Problem 3072: Distribute Elements Into Two Arrays II

Problem Information

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

Problem Description

You are given a

1-indexed

array of integers

nums

of length

n

.

We define a function

greaterCount

such that

greaterCount(arr, val)

returns the number of elements in

arr

that are

strictly greater

than

val

.

You need to distribute all the elements of

nums

between two arrays

arr1

and

arr2

using

n

operations. In the first operation, append

nums[1]

to

arr1

. In the second operation, append

nums[2]

to

arr2

. Afterwards, in the

i

th

operation:

If

greaterCount(arr1, nums[i]) > greaterCount(arr2, nums[i])

, append

nums[i]

to

arr1

.

If

greaterCount(arr1, nums[i]) < greaterCount(arr2, nums[i])

, append

nums[i]

to

arr2

.

If

greaterCount(arr1, nums[i]) == greaterCount(arr2, nums[i])

, append

nums[i]

to the array with a

lesser

number of elements.

If there is still a tie, append

nums[i]

to

arr1

.

The array

result

is formed by concatenating the arrays

arr1

and

arr2

. For example, if

arr1 == [1,2,3]

and

arr2 == [4,5,6]

, then

result = [1,2,3,4,5,6]

.

Return

the integer array

result

.

Example 1:

Input:

nums = [2,1,3,3]

Output:

[2,3,1,3]

Explanation:

After the first 2 operations, arr1 = [2] and arr2 = [1]. In the 3

rd

operation, the number of elements greater than 3 is zero in both arrays. Also, the lengths are equal, hence, append nums[3] to arr1. In the 4

th

operation, the number of elements greater than 3 is zero in both arrays. As the length of arr2 is lesser, hence, append nums[4] to arr2. After 4 operations, arr1 = [2,3] and arr2 = [1,3]. Hence, the array result formed by concatenation is [2,3,1,3].

Example 2:

Input:

nums = [5,14,3,1,2]

Output:

[5,3,1,2,14]

Explanation:

After the first 2 operations, arr1 = [5] and arr2 = [14]. In the 3

rd

operation, the number of elements greater than 3 is one in both arrays. Also, the lengths are equal, hence, append nums[3] to arr1. In the 4

th

operation, the number of elements greater than 1 is greater in arr1 than arr2 (2 > 1). Hence, append nums[4] to arr1. In the 5

th

operation, the number of elements greater than 2 is greater in arr1 than arr2 (2 > 1). Hence, append nums[5] to arr1. After 5 operations, arr1 = [5,3,1,2] and arr2 = [14]. Hence, the array result formed by concatenation is [5,3,1,2,14].

Example 3:

Input:

nums = [3,3,3,3]

Output:

[3,3,3,3]

Explanation:

At the end of 4 operations, arr1 = [3,3] and arr2 = [3,3]. Hence, the array result formed by concatenation is [3,3,3,3].

Constraints:

3 <= n <= 10

5

1 <= nums[i] <= 10

9

Code Snippets

C++:

```
class Solution {
public:
vector<int> resultArray(vector<int>& nums) {

}
};
```

Java:

```
class Solution {
public int[] resultArray(int[] nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function resultArray(nums: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] ResultArray(int[] nums) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
```

```
int* resultArray(int* nums, int numsSize, int* returnSize) {

}
```

Go:

```
func resultArray(nums []int) []int {

}
```

Kotlin:

```
class Solution {
fun resultArray(nums: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func resultArray(_ nums: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn result_array(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @return {Integer[]}
def result_array(nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @return Integer[]
*/
function resultArray($nums) {

}
}
```

Dart:

```
class Solution {
List<int> resultArray(List<int> nums) {

}
}
```

Scala:

```
object Solution {
def resultArray(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec result_array(nums :: [integer]) :: [integer]
def result_array(nums) do

end
end
```

Erlang:

```
-spec result_array(Nums :: [integer()]) -> [integer()].
result_array(Nums) ->
.
```

Racket:

```
(define/contract (result-array nums)
(-> (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Distribute Elements Into Two Arrays II
* Difficulty: Hard
* Tags: array, tree
*
* 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:
vector<int> resultArray(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Distribute Elements Into Two Arrays II
* Difficulty: Hard
* Tags: array, tree
*
* 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[] resultArray(int[] nums) {
```

```
}
}
```

Python3 Solution:

```
"""
Problem: Distribute Elements Into Two Arrays II
Difficulty: Hard
Tags: array, tree

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 resultArray(self, nums: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Distribute Elements Into Two Arrays II
* Difficulty: Hard
* Tags: array, tree
*
* 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[]} nums
* @return {number[]}
*/
var resultArray = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Distribute Elements Into Two Arrays II
* Difficulty: Hard
* Tags: array, tree
*
* 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 resultArray(nums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Distribute Elements Into Two Arrays II
* Difficulty: Hard
* Tags: array, tree
*
* 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[] ResultArray(int[] nums) {

}
```

```
}
```

C Solution:

```
/*
* Problem: Distribute Elements Into Two Arrays II
* Difficulty: Hard
* Tags: array, tree
*
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* resultArray(int* nums, int numsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Distribute Elements Into Two Arrays II
// Difficulty: Hard
// Tags: array, tree
//
// 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 resultArray(nums []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun resultArray(nums: IntArray): IntArray {

}
```

```
}
```

Swift Solution:

```
class Solution {
func resultArray(_ nums: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Distribute Elements Into Two Arrays II
// Difficulty: Hard
// Tags: array, tree
//
// 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 result_array(nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @return {Integer[]}
def result_array(nums)

end
```

PHP Solution:

```
class Solution {

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

```
*/
function resultArray($nums) {

}
}
```

Dart Solution:

```
class Solution {
List<int> resultArray(List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def resultArray(nums: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec result_array(nums :: [integer]) :: [integer]
def result_array(nums) do

end
end
```

Erlang Solution:

```
-spec result_array(Nums :: [integer()]) -> [integer()].
result_array(Nums) ->
.
```

Racket Solution:

```
(define/contract (result-array nums)
(-> (listof exact-integer?) (listof exact-integer?))
```

```
)
```