Problem 1982: Find Array Given Subset Sums

Problem Information

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

Problem Description

You are given an integer

n

representing the length of an unknown array that you are trying to recover. You are also given an array

sums

containing the values of all

2

n

subset sums

of the unknown array (in no particular order).

Return

the array

ans

of length

n

representing the unknown array. If

multiple

answers exist, return

any

of them

.

An array

sub

is a

subset

of an array

arr

if

sub

can be obtained from

arr

by deleting some (possibly zero or all) elements of

arr

. The sum of the elements in

sub

is one possible

subset sum

of

arr

. The sum of an empty array is considered to be

0

.

Note:

Test cases are generated such that there will

always

be at least one correct answer.

Example 1:

Input:

n = 3, sums = [-3,-2,-1,0,0,1,2,3]

Output:

[1,2,-3]

Explanation:

[1,2,-3] is able to achieve the given subset sums: - []: sum is 0 - [1]: sum is 1 - [2]: sum is 2 - [1,2]: sum is 3 - [-3]: sum is -3 - [1,-3]: sum is -2 - [2,-3]: sum is -1 - [1,2,-3]: sum is 0 Note that any permutation of [1,2,-3] and also any permutation of [-1,-2,3] will also be accepted.

Example 2:

Input:

n = 2, sums = [0,0,0,0]

Output:

[0,0]

Explanation:

The only correct answer is [0,0].

Example 3:

Input:

n = 4, sums = [0,0,5,5,4,-1,4,9,9,-1,4,3,4,8,3,8]

Output:

[0,-1,4,5]

Explanation:

[0,-1,4,5] is able to achieve the given subset sums.

Constraints:

1 <= n <= 15

sums.length == 2

n

-10

4

<= sums[i] <= 10

4

Code Snippets

C++:

```
class Solution {
public:
vector<int> recoverArray(int n, vector<int>& sums) {

}
};
```

Java:

```
class Solution {
public int[] recoverArray(int n, int[] sums) {

}
}
```

Python3:

```
class Solution:
def recoverArray(self, n: int, sums: List[int]) -> List[int]:
```

Python:

```
class Solution(object):
def recoverArray(self, n, sums):
"""
:type n: int
:type sums: List[int]
:rtype: List[int]
```

```
"""
```

JavaScript:

```
/**
* @param {number} n
* @param {number[]} sums
* @return {number[]}
*/
var recoverArray = function(n, sums) {

};
```

TypeScript:

```
function recoverArray(n: number, sums: number[]): number[] {

};
```

C#:

```
public class Solution {
public int[] RecoverArray(int n, int[] sums) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* recoverArray(int n, int* sums, int sumsSize, int* returnSize) {

}
```

Go:

```
func recoverArray(n int, sums []int) []int {

}
```

Kotlin:

```
class Solution {
fun recoverArray(n: Int, sums: IntArray): IntArray {

}
}
```

Swift:

```
class Solution {
func recoverArray(_ n: Int, _ sums: [Int]) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn recover_array(n: i32, sums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer} n
# @param {Integer[]} sums
# @return {Integer[]}
def recover_array(n, sums)

end
```

PHP:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $sums
* @return Integer[]
*/
function recoverArray($n, $sums) {
```

```
}
}
```

Dart:

```
class Solution {
List<int> recoverArray(int n, List<int> sums) {

}
}
```

Scala:

```
object Solution {
def recoverArray(n: Int, sums: Array[Int]): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec recover_array(n :: integer, sums :: [integer]) :: [integer]
def recover_array(n, sums) do

end
end
```

Erlang:

```
-spec recover_array(N :: integer(), Sums :: [integer()]) -> [integer()].
recover_array(N, Sums) ->
.
```

Racket:

```
(define/contract (recover-array n sums)
(-> exact-integer? (listof exact-integer?) (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Find Array Given Subset Sums
* Difficulty: Hard
* 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<int> recoverArray(int n, vector<int>& sums) {

}
};
```

Java Solution:

```
/**
* Problem: Find Array Given Subset Sums
* Difficulty: Hard
* 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 int[] recoverArray(int n, int[] sums) {

}
}
```

Python3 Solution:

```
"""
Problem: Find Array Given Subset Sums
```

```
Difficulty: Hard
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 recoverArray(self, n: int, sums: List[int]) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def recoverArray(self, n, sums):
"""
:type n: int
:type sums: List[int]
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Find Array Given Subset Sums
* Difficulty: Hard
* 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} n
* @param {number[]} sums
* @return {number[]}
*/
var recoverArray = function(n, sums) {
```

```
};
```

TypeScript Solution:

```
/**
* Problem: Find Array Given Subset Sums
* Difficulty: Hard
* 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 recoverArray(n: number, sums: number[]): number[] {

};
```

C# Solution:

```
/*
* Problem: Find Array Given Subset Sums
* Difficulty: Hard
* 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 int[] RecoverArray(int n, int[] sums) {

}
}
```

C Solution:

```
/*
* Problem: Find Array Given Subset Sums
```

```
* Difficulty: Hard
* 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
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* recoverArray(int n, int* sums, int sumsSize, int* returnSize) {

}
```

Go Solution:

```
// Problem: Find Array Given Subset Sums
// Difficulty: Hard
// 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 recoverArray(n int, sums []int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun recoverArray(n: Int, sums: IntArray): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func recoverArray(_ n: Int, _ sums: [Int]) -> [Int] {
```

```
}
}
```

Rust Solution:

```
// Problem: Find Array Given Subset Sums
// Difficulty: Hard
// 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 recover_array(n: i32, sums: Vec<i32>) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer} n
# @param {Integer[]} sums
# @return {Integer[]}
def recover_array(n, sums)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $n
* @param Integer[] $sums
* @return Integer[]
*/
function recoverArray($n, $sums) {

}
```

```
}
```

Dart Solution:

```
class Solution {
List<int> recoverArray(int n, List<int> sums) {

}
}
```

Scala Solution:

```
object Solution {
def recoverArray(n: Int, sums: Array[Int]): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec recover_array(n :: integer, sums :: [integer]) :: [integer]
def recover_array(n, sums) do

end
end
```

Erlang Solution:

```
-spec recover_array(N :: integer(), Sums :: [integer()]) -> [integer()].
recover_array(N, Sums) ->
.
```

Racket Solution:

```
(define/contract (recover-array n sums)
(-> exact-integer? (listof exact-integer?) (listof exact-integer?))
)
```