Problem 90: Subsets II

Problem Information

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

Problem Description

Given an integer array

nums

that may contain duplicates, return

all possible

subsets

(the power set)

.

The solution set

must not

contain duplicate subsets. Return the solution in

any order

.

Example 1:

Input:

nums = [1,2,2]

Output:

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

Example 2:

Input:

nums = [0]

Output:

[[],[0]]

Constraints:

1 <= nums.length <= 10

-10 <= nums[i] <= 10

Code Snippets

C++:

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

}
};
```

Java:

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

```
}
}
```

Python3:

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

Python:

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

JavaScript:

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

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public IList<IList<int>> SubsetsWithDup(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** subsetsWithDup(int* nums, int numsSize, int* returnSize, int**
returnColumnSizes) {

}
```

Go:

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

}
```

Kotlin:

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

}
}
```

Swift:

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

}
}
```

Rust:

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

}
}
```

Ruby:

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

end
```

PHP:

```
class Solution {

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

}
}
```

Dart:

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

}
}
```

Scala:

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

}
}
```

Elixir:

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

```
end
end
```

Erlang:

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

Racket:

```
(define/contract (subsets-with-dup nums)
(-> (listof exact-integer?) (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Subsets II
* Difficulty: Medium
* Tags: array
*
* 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>> subsetsWithDup(vector<int>& nums) {

}
};
```

Java Solution:

```
/**
* Problem: Subsets II
```

```
* Difficulty: Medium
* Tags: array
*
* 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>> subsetsWithDup(int[] nums) {

}
}
```

Python3 Solution:

```
"""
Problem: Subsets II
Difficulty: Medium
Tags: array

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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: Subsets II
* Difficulty: Medium
* Tags: array
*
* 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 subsetsWithDup = function(nums) {

};
```

TypeScript Solution:

```
/**
* Problem: Subsets II
* Difficulty: Medium
* Tags: array
*
* 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 subsetsWithDup(nums: number[]): number[][] {

};
```

C# Solution:

```
/*
* Problem: Subsets II
* Difficulty: Medium
* Tags: array
*
* 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>> SubsetsWithDup(int[] nums) {

}
}
```

C Solution:

```
/*
* Problem: Subsets II
* Difficulty: Medium
* Tags: array
*
* 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** subsetsWithDup(int* nums, int numsSize, int* returnSize, int**
returnColumnSizes) {

}
```

Go Solution:

```
// Problem: Subsets II
// Difficulty: Medium
// Tags: array
//
// 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 subsetsWithDup(nums []int) [][]int {

}
```

Kotlin Solution:

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

}
}
```

Swift Solution:

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

}
}
```

Rust Solution:

```
// Problem: Subsets II
// Difficulty: Medium
// Tags: array
//
// 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 subsets_with_dup(nums: Vec<i32>) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

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

end
```

PHP Solution:

```
class Solution {

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

}
}
```

Dart Solution:

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

}
}
```

Scala Solution:

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

}
}
```

Elixir Solution:

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

end
```

```
end
```

Erlang Solution:

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

Racket Solution:

```
(define/contract (subsets-with-dup nums)
(-> (listof exact-integer?) (listof (listof exact-integer?)))
)
```