets, int
basketsSize) {

}
```

Go:

```
func numOfUnplacedFruits(fruits []int, baskets []int) int {

}
```

Kotlin:

```
class Solution {
fun numOfUnplacedFruits(fruits: IntArray, baskets: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func numOfUnplacedFruits(_ fruits: [Int], _ baskets: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn num_of_unplaced_fruits(fruits: Vec<i32>, baskets: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} fruits
# @param {Integer[]} baskets
# @return {Integer}
def num_of_unplaced_fruits(fruits, baskets)

end
```

PHP:

```
class Solution {
```

```
/**
* @param Integer[] $fruits
* @param Integer[] $baskets
* @return Integer
*/
function numOfUnplacedFruits($fruits, $baskets) {

}
}
```

Dart:

```
class Solution {
int numOfUnplacedFruits(List<int> fruits, List<int> baskets) {

}
}
```

Scala:

```
object Solution {
def numOfUnplacedFruits(fruits: Array[Int], baskets: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec num_of_unplaced_fruits(fruits :: [integer], baskets :: [integer]) ::
integer
def num_of_unplaced_fruits(fruits, baskets) do

end
end
```

Erlang:

```
-spec num_of_unplaced_fruits(Fruits :: [integer()], Baskets :: [integer()])
-> integer().
num_of_unplaced_fruits(Fruits, Baskets) ->
.
```

Racket:

```
(define/contract (num-of-unplaced-fruits fruits baskets)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Fruits Into Baskets II
* Difficulty: Easy
* Tags: array, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public:
int numOfUnplacedFruits(vector<int>& fruits, vector<int>& baskets) {

}
};
```

Java Solution:

```
/**
* Problem: Fruits Into Baskets II
* Difficulty: Easy
* Tags: array, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Solution {
public int numOfUnplacedFruits(int[] fruits, int[] baskets) {
}
}
```

Python3 Solution:

```
"""
Problem: Fruits Into Baskets II
Difficulty: Easy
Tags: array, tree, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

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

Python Solution:

```
class Solution(object):
def numOfUnplacedFruits(self, fruits, baskets):
"""
:type fruits: List[int]
:type baskets: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Fruits Into Baskets II
* Difficulty: Easy
* Tags: array, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
```

```
*/

/**
* @param {number[]} fruits
* @param {number[]} baskets
* @return {number}
*/
var numOfUnplacedFruits = function(fruits, baskets) {

};
```

TypeScript Solution:

```
/**
* Problem: Fruits Into Baskets II
* Difficulty: Easy
* Tags: array, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

function numOfUnplacedFruits(fruits: number[], baskets: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Fruits Into Baskets II
* Difficulty: Easy
* Tags: array, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Solution {
public int NumOfUnplacedFruits(int[] fruits, int[] baskets) {
}
}
```

C Solution:

```
/*
* Problem: Fruits Into Baskets II
* Difficulty: Easy
* Tags: array, tree, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

int numOfUnplacedFruits(int* fruits, int fruitsSize, int* baskets, int
basketsSize) {

}
```

Go Solution:

```
// Problem: Fruits Into Baskets II
// Difficulty: Easy
// Tags: array, tree, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

func numOfUnplacedFruits(fruits []int, baskets []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun numOfUnplacedFruits(fruits: IntArray, baskets: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func numOfUnplacedFruits(_ fruits: [Int], _ baskets: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Fruits Into Baskets II
// Difficulty: Easy
// Tags: array, tree, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

impl Solution {
pub fn num_of_unplaced_fruits(fruits: Vec<i32>, baskets: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} fruits
# @param {Integer[]} baskets
# @return {Integer}
def num_of_unplaced_fruits(fruits, baskets)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $fruits
* @param Integer[] $baskets
* @return Integer
*/
function numOfUnplacedFruits($fruits, $baskets) {

}
}
```

Dart Solution:

```
class Solution {
int numOfUnplacedFruits(List<int> fruits, List<int> baskets) {

}
}
```

Scala Solution:

```
object Solution {
def numOfUnplacedFruits(fruits: Array[Int], baskets: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec num_of_unplaced_fruits(fruits :: [integer], baskets :: [integer]) ::
integer
def num_of_unplaced_fruits(fruits, baskets) do

end
end
```

Erlang Solution:

```
-spec num_of_unplaced_fruits(Fruits :: [integer()], Baskets :: [integer()])
-> integer().
num_of_unplaced_fruits(Fruits, Baskets) ->
.
```

Racket Solution:

```
(define/contract (num-of-unplaced-fruits fruits baskets)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```