Problem 2032: Two Out of Three

Problem Information

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

Problem Description

Given three integer arrays

nums1

,

nums2

, and

nums3

, return

a

distinct

array containing all the values that are present in

at least two

out of the three arrays. You may return the values in

any

order

.

Example 1:

Input:

nums1 = [1,1,3,2], nums2 = [2,3], nums3 = [3]

Output:

[3,2]

Explanation:

The values that are present in at least two arrays are: - 3, in all three arrays. - 2, in nums1 and nums2.

Example 2:

Input:

nums1 = [3,1], nums2 = [2,3], nums3 = [1,2]

Output:

[2,3,1]

Explanation:

The values that are present in at least two arrays are: - 2, in nums2 and nums3. - 3, in nums1 and nums2. - 1, in nums1 and nums3.

Example 3:

Input:

nums1 = [1,2,2], nums2 = [4,3,3], nums3 = [5]

Output:

[]

Explanation:

No value is present in at least two arrays.

Constraints:

1 <= nums1.length, nums2.length, nums3.length <= 100

1 <= nums1[i], nums2[j], nums3[k] <= 100

Code Snippets

C++:

```
class Solution {
public:
vector<int> twoOutOfThree(vector<int>& nums1, vector<int>& nums2,
vector<int>& nums3) {

}
};
```

Java:

```
class Solution {
public List<Integer> twoOutOfThree(int[] nums1, int[] nums2, int[] nums3) {

}
}
```

Python3:

```
class Solution:
def twoOutOfThree(self, nums1: List[int], nums2: List[int], nums3: List[int])
-> List[int]:
```

Python:

```
class Solution(object):
def twoOutOfThree(self, nums1, nums2, nums3):
"""
:type nums1: List[int]
:type nums2: List[int]
:type nums3: List[int]
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} nums1
* @param {number[]} nums2
* @param {number[]} nums3
* @return {number[]}
*/
var twoOutOfThree = function(nums1, nums2, nums3) {

};
```

TypeScript:

```
function twoOutOfThree(nums1: number[], nums2: number[], nums3: number[]):
number[] {

};
```

C#:

```
public class Solution {
public IList<int> TwoOutOfThree(int[] nums1, int[] nums2, int[] nums3) {

}
}
```

C:

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

```
int* twoOutOfThree(int* nums1, int nums1Size, int* nums2, int nums2Size, int*
nums3, int nums3Size, int* returnSize) {

}
```

Go:

```
func twoOutOfThree(nums1 []int, nums2 []int, nums3 []int) []int {

}
```

Kotlin:

```
class Solution {
fun twoOutOfThree(nums1: IntArray, nums2: IntArray, nums3: IntArray):
List<Int> {

}
}
```

Swift:

```
class Solution {
func twoOutOfThree(_ nums1: [Int], _ nums2: [Int], _ nums3: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn two_out_of_three(nums1: Vec<i32>, nums2: Vec<i32>, nums3: Vec<i32>) ->
Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @param {Integer[]} nums3
```

```
# @return {Integer[]}
def two_out_of_three(nums1, nums2, nums3)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @param Integer[] $nums3
* @return Integer[]
*/
function twoOutOfThree($nums1, $nums2, $nums3) {

}
}
```

Dart:

```
class Solution {
List<int> twoOutOfThree(List<int> nums1, List<int> nums2, List<int> nums3) {

}
}
```

Scala:

```
object Solution {
def twoOutOfThree(nums1: Array[Int], nums2: Array[Int], nums3: Array[Int]):
List[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec two_out_of_three(nums1 :: [integer], nums2 :: [integer], nums3 ::
[integer]) :: [integer]
```

```
def two_out_of_three(nums1, nums2, nums3) do

end
end
```

Erlang:

```
-spec two_out_of_three(Nums1 :: [integer()], Nums2 :: [integer()], Nums3 ::
[integer()]) -> [integer()].
two_out_of_three(Nums1, Nums2, Nums3) ->
.
```

Racket:

```
(define/contract (two-out-of-three nums1 nums2 nums3)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
(listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Two Out of Three
* Difficulty: Easy
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<int> twoOutOfThree(vector<int>& nums1, vector<int>& nums2,
vector<int>& nums3) {

}
};
```

Java Solution:

```
/**
* Problem: Two Out of Three
* Difficulty: Easy
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public List<Integer> twoOutOfThree(int[] nums1, int[] nums2, int[] nums3) {

}
}
```

Python3 Solution:

```
"""
Problem: Two Out of Three
Difficulty: Easy
Tags: array, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def twoOutOfThree(self, nums1: List[int], nums2: List[int], nums3: List[int])
-> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def twoOutOfThree(self, nums1, nums2, nums3):
"""
:type nums1: List[int]
```

```
:type nums2: List[int]
:type nums3: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Two Out of Three
* Difficulty: Easy
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number[]} nums1
* @param {number[]} nums2
* @param {number[]} nums3
* @return {number[]}
*/
var twoOutOfThree = function(nums1, nums2, nums3) {

};
```

TypeScript Solution:

```
/**
* Problem: Two Out of Three
* Difficulty: Easy
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

function twoOutOfThree(nums1: number[], nums2: number[], nums3: number[]):
number[] {
```

```
};
```

C# Solution:

```
/*
* Problem: Two Out of Three
* Difficulty: Easy
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public IList<int> TwoOutOfThree(int[] nums1, int[] nums2, int[] nums3) {

}
}
```

C Solution:

```
/*
* Problem: Two Out of Three
* Difficulty: Easy
* Tags: array, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* twoOutOfThree(int* nums1, int nums1Size, int* nums2, int nums2Size, int*
nums3, int nums3Size, int* returnSize) {

}
```

Go Solution:

```
// Problem: Two Out of Three
// Difficulty: Easy
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func twoOutOfThree(nums1 []int, nums2 []int, nums3 []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun twoOutOfThree(nums1: IntArray, nums2: IntArray, nums3: IntArray):
List<Int> {

}
}
```

Swift Solution:

```
class Solution {
func twoOutOfThree(_ nums1: [Int], _ nums2: [Int], _ nums3: [Int]) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Two Out of Three
// Difficulty: Easy
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
```

```
pub fn two_out_of_three(nums1: Vec<i32>, nums2: Vec<i32>, nums3: Vec<i32>) ->
Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums1
# @param {Integer[]} nums2
# @param {Integer[]} nums3
# @return {Integer[]}
def two_out_of_three(nums1, nums2, nums3)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums1
* @param Integer[] $nums2
* @param Integer[] $nums3
* @return Integer[]
*/
function twoOutOfThree($nums1, $nums2, $nums3) {

}
}
```

Dart Solution:

```
class Solution {
List<int> twoOutOfThree(List<int> nums1, List<int> nums2, List<int> nums3) {

}
}
```

Scala Solution:

```
object Solution {
def twoOutOfThree(nums1: Array[Int], nums2: Array[Int], nums3: Array[Int]):
List[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec two_out_of_three(nums1 :: [integer], nums2 :: [integer], nums3 ::
[integer]) :: [integer]
def two_out_of_three(nums1, nums2, nums3) do

end
end
```

Erlang Solution:

```
-spec two_out_of_three(Nums1 :: [integer()], Nums2 :: [integer()], Nums3 ::
[integer()]) -> [integer()].
two_out_of_three(Nums1, Nums2, Nums3) ->
.
```

Racket Solution:

```
(define/contract (two-out-of-three nums1 nums2 nums3)
(-> (listof exact-integer?) (listof exact-integer?) (listof exact-integer?)
(listof exact-integer?))
)
```