Problem 1655: Distribute Repeating Integers

Problem Information

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

Problem Description

You are given an array of

n

integers,

nums

, where there are at most

50

unique values in the array. You are also given an array of

m

customer order quantities,

quantity

, where

quantity[i]

is the amount of integers the

i

th

customer ordered. Determine if it is possible to distribute

nums

such that:

The

i

th

customer gets

exactly

quantity[i]

integers,

The integers the

i

th

customer gets are

all equal

, and

Every customer is satisfied.

Return

true

if it is possible to distribute

nums

according to the above conditions

.

Example 1:

Input:

nums = [1,2,3,4], quantity = [2]

Output:

false

Explanation:

The 0

th

customer cannot be given two different integers.

Example 2:

Input:

nums = [1,2,3,3], quantity = [2]

Output:

true

Explanation:

The 0

th

customer is given [3,3]. The integers [1,2] are not used.

Example 3:

Input:

nums = [1,1,2,2], quantity = [2,2]

Output:

true

Explanation:

The 0

th

customer is given [1,1], and the 1st customer is given [2,2].

Constraints:

n == nums.length

1 <= n <= 10

5

1 <= nums[i] <= 1000

m == quantity.length

1 <= m <= 10

1 <= quantity[i] <= 10

5

There are at most

50

unique values in

nums

.

Code Snippets

C++:

```
class Solution {
public:
bool canDistribute(vector<int>& nums, vector<int>& quantity) {

}
};
```

Java:

```
class Solution {
public boolean canDistribute(int[] nums, int[] quantity) {

}
}
```

Python3:

```
class Solution:
def canDistribute(self, nums: List[int], quantity: List[int]) -> bool:
```

Python:

```
class Solution(object):
def canDistribute(self, nums, quantity):
"""
:type nums: List[int]
:type quantity: List[int]
:rtype: bool
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @param {number[]} quantity
* @return {boolean}
*/
var canDistribute = function(nums, quantity) {

};
```

TypeScript:

```
function canDistribute(nums: number[], quantity: number[]): boolean {

};
```

C#:

```
public class Solution {
public bool CanDistribute(int[] nums, int[] quantity) {

}
}
```

C:

```
bool canDistribute(int* nums, int numsSize, int* quantity, int quantitySize)
{

}
```

Go:

```
func canDistribute(nums []int, quantity []int) bool {

}
```

Kotlin:

```
class Solution {
fun canDistribute(nums: IntArray, quantity: IntArray): Boolean {

}
}
```

Swift:

```
class Solution {
func canDistribute(_ nums: [Int], _ quantity: [Int]) -> Bool {

}
}
```

Rust:

```
impl Solution {
pub fn can_distribute(nums: Vec<i32>, quantity: Vec<i32>) -> bool {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer[]} quantity
# @return {Boolean}
def can_distribute(nums, quantity)

end
```

PHP:

```
class Solution {

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

```
* @param Integer[] $quantity
* @return Boolean
*/
function canDistribute($nums, $quantity) {

}
}
```

Dart:

```
class Solution {
bool canDistribute(List<int> nums, List<int> quantity) {

}
}
```

Scala:

```
object Solution {
def canDistribute(nums: Array[Int], quantity: Array[Int]): Boolean = {

}
}
```

Elixir:

```
defmodule Solution do
@spec can_distribute(nums :: [integer], quantity :: [integer]) :: boolean
def can_distribute(nums, quantity) do

end
end
```

Erlang:

```
-spec can_distribute(Nums :: [integer()], Quantity :: [integer()]) ->
boolean().
can_distribute(Nums, Quantity) ->
.
```

Racket:

```
(define/contract (can-distribute nums quantity)
(-> (listof exact-integer?) (listof exact-integer?) boolean?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Distribute Repeating Integers
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
bool canDistribute(vector<int>& nums, vector<int>& quantity) {

}
};
```

Java Solution:

```
/**
* Problem: Distribute Repeating Integers
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public boolean canDistribute(int[] nums, int[] quantity) {

}
```

```
}
```

Python3 Solution:

```
"""
Problem: Distribute Repeating Integers
Difficulty: Hard
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def canDistribute(self, nums: List[int], quantity: List[int]) -> bool:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def canDistribute(self, nums, quantity):
"""
:type nums: List[int]
:type quantity: List[int]
:rtype: bool
"""
```

JavaScript Solution:

```
/**
* Problem: Distribute Repeating Integers
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/
```

```
/**
* @param {number[]} nums
* @param {number[]} quantity
* @return {boolean}
*/
var canDistribute = function(nums, quantity) {

};
```

TypeScript Solution:

```
/**
* Problem: Distribute Repeating Integers
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function canDistribute(nums: number[], quantity: number[]): boolean {

};
```

C# Solution:

```
/*
* Problem: Distribute Repeating Integers
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public bool CanDistribute(int[] nums, int[] quantity) {

}
```

```
}
```

C Solution:

```
/*
* Problem: Distribute Repeating Integers
* Difficulty: Hard
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

bool canDistribute(int* nums, int numsSize, int* quantity, int quantitySize)
{

}
```

Go Solution:

```
// Problem: Distribute Repeating Integers
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func canDistribute(nums []int, quantity []int) bool {

}
```

Kotlin Solution:

```
class Solution {
fun canDistribute(nums: IntArray, quantity: IntArray): Boolean {

}
}
```

Swift Solution:

```
class Solution {
func canDistribute(_ nums: [Int], _ quantity: [Int]) -> Bool {

}
}
```

Rust Solution:

```
// Problem: Distribute Repeating Integers
// Difficulty: Hard
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn can_distribute(nums: Vec<i32>, quantity: Vec<i32>) -> bool {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer[]} quantity
# @return {Boolean}
def can_distribute(nums, quantity)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer[] $quantity
* @return Boolean
*/
```

```
function canDistribute($nums, $quantity) {

}
}
```

Dart Solution:

```
class Solution {
bool canDistribute(List<int> nums, List<int> quantity) {

}
}
```

Scala Solution:

```
object Solution {
def canDistribute(nums: Array[Int], quantity: Array[Int]): Boolean = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec can_distribute(nums :: [integer], quantity :: [integer]) :: boolean
def can_distribute(nums, quantity) do

end
end
```

Erlang Solution:

```
-spec can_distribute(Nums :: [integer()], Quantity :: [integer()]) ->
boolean().
can_distribute(Nums, Quantity) ->
.
```

Racket Solution:

```
(define/contract (can-distribute nums quantity)
(-> (listof exact-integer?) (listof exact-integer?) boolean?)
```

```
)
```