Problem 1424: Diagonal Traverse II

Problem Information

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

Problem Description

Given a 2D integer array

nums

, return

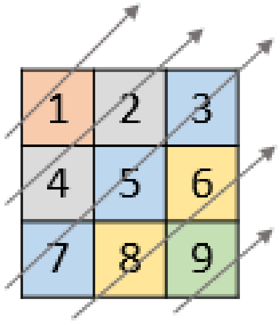
all elements of

nums

in diagonal order as shown in the below images

.

Example 1:



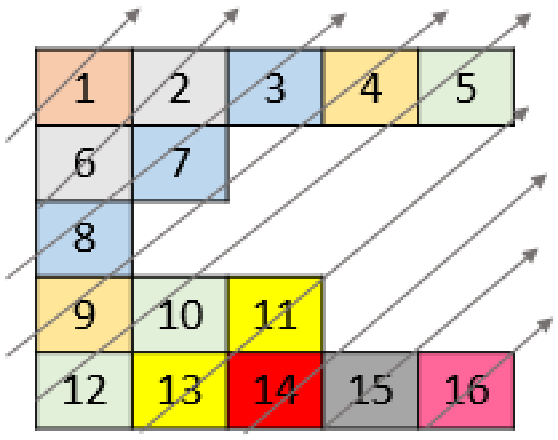
Input:

nums = [[1,2,3],[4,5,6],[7,8,9]]

Output:

[1,4,2,7,5,3,8,6,9]

Example 2:



Input:

nums = [[1,2,3,4,5],[6,7],[8],[9,10,11],[12,13,14,15,16]]

Output:

[1,6,2,8,7,3,9,4,12,10,5,13,11,14,15,16]

Constraints:

1 <= nums.length <= 10

5

1 <= nums[i].length <= 10

5

1 <= sum(nums[i].length) <= 10

5

1 <= nums[i][j] <= 10

5

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

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

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

```
(define/contract (find-diagonal-order nums)
(-> (listof (listof exact-integer?)) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Diagonal Traverse II
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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> findDiagonalOrder(vector<vector<int>>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Diagonal Traverse II
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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[] findDiagonalOrder(List<List<Integer>> nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Diagonal Traverse II
Difficulty: Medium
Tags: array, sort, queue, heap

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Diagonal Traverse II
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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[][]} nums
* @return {number[]}
*/
var findDiagonalOrder = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Diagonal Traverse II
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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 findDiagonalOrder(nums: number[][]): number[] {

};
```

C# Solution:

```
/*
* Problem: Diagonal Traverse II
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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[] FindDiagonalOrder(IList<IList<int>> nums) {

}
}
```

C Solution:

```
/*
* Problem: Diagonal Traverse II
* Difficulty: Medium
* Tags: array, sort, queue, heap
*
* 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* findDiagonalOrder(int** nums, int numsSize, int* numsColSize, int*
returnSize) {

}
```

Go Solution:

```
// Problem: Diagonal Traverse II
// Difficulty: Medium
// Tags: array, sort, queue, heap
//
// 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 findDiagonalOrder(nums [][]int) []int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Diagonal Traverse II
// Difficulty: Medium
// Tags: array, sort, queue, heap
//
// 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 find_diagonal_order(nums: Vec<Vec<i32>>) -> Vec<i32> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (find-diagonal-order nums)
(-> (listof (listof exact-integer?)) (listof exact-integer?))
)
```