518. Coin Change II **Dynamic Programming** Medium Array

# **Problem Description**

unlimited supply of each coin denomination in the array coins. The answer needs to be such that it can fit within a 32-bit signed integer, effectively suggesting that the solution does not need to handle extremely large numbers that exceed this size.

The problem provides an array called coins which contains integers representing the denominations of different coins. In

addition, you are given an integer amount which signifies the total amount of money you need to create using these coins. The

question is to find out the total number of distinct combinations of coins that can add up to exactly that amount. If it is not

possible to get to that amount using the given coins, the function should return 0. Notably, it is assumed that you have an

Intuition

The solution requires a methodical approach to count the combinations without having to consider each one explicitly, which

distinct combinations of coins can sum up to the value x.

complex problems by breaking them down into simpler subproblems, solving each subproblem just once, and storing their solutions – ideally, using a memory-based data structure. The intuition for the solution arises from realizing that the problem resembles the classic "Complete Knapsack Problem". With the

would be inefficient. This is where <u>dynamic programming</u> becomes useful. Dynamic programming is a strategy for solving

knapsack problem, imagine you have a bag (knapsack) and you aim to fill it with items (in this case, coins) up to a certain weight limit (the amount), and you are interested in counting the number of ways to reach exactly that limit. We use a list dp to store the number of ways to reach every amount from 0 to amount. So, dp[x] will indicate how many

an amount of 0, there is exactly one way to reach it – by choosing no coins. This means dp[0] is initialized to 1. Iterate over each coin in coins, considering it as a potential candidate for making up the amount. For each coin, update the

Initialize the dp list with zeros since initially, we have no coin, so there are zero ways to reach any amount except for 0. For

dp list. For each value from the coin's denomination up to amount, increment dp[j] by dp[j - coin]. This represents that the

number of combinations for an amount j includes all the combinations we had for amount (j - coin) plus this coin.

- By filling the dp list iteratively and using the previously computed values, we build up the solution without redundant calculations and eventually dp[amount] gives us the total number of combinations to form amount using the given denominations.
- take a detailed look at each step that is performed in the provided code: A <u>dynamic programming</u> array dp is initiated with a length of one more than the <u>amount</u>, because we need to store ways to

reach amounts ranging from 0 to amount inclusively. Every entry in dp is set to 0 to start.

The implementation of the solution is grounded in the <u>dynamic programming</u> approach. To better understand the solution, let's

## To establish our base case, dp[0] is set to 1, because there is always exactly one way to reach an amount of 0 — by not

for coin in coins:

# loop body

to reach j - coin.

larger problem.

**Example Walkthrough** 

dp = [0, 0, 0, 0, 0, 0]

Iterate over each coin in coins:

thus, is returned as the final answer:

dp = [0] \* (amount + 1)

Solution Approach

using any coins. dp[0] = 1

- The primary loop iterates over each coin in the coins array. This loop is necessary to consider each coin denomination for making up different amounts.
- Within this loop, a nested loop runs from coin (the current coin's value) up to amount + 1. This loop updates dp[j] where j

is the current target amount being considered. It is important we start at coin because that is the smallest amount that can

be made with the current coin denomination, and previous amounts would have been evaluated already with the other coins.

With each iteration of the inner loop, dp[j] is increased by dp[j - coin]. The reasoning behind this is the essence of

dynamic programming where we use previously computed results to construct the current result. In this aspect, if dp[j -

coin] represents the number of ways to reach j - coin amount, then we are effectively saying if you add the current coin

to all those combinations, you now reach j. This way we are adding to the count of reaching j with all the ways that existed

- for j in range(coin, amount + 1): dp[j] += dp[j - coin]
- return dp[-1] This dynamic programming table (dp array) allows us to avoid re-calculating combinations for smaller amounts, making the algorithm efficient and scalable for large inputs. By relying on the iterative addition of combinations, the algorithm effectively

builds the solution from the ground up, solving smaller sub-problems before piecing them together to yield the solution for the

Let's illustrate the solution approach with a small example. Suppose we're given coins = [1, 2, 5] and amount = 5. We want to

Initialize the dynamic programming array dp: Create a dp array of size amount + 1, which in this case is 6 because our

find the total number of distinct combinations that can add up to the amount.

stored in dp) to the number of ways to make the current amount using this coin.

For coin 2, we start at amount 2 and update the dp array similarly.

target amount is 5. Initially, it will look like this after initialization:

Following these steps, dp[amount] eventually holds the total number of combinations that can be used to reach the amount, and

Establish the base case: Since there's only one way to reach 0 (using no coins), we set dp[0] to 1: dp = [1, 0, 0, 0, 0, 0]

Start with the first coin 1. For each amount from 1 to 5 (amount + 1), we add the number of combinations without using the coin (already)

### dp = [1, 1+dp[0], 1+dp[1], 1+dp[2], 1+dp[3], 1+dp[4]]After iteration: dp = [1, 1, 1, 1, 1, 1]

For coin = 5:

• 1 + 1 + 1 + 1 (five 1's)

• 1 + 2 + 2 (one 1 and two 2's)

Solution Implementation

dp = [0] \* (amount + 1)

for coin in coins:

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

# There is 1 way to make change for 0 amount: use no coins

# each value from 0 up to the given `amount`.

# than or equal to the current `coin`.

# value of the current coin.

for current amount in range(coin, amount + 1):

dp[current\_amount] += dp[current\_amount - coin]

# Return the last element in dp which contains the number of ways

# to make change for the original `amount` using the given `coins`.

// dp array to store the number of ways to make change for each amount

// There is 1 way to make change for the amount zero, that is to choose no coins

// Return the total number of ways to make change for the specified amount

# Iterate over each coin in coins list

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

int[] dp = new int[amount + 1];

// Iterate over each type of coin

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

waysToMakeChange[0] = 1;

// Iterate over each coin

for (let coin of coins) {

dp = [0] \* (amount + 1)

for coin in coins:

return dp[-1]

Time and Space Complexity

dp[0] = 1

let waysToMakeChange = new Array(amount + 1).fill(0);

# each value from 0 up to the given `amount`.

# than or equal to the current `coin`.

# value of the current coin.

for current amount in range(coin, amount + 1):

dp[current\_amount] += dp[current\_amount - coin]

# Return the last element in dp which contains the number of wavs

# to make change for the original `amount` using the given `coins`.

# Iterate over each coin in coins list

# There is 1 way to make change for 0 amount: use no coins

// There is 1 way to make change for amount 0, which is to use no coins

for (let currentAmount = coin; currentAmount <= amount; ++currentAmount) {</pre>

waysToMakeChange[currentAmount] += waysToMakeChange[currentAmount - coin];

return waysToMakeChange.pop(); // Return the last element of the waysToMakeChange array

# Initializing a list `dp` to store the number of ways to make change for

# For each `coin` value, update `dp` for all amounts that are greater

# The number of ways to create the current amount includes the number

# of ways to create the amount that is the current amount minus the

for (int coin : coins) {

• 5 (one 5)

return dp[-1]

class Solution:

dp[0] = 1

• 1 + 1 + 1 + 2 (three 1's and one 2)

dp = [1, 1, 2, 2, 3, 3+dp[0]]

After iteration: dp = [1, 1, 2, 2, 3, 4]

For coin = 1:

For coin = 2: dp = [1, 1, 1+dp[0], 1+dp[1], 1+dp[2], 1+dp[3]]After iteration: dp = [1, 1, 2, 2, 3, 3]

So, there are 4 distinct combinations that can add up to 5 using the denominations [1, 2, 5]. These combinations are:

**Result:** The final dp array represents the number of ways to make each amount up to 5. dp[5] is our answer because it's the

number of combinations that make up the amount 5. dp = [1, 1, 2, 2, 3, 4]

In Python, the final result would be obtained by returning the last element of the dp array:

# Initializing a list `dp` to store the number of ways to make change for

# For each `coin` value, update `dp` for all amounts that are greater

# The number of ways to create the current amount includes the number

# of ways to create the amount that is the current amount minus the

Lastly, for 5, we start at amount 5, as it is the only value that can be updated by a coin of denomination 5.

**Python** # Importing typing module to use the List type annotation. from typing import List

#### // Update the dp array for all amounts that can be reached with the current coin for (int currentAmount = coin; currentAmount <= amount; currentAmount++) {</pre> // The number of ways to make change for currentAmount includes the number of ways // to make change for (currentAmount - coin value) dp[currentAmount] += dp[currentAmount - coin];

return dp[amount];

dp[0] = 1;

return dp[-1]

Java

C++

public:

class Solution {

class Solution {

// The function "change" computes the number of ways to make up the amount // with the given set of coin denominations. int change(int amount, vector<int>& coins) { // dp is the dvnamic programming table where dp[i] will store // the number of ways to make up the amount i. vector<int> dp(amount + 1, 0); // There is one way to make amount 0, which is not using any coins. dp[0] = 1;// Iterate over each type of coin. for (int coin : coins) { // For each coin, update the dp table for all amounts from coin to 'amount'. for (int currentAmount = coin; currentAmount <= amount; ++currentAmount) {</pre> // The number of ways to make up currentAmount includes the number of ways // to make (currentAmount - coin), as we can add the current coin // to those combinations to get currentAmount. dp[currentAmount] += dp[currentAmount - coin]; // Return the total number of ways to make up the original amount. return dp[amount]; **}**; **TypeScript** 

// Create a dp array to store the number of ways to make change for each amount from 0 to amount

// For each coin, update the ways to make change for amounts greater than or equal to the coin value

// to make change for the amount that remains after using this coin (currentAmount - coin)

// The number of ways to make change for the current amount is increased by the number of ways

// After filling the dp array, the last element holds the number of ways to make change for the original amount

#### # Importing typing module to use the List type annotation. from typing import List class Solution: def change(self, amount: int, coins: List[int]) -> int:

The given code is a dynamic programming solution that calculates the number of ways to make up a certain amount using the given denominations of coins. **Time Complexity:** 

# • The inner loop runs for each value from coin up to amount. In the worst case, this will run from 1 to amount, which we represent as m.

For each combination of a coin and an amount, the algorithm performs a constant amount of work by updating the dp array. Therefore, the total number of operations is proportional to the number of times the inner loop runs times the number of coins,

To determine the time complexity, we need to consider the two nested loops in the code:

• The outer loop runs once for each coin in the coins list. Let's assume the number of coins is n.

which is 0(n\*m). So, the time complexity is O(n\*m) where n is the number of coins, and m is the amount.

structure, the dp array, which has a size of amount + 1. No other data structures depend on n or m.

**Space Complexity:** 

The space complexity is determined by the size of the data structures used in the code. Here, there is one primary data

Therefore, the space complexity is O(m), where m is the amount.