Problem 518: Coin Change II

Problem Information

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

Problem Description

You are given an integer array

coins

representing coins of different denominations and an integer

amount

representing a total amount of money.

Return

the number of combinations that make up that amount

. If that amount of money cannot be made up by any combination of the coins, return

0

.

You may assume that you have an infinite number of each kind of coin.

The answer is

guaranteed

to fit into a signed

32-bit

integer.

Example 1:

Input:

amount = 5, coins = [1,2,5]

Output:

4

Explanation:

there are four ways to make up the amount: 5=5 5=2+2+1 5=2+1+1+1 5=1+1+1+1+1

Example 2:

Input:

amount = 3, coins = [2]

Output:

0

Explanation:

the amount of 3 cannot be made up just with coins of 2.

Example 3:

Input:

amount = 10, coins = [10]

Output:

1

Constraints:

1 <= coins.length <= 300

1 <= coins[i] <= 5000

All the values of

coins

are

unique

.

0 <= amount <= 5000

Code Snippets

C++:

```
class Solution {
public:
int change(int amount, vector<int>& coins) {

}
};
```

Java:

```
class Solution {
public int change(int amount, int[] coins) {

}
```

```
}
```

Python3:

```
class Solution:
def change(self, amount: int, coins: List[int]) -> int:
```

Python:

```
class Solution(object):
def change(self, amount, coins):
"""
:type amount: int
:type coins: List[int]
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number} amount
* @param {number[]} coins
* @return {number}
*/
var change = function(amount, coins) {

};
```

TypeScript:

```
function change(amount: number, coins: number[]): number {

};
```

C#:

```
public class Solution {
public int Change(int amount, int[] coins) {

}
}
```

C:

```
int change(int amount, int* coins, int coinsSize) {

}
```

Go:

```
func change(amount int, coins []int) int {

}
```

Kotlin:

```
class Solution {
fun change(amount: Int, coins: IntArray): Int {

}
}
```

Swift:

```
class Solution {
func change(_ amount: Int, _ coins: [Int]) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn change(amount: i32, coins: Vec<i32>) -> i32 {

}
}
```

Ruby:

```
# @param {Integer} amount
# @param {Integer[]} coins
# @return {Integer}
def change(amount, coins)
```

```
end
```

PHP:

```
class Solution {

/**
* @param Integer $amount
* @param Integer[] $coins
* @return Integer
*/
function change($amount, $coins) {

}
}
```

Dart:

```
class Solution {
int change(int amount, List<int> coins) {

}
}
```

Scala:

```
object Solution {
def change(amount: Int, coins: Array[Int]): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec change(amount :: integer, coins :: [integer]) :: integer
def change(amount, coins) do

end
end
```

Erlang:

```
-spec change(Amount :: integer(), Coins :: [integer()]) -> integer().
change(Amount, Coins) ->
.
```

Racket:

```
(define/contract (change amount coins)
(-> exact-integer? (listof exact-integer?) exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Coin Change II
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
int change(int amount, vector<int>& coins) {

}
};
```

Java Solution:

```
/**
* Problem: Coin Change II
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

class Solution {
public int change(int amount, int[] coins) {

}
}
```

Python3 Solution:

```
"""
Problem: Coin Change II
Difficulty: Medium
Tags: array, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def change(self, amount: int, coins: List[int]) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def change(self, amount, coins):
"""
:type amount: int
:type coins: List[int]
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Coin Change II
* Difficulty: Medium
* Tags: array, dp
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number} amount
* @param {number[]} coins
* @return {number}
*/
var change = function(amount, coins) {

};
```

TypeScript Solution:

```
/**
* Problem: Coin Change II
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function change(amount: number, coins: number[]): number {

};
```

C# Solution:

```
/*
* Problem: Coin Change II
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
```

```
*/

public class Solution {
public int Change(int amount, int[] coins) {

}
}
```

C Solution:

```
/*
* Problem: Coin Change II
* Difficulty: Medium
* Tags: array, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

int change(int amount, int* coins, int coinsSize) {

}
```

Go Solution:

```
// Problem: Coin Change II
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func change(amount int, coins []int) int {

}
```

Kotlin Solution:

```
class Solution {
fun change(amount: Int, coins: IntArray): Int {

}
}
```

Swift Solution:

```
class Solution {
func change(_ amount: Int, _ coins: [Int]) -> Int {

}
}
```

Rust Solution:

```
// Problem: Coin Change II
// Difficulty: Medium
// Tags: array, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn change(amount: i32, coins: Vec<i32>) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer} amount
# @param {Integer[]} coins
# @return {Integer}
def change(amount, coins)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer $amount
* @param Integer[] $coins
* @return Integer
*/
function change($amount, $coins) {

}
}
```

Dart Solution:

```
class Solution {
int change(int amount, List<int> coins) {

}
}
```

Scala Solution:

```
object Solution {
def change(amount: Int, coins: Array[Int]): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec change(amount :: integer, coins :: [integer]) :: integer
def change(amount, coins) do

end
end
```

Erlang Solution:

```
-spec change(Amount :: integer(), Coins :: [integer()]) -> integer().
change(Amount, Coins) ->
.
```

Racket Solution:

```
(define/contract (change amount coins)
(-> exact-integer? (listof exact-integer?) exact-integer?)
)
```