Problem 47: Permutations II

Problem Information

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

Problem Description

Given a collection of numbers,

nums

, that might contain duplicates, return

all possible unique permutations

in any order

.

Example 1:

Input:

nums = [1,1,2]

Output:

[[1,1,2], [1,2,1], [2,1,1]]

Example 2:

Input:

nums = [1,2,3]

Output:

[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

Constraints:

1 <= nums.length <= 8

-10 <= nums[i] <= 10

Code Snippets

C++:

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

}
};
```

Java:

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

}
}
```

Python3:

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

Python:

```
class Solution(object):
def permuteUnique(self, nums):
```

```
"""
:type nums: List[int]
:rtype: List[List[int]]
"""
```

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

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

}
}
```

C:

```
/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** permuteUnique(int* nums, int numsSize, int* returnSize, int**
returnColumnSizes) {

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

/**
```

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

end
end
```

Erlang:

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: Permutations II
* 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<vector<int>> permuteUnique(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Permutations II
* 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 List<List<Integer>> permuteUnique(int[] nums) {

}
```

```
}
```

Python3 Solution:

```
"""
Problem: Permutations II
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 permuteUnique(self, nums: List[int]) -> List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Permutations II
* 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[]} nums
* @return {number[][]}
*/
var permuteUnique = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Permutations II
* 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 permuteUnique(nums: number[]): number[][] {

};
```

C# Solution:

```
/*
* Problem: Permutations II
* 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 IList<IList<int>> PermuteUnique(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Permutations II
* 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
*/

/**
* Return an array of arrays of size *returnSize.
* The sizes of the arrays are returned as *returnColumnSizes array.
* Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
*/
int** permuteUnique(int* nums, int numsSize, int* returnSize, int**
returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Permutations II
// 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 permuteUnique(nums []int) [][]int {

}
```

Kotlin Solution:

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

}
```

```
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Permutations II
// 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 permute_unique(nums: Vec<i32>) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
end
```

Erlang Solution:

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

Racket Solution:

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

```
)
```