Problem 216: Combination Sum III

Problem Information

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

Problem Description

Find all valid combinations of

k

numbers that sum up to

n

such that the following conditions are true:

Only numbers

1

through

9

are used.

Each number is used

at most once

.

Return

a list of all possible valid combinations

. The list must not contain the same combination twice, and the combinations may be returned in any order.

Example 1:

Input:

k = 3, n = 7

Output:

[[1,2,4]]

Explanation:

1 + 2 + 4 = 7 There are no other valid combinations.

Example 2:

Input:

k = 3, n = 9

Output:

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

Explanation:

1 + 2 + 6 = 9 1 + 3 + 5 = 9 2 + 3 + 4 = 9 There are no other valid combinations.

Example 3:

Input:

k = 4, n = 1

Output:

[]

Explanation:

There are no valid combinations. Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and since 10 > 1, there are no valid combination.

Constraints:

2 <= k <= 9

1 <= n <= 60

Code Snippets

C++:

```
class Solution {
public:
vector<vector<int>> combinationSum3(int k, int n) {

}
};
```

Java:

```
class Solution {
public List<List<Integer>> combinationSum3(int k, int n) {

}
}
```

Python3:

```
class Solution:
def combinationSum3(self, k: int, n: int) -> List[List[int]]:
```

Python:

```
class Solution(object):
def combinationSum3(self, k, n):
"""
:type k: int
:type n: int
:rtype: List[List[int]]
"""
```

JavaScript:

```
/**
* @param {number} k
* @param {number} n
* @return {number[][]}
*/
var combinationSum3 = function(k, n) {

};
```

TypeScript:

```
function combinationSum3(k: number, n: number): number[][] {

};
```

C#:

```
public class Solution {
public IList<IList<int>> CombinationSum3(int k, int n) {

}
}
```

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** combinationSum3(int k, int n, int* returnSize, int** returnColumnSizes)
{

}
```

Go:

```
func combinationSum3(k int, n int) [][]int {

}
```

Kotlin:

```
class Solution {
fun combinationSum3(k: Int, n: Int): List<List<Int>> {

}
}
```

Swift:

```
class Solution {
func combinationSum3(_ k: Int, _ n: Int) -> [[Int]] {

}
}
```

Rust:

```
impl Solution {
pub fn combination_sum3(k: i32, n: i32) -> Vec<Vec<i32>> {

}
}
```

Ruby:

```
# @param {Integer} k
# @param {Integer} n
# @return {Integer[][]}
def combination_sum3(k, n)
```

```
end
```

PHP:

```
class Solution {

/**
* @param Integer $k
* @param Integer $n
* @return Integer[][]
*/
function combinationSum3($k, $n) {

}
}
```

Dart:

```
class Solution {
List<List<int>> combinationSum3(int k, int n) {

}
}
```

Scala:

```
object Solution {
def combinationSum3(k: Int, n: Int): List[List[Int]] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec combination_sum3(k :: integer, n :: integer) :: [[integer]]
def combination_sum3(k, n) do

end
end
```

Erlang:

```
-spec combination_sum3(K :: integer(), N :: integer()) -> [[integer()]].
combination_sum3(K, N) ->
.
```

Racket:

```
(define/contract (combination-sum3 k n)
(-> exact-integer? exact-integer? (listof (listof exact-integer?)))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Combination Sum III
* 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>> combinationSum3(int k, int n) {

}
};
```

Java Solution:

```
/**
* Problem: Combination Sum III
* 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>> combinationSum3(int k, int n) {

}
}
```

Python3 Solution:

```
"""
Problem: Combination Sum III
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 combinationSum3(self, k: int, n: int) -> List[List[int]]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def combinationSum3(self, k, n):
"""
:type k: int
:type n: int
:rtype: List[List[int]]
"""
```

JavaScript Solution:

```
/**
* Problem: Combination Sum III
* 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} k
* @param {number} n
* @return {number[][]}
*/
var combinationSum3 = function(k, n) {

};
```

TypeScript Solution:

```
/**
* Problem: Combination Sum III
* 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 combinationSum3(k: number, n: number): number[][] {

};
```

C# Solution:

```
/*
* Problem: Combination Sum III
* 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>> CombinationSum3(int k, int n) {

}
}
```

C Solution:

```
/*
* Problem: Combination Sum III
* 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** combinationSum3(int k, int n, int* returnSize, int** returnColumnSizes)
{

}
```

Go Solution:

```
// Problem: Combination Sum III
// 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 combinationSum3(k int, n int) [][]int {

}
```

Kotlin Solution:

```
class Solution {
fun combinationSum3(k: Int, n: Int): List<List<Int>> {

}
}
```

Swift Solution:

```
class Solution {
func combinationSum3(_ k: Int, _ n: Int) -> [[Int]] {

}
}
```

Rust Solution:

```
// Problem: Combination Sum III
// 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 combination_sum3(k: i32, n: i32) -> Vec<Vec<i32>> {

}
}
```

Ruby Solution:

```
# @param {Integer} k
# @param {Integer} n
# @return {Integer[][]}
```

```
def combination_sum3(k, n)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $k
* @param Integer $n
* @return Integer[][]
*/
function combinationSum3($k, $n) {

}
}
```

Dart Solution:

```
class Solution {
List<List<int>> combinationSum3(int k, int n) {

}
}
```

Scala Solution:

```
object Solution {
def combinationSum3(k: Int, n: Int): List[List[Int]] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec combination_sum3(k :: integer, n :: integer) :: [[integer]]
def combination_sum3(k, n) do

end
```

```


end
```

Erlang Solution:

```
-spec combination_sum3(K :: integer(), N :: integer()) -> [[integer()]].
combination_sum3(K, N) ->
.
```

Racket Solution:

```
(define/contract (combination-sum3 k n)
(-> exact-integer? exact-integer? (listof (listof exact-integer?)))
)
```