Problem 3074: Apple Redistribution into Boxes

Problem Information

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

Problem Description

You are given an array

apple

of size

n

and an array

capacity

of size

m

.

There are

n

packs where the

i

th

pack contains

apple[i]

apples. There are

m

boxes as well, and the

i

th

box has a capacity of

capacity[i]

apples.

Return

the

minimum

number of boxes you need to select to redistribute these

n

packs of apples into boxes

.

Note

that, apples from the same pack can be distributed into different boxes.

Example 1:

Input:

apple = [1,3,2], capacity = [4,3,1,5,2]

Output:

2

Explanation:

We will use boxes with capacities 4 and 5. It is possible to distribute the apples as the total capacity is greater than or equal to the total number of apples.

Example 2:

Input:

apple = [5,5,5], capacity = [2,4,2,7]

Output:

4

Explanation:

We will need to use all the boxes.

Constraints:

1 <= n == apple.length <= 50

1 <= m == capacity.length <= 50

1 <= apple[i], capacity[i] <= 50

The input is generated such that it's possible to redistribute packs of apples into boxes.

Code Snippets

C++:

```
class Solution {
public:
int minimumBoxes(vector<int>& apple, vector<int>& capacity) {

}
};
```

Java:

```
class Solution {
public int minimumBoxes(int[] apple, int[] capacity) {

}
}
```

Python3:

```
class Solution:
def minimumBoxes(self, apple: List[int], capacity: List[int]) -> int:
```

Python:

```
class Solution(object):
def minimumBoxes(self, apple, capacity):
"""
:type apple: List[int]
:type capacity: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} apple
* @param {number[]} capacity
* @return {number}
*/
```

```
var minimumBoxes = function(apple, capacity) {

};
```

TypeScript:

```
function minimumBoxes(apple: number[], capacity: number[]): number {

};
```

C#:

```
public class Solution {
public int MinimumBoxes(int[] apple, int[] capacity) {

}
}
```

C:

```
int minimumBoxes(int* apple, int appleSize, int* capacity, int capacitySize)
{

}
```

Go:

```
func minimumBoxes(apple []int, capacity []int) int {

}
```

Kotlin:

```
class Solution {
fun minimumBoxes(apple: IntArray, capacity: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func minimumBoxes(_ apple: [Int], _ capacity: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn minimum_boxes(apple: Vec<i32>, capacity: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} apple
# @param {Integer[]} capacity
# @return {Integer}
def minimum_boxes(apple, capacity)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $apple
* @param Integer[] $capacity
* @return Integer
*/
function minimumBoxes($apple, $capacity) {

}
}
```

Dart:

```
class Solution {
int minimumBoxes(List<int> apple, List<int> capacity) {

}

}
```

Scala:

```
object Solution {
def minimumBoxes(apple: Array[Int], capacity: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec minimum_boxes(apple :: [integer], capacity :: [integer]) :: integer
def minimum_boxes(apple, capacity) do

end
end
```

Erlang:

```
-spec minimum_boxes(Apple :: [integer()], Capacity :: [integer()]) ->
integer().
minimum_boxes(Apple, Capacity) ->
.
```

Racket:

```
(define/contract (minimum-boxes apple capacity)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Apple Redistribution into Boxes
* Difficulty: Easy
* Tags: array, greedy, sort
```

```
*
* 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 minimumBoxes(vector<int>& apple, vector<int>& capacity) {

}
};
```

Java Solution:

```
/**
* Problem: Apple Redistribution into Boxes
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 minimumBoxes(int[] apple, int[] capacity) {

}
}
```

Python3 Solution:

```
"""
Problem: Apple Redistribution into Boxes
Difficulty: Easy
Tags: array, greedy, sort

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 minimumBoxes(self, apple: List[int], capacity: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minimumBoxes(self, apple, capacity):
"""
:type apple: List[int]
:type capacity: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Apple Redistribution into Boxes
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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[]} apple
* @param {number[]} capacity
* @return {number}
*/
var minimumBoxes = function(apple, capacity) {

};
```

TypeScript Solution:

```
/**
* Problem: Apple Redistribution into Boxes
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 minimumBoxes(apple: number[], capacity: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Apple Redistribution into Boxes
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 MinimumBoxes(int[] apple, int[] capacity) {

}
}
```

C Solution:

```
/*
* Problem: Apple Redistribution into Boxes
* Difficulty: Easy
* Tags: array, greedy, sort
*
* 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 minimumBoxes(int* apple, int appleSize, int* capacity, int capacitySize)
{

}
```

Go Solution:

```
// Problem: Apple Redistribution into Boxes
// Difficulty: Easy
// Tags: array, greedy, sort
//
// 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 minimumBoxes(apple []int, capacity []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun minimumBoxes(apple: IntArray, capacity: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func minimumBoxes(_ apple: [Int], _ capacity: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Apple Redistribution into Boxes
// Difficulty: Easy
// Tags: array, greedy, sort
//
// 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_boxes(apple: Vec<i32>, capacity: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} apple
# @param {Integer[]} capacity
# @return {Integer}
def minimum_boxes(apple, capacity)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $apple
* @param Integer[] $capacity
* @return Integer
*/
function minimumBoxes($apple, $capacity) {

}
}
```

Dart Solution:

```
class Solution {
int minimumBoxes(List<int> apple, List<int> capacity) {

}

}
```

Scala Solution:

```
object Solution {
def minimumBoxes(apple: Array[Int], capacity: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec minimum_boxes(apple :: [integer], capacity :: [integer]) :: integer
def minimum_boxes(apple, capacity) do

end
end
```

Erlang Solution:

```
-spec minimum_boxes(Apple :: [integer()], Capacity :: [integer()]) ->
integer().
minimum_boxes(Apple, Capacity) ->
.
```

Racket Solution:

```
(define/contract (minimum-boxes apple capacity)
(-> (listof exact-integer?) (listof exact-integer?) exact-integer?)
)
```