Problem 2191: Sort the Jumbled Numbers

Problem Information

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

Problem Description

You are given a

0-indexed

integer array

mapping

which represents the mapping rule of a shuffled decimal system.

mapping[i] = j

means digit

i

should be mapped to digit

j

in this system.

The

mapped value

of an integer is the new integer obtained by replacing each occurrence of digit

i

in the integer with

mapping[i]

for all

0 <= i <= 9

.

You are also given another integer array

nums

. Return

the array

nums

sorted in

non-decreasing

order based on the

mapped values

of its elements.

Notes:

Elements with the same mapped values should appear in the

same relative order

as in the input.

The elements of

nums

should only be sorted based on their mapped values and

not be replaced

by them.

Example 1:

Input:

mapping = [8,9,4,0,2,1,3,5,7,6], nums = [991,338,38]

Output:

[338,38,991]

Explanation:

Map the number 991 as follows: 1. mapping[9] = 6, so all occurrences of the digit 9 will become 6. 2. mapping[1] = 9, so all occurrences of the digit 1 will become 9. Therefore, the mapped value of 991 is 669. 338 maps to 007, or 7 after removing the leading zeros. 38 maps to 07, which is also 7 after removing leading zeros. Since 338 and 38 share the same mapped value, they should remain in the same relative order, so 338 comes before 38. Thus, the sorted array is [338,38,991].

Example 2:

Input:

mapping = [0,1,2,3,4,5,6,7,8,9], nums = [789,456,123]

Output:

[123,456,789]

Explanation:

789 maps to 789, 456 maps to 456, and 123 maps to 123. Thus, the sorted array is [123,456,789].

Constraints:

mapping.length == 10

0 <= mapping[i] <= 9

All the values of

mapping[i]

are

unique

.

1 <= nums.length <= 3 * 10

4

0 <= nums[i] < 10

9

Code Snippets

C++:

```
class Solution {
public:
vector<int> sortJumbled(vector<int>& mapping, vector<int>& nums) {
```

```
}
};
```

Java:

```
class Solution {
public int[] sortJumbled(int[] mapping, int[] nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

```
function sortJumbled(mapping: number[], nums: number[]): number[] {
```

```
};
```

C#:

```
public class Solution {
public int[] SortJumbled(int[] mapping, int[] nums) {

}
}
```

C:

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

}
```

Go:

```
func sortJumbled(mapping []int, nums []int) []int {

}
```

Kotlin:

```
class Solution {
fun sortJumbled(mapping: IntArray, nums: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func sortJumbled(_ mapping: [Int], _ nums: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn sort_jumbled(mapping: Vec<i32>, nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} mapping
# @param {Integer[]} nums
# @return {Integer[]}
def sort_jumbled(mapping, nums)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $mapping
* @param Integer[] $nums
* @return Integer[]
*/
function sortJumbled($mapping, $nums) {

}
}
```

Dart:

```
class Solution {
List<int> sortJumbled(List<int> mapping, List<int> nums) {

}
}
```

Scala:

```
object Solution {
def sortJumbled(mapping: Array[Int], nums: Array[Int]): Array[Int] = {
```

```
}
}
```

Elixir:

```
defmodule Solution do
@spec sort_jumbled(mapping :: [integer], nums :: [integer]) :: [integer]
def sort_jumbled(mapping, nums) do

end
end
```

Erlang:

```
-spec sort_jumbled(Mapping :: [integer()], Nums :: [integer()]) ->
[integer()].
sort_jumbled(Mapping, Nums) ->
.
```

Racket:

```
(define/contract (sort-jumbled mapping nums)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Sort the Jumbled Numbers
* Difficulty: Medium
* Tags: array, 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:
vector<int> sortJumbled(vector<int>& mapping, vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Sort the Jumbled Numbers
* Difficulty: Medium
* Tags: array, 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[] sortJumbled(int[] mapping, int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Sort the Jumbled Numbers
Difficulty: Medium
Tags: array, 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 sortJumbled(self, mapping: List[int], nums: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Sort the Jumbled Numbers
* Difficulty: Medium
* Tags: array, 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[]} mapping
* @param {number[]} nums
* @return {number[]}
*/
var sortJumbled = function(mapping, nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Sort the Jumbled Numbers
* Difficulty: Medium
* Tags: array, 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 sortJumbled(mapping: number[], nums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Sort the Jumbled Numbers
* Difficulty: Medium
* Tags: array, 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[] SortJumbled(int[] mapping, int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Sort the Jumbled Numbers
* Difficulty: Medium
* Tags: array, 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
*/

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

```
}
```

Go Solution:

```
// Problem: Sort the Jumbled Numbers
// Difficulty: Medium
// Tags: array, 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 sortJumbled(mapping []int, nums []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun sortJumbled(mapping: IntArray, nums: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func sortJumbled(_ mapping: [Int], _ nums: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Sort the Jumbled Numbers
// Difficulty: Medium
// Tags: array, 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 sort_jumbled(mapping: Vec<i32>, nums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} mapping
# @param {Integer[]} nums
# @return {Integer[]}
def sort_jumbled(mapping, nums)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $mapping
* @param Integer[] $nums
* @return Integer[]
*/
function sortJumbled($mapping, $nums) {

}
}
```

Dart Solution:

```
class Solution {
List<int> sortJumbled(List<int> mapping, List<int> nums) {

}
}
```

Scala Solution:

```
object Solution {
def sortJumbled(mapping: Array[Int], nums: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec sort_jumbled(mapping :: [integer], nums :: [integer]) :: [integer]
def sort_jumbled(mapping, nums) do

end
end
```

Erlang Solution:

```
-spec sort_jumbled(Mapping :: [integer()], Nums :: [integer()]) ->
[integer()].
sort_jumbled(Mapping, Nums) ->
.
```

Racket Solution:

```
(define/contract (sort-jumbled mapping nums)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?))
)
```