Problem 2248: Intersection of Multiple Arrays

Problem Information

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

Problem Description

Given a 2D integer array

nums

where

nums[i]

is a non-empty array of

distinct

positive integers, return

the list of integers that are present in

each array

of

nums

sorted in

ascending order

.

Example 1:

Input:

nums = [[

3

,1,2,

4

,5],[1,2,

3

,

4

],[

3

,

4

,5,6]]

Output:

[3,4]

Explanation:

The only integers present in each of nums[0] = [

3

,1,2,

4

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

3

,

4

], and nums[2] = [

3

,

4

,5,6] are 3 and 4, so we return [3,4].

Example 2:

Input:

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

Output:

[]

Explanation:

There does not exist any integer present both in nums[0] and nums[1], so we return an empty list [].

Constraints:

1 <= nums.length <= 1000

1 <= sum(nums[i].length) <= 1000

1 <= nums[i][j] <= 1000

All the values of

nums[i]

are

unique

.

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public List<Integer> intersection(int[][] nums) {

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public IList<int> Intersection(int[][] nums) {

}
}
```

C:

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

```
}
```

Go:

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

}
```

Kotlin:

```
class Solution {
fun intersection(nums: Array<IntArray>): List<Int> {

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Intersection of Multiple Arrays
* Difficulty: Easy
* Tags: array, hash, sort
*
* 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> intersection(vector<vector<int>>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Intersection of Multiple Arrays
* Difficulty: Easy
* Tags: array, hash, sort
*
* 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> intersection(int[][] nums) {

}
```

```
}
```

Python3 Solution:

```
"""
Problem: Intersection of Multiple Arrays
Difficulty: Easy
Tags: array, hash, sort

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Intersection of Multiple Arrays
* Difficulty: Easy
* Tags: array, hash, sort
*
* 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[][]} nums
* @return {number[]}
*/
var intersection = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Intersection of Multiple Arrays
* Difficulty: Easy
* Tags: array, hash, sort
*
* 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 intersection(nums: number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Intersection of Multiple Arrays
* Difficulty: Easy
* Tags: array, hash, sort
*
* 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> Intersection(int[][] nums) {

}
}
```

C Solution:

```
/*
* Problem: Intersection of Multiple Arrays
* Difficulty: Easy
* Tags: array, hash, sort
*
* 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* intersection(int** nums, int numsSize, int* numsColSize, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Intersection of Multiple Arrays
// Difficulty: Easy
// Tags: array, hash, sort
//
// 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 intersection(nums [][]int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun intersection(nums: Array<IntArray>): List<Int> {

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Intersection of Multiple Arrays
// Difficulty: Easy
// Tags: array, hash, sort
//
// 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 intersection(nums: Vec<Vec<i32>>) -> Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

```
}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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